

# Microarchitectural Characterization of a High-Performance Text Search Library

Rosalita Rosenberg

## Abstract—

Search engines are ubiquitous in our society, supporting some of our most popular applications and activities as we use them for research, social media interactions, and e-commerce. For example, Twitter and LinkedIn use the popular search engine Lucene to support user queries. Many studies try to increase Lucene's efficiency by optimising its indexing and query evaluation functionalities. Unfortunately, few studies use low-level metrics to quantify microarchitectural bottlenecks, so may not optimise the parts of Lucene that would most reduce inefficiency. We use Intel's (2014) top-down analysis framework to characterise the microarchitectural performance of Lucene's index and query evaluation functionalities. This allows us to identify possible inefficiencies and thus suggest optimisations that reduce the most significant bottlenecks. Our key findings are: 1) While indexing is relatively efficient (45% of slots retired), backend related delays stall about 31% of pipeline slots. 2) Frontend and backend related delays reduce Analyzers' efficiency, with each stalling between 30% and 40% of pipeline slots. 3) The merge process is highly inefficient, with backend processes stalling 65% of pipeline slots. 4) Query evaluation is highly efficient, with between 45% and 75% of pipeline slots being retired. 5) Query evaluation's efficiency depends on the type of query being evaluated. Frontend, backend, and bad speculation are all significant sources of inefficiency, as they can all stall at least 22% of slots.

By identifying these bottlenecks, we can suggest optimisations that would most reduce Lucene's inefficiency. For example, we advise future work to focus on optimising merging, instruction decoding, and data decompression.



## 1 INTRODUCTION

Information retrieval (IR) permeates many aspects of daily life. It is one of the foundations of social media, networking, and e-commerce platforms, and is used to locate web pages and information on the internet [1]. For example, the search engine library Lucene is used to index and search documents, information contained within documents, and/or document metadata for specific terms [2]. Many popular applications, such as Twitter and LinkedIn, incorporate Lucene into their products [1].

The time taken to evaluate a user query has large implications for the service's user retention, with Amazon finding that it loses revenue due to even a 100ms delay. Thus, search engine providers have strong motivation to identify and eliminate potential inefficiencies to improve performance [1].

Lucene's two major functionalities are its indexing and query evaluation operations. During indexing, Lucene stores text in an inverted index, either creating new index segments or updating existing entries. To evaluate user queries, Lucene can then search through the inverted index to find the most relevant results.

There are a variety of books and blogs that provide developers with tools to tune Lucene's performance [2]. Additionally, a variety of academic articles suggest different optimisations to Lucene's algorithms and memory structure, as we describe in the next section [1]. The majority of academic papers written about Lucene use high-level metrics, such as execution time and throughput, to characterise potential bottlenecks. However, Lucene's operations are complex and can be stalled by many possible bottlenecks. These high-level metrics often do not provide sufficient detail to be able to pinpoint Lucene's microarchitectural bottlenecks. In this report, we identify potential microarchitectural bottlenecks in Lucene's index creation and query evaluation functions using Intel's (2014) top-down analysis framework. This enables us to focus optimisations in areas that will most increase Lucene's efficiency.

The key findings of this report are:

- The indexing process is relatively efficient, retiring 45% of slots. Backend stalls are the largest source of inefficiency, affecting 31% of pipeline slots.
- Frontend and backend stalls are the most significant sources of inefficiency for Lucene Analyzers. Frontend stalls affect between 31% and 36% of pipeline slots and backend processes affect between 32% and 40% of pipeline slots.
- Merging is a major source of inefficiency. During merging, backend processes stall 65% of pipeline slots.
- Query evaluation is efficient. It is more efficient for single-term queries than multi-term queries. Single-term query workloads retire between 52% and 75% of pipeline slots while multi-term query workloads only retire between 45% and 50% of pipeline slots.
- During query evaluation, frontend processes stall 4% to 24% of slots and backend processes stall 13% to 22% of slots. Bad speculation has a larger impact on multi-term queries than single-term queries (11% to 23% versus 2% to 8%).

As discussed in depth later in this report, we can hypothesise about the causes of these microarchitectural bottlenecks.

- We believe cache misses cause a large proportion of backend stalls.
- Backend stalls may increase with Analyzer complexity because an increased number of filters need to access tokens from memory.
- We suggest that frontend stalls are likely due to bottlenecks in the instruction decoding process.
- We hypothesise that checking for a document's presence in the index or delete file, as well as decompression, causes branch misprediction and thus bad speculation.

Using our findings and hypotheses about Lucene's microarchitectural bottlenecks, we can recommend optimisations that would most reduce Lucene's inefficiencies:

- Larger data caches may resolve the high proportion of backend stalls in the indexing process.
- A larger instruction cache may reduce the incidence of decoder bottlenecks.
- Using the simplest possible Analyzer may reduce backend stalls.
- We identify merging as an optimisation target. Implementations should strongly consider limiting merge operations to reduce backend stalls.
- Given query workloads’ different associated microarchitectural bottlenecks, we suggest implementing query specific management and optimisation.
- We encourage future studies to focus on index decompression as an optimisation target.

## 2 RELATED WORK

Lucene’s popularity and high-performance makes it a good model for indexing and query evaluation. Many investigations explore how these functionalities can be made more efficient, such as through optimising memory utilisation, indexing, and/or query evaluation.

For example, Akram et al. (2023) delve into how Lucene uses memory [1]. In this work, they discuss how DRAM versus non-volatile memory affect the efficiency of memory accesses. They further distinguish the effect of these different memory technologies on compressed versus non-compressed indexes. They find that decompression is a major source of inefficiency in the search process, corroborating our results [1].

A fair number of studies also explore Lucene’s query retrieval process. For example, studies show how latency scales with the number of query terms [3], how considering term position can improve the scoring of search results [4], and how PFOR compression can be optimised [5]. More specifically, Hadjilabrou et al. (2019) discuss how index search operations are cache-friendly, as they do not read much data from memory and rely heavily on caches. For example, they find that last-level cache misses per thousand instructions is about 0.77. Instead, they observe that decompression of blocks fetched into cache from memory strongly influences the efficiency of the index search operation [3].

Many of these analyses use high-level metrics to determine possible bottlenecks, such as time taken to perform indexing or query evaluation [3]. With a few exceptions, to our knowledge there are few studies that use a top-down analysis of microarchitectural processes to identify inefficiencies in Lucene’s index and query evaluation functionalities. If the report does use performance counters, such as in Hadjilabrou et al. (2019), the metrics take into account backend and frontend stalls, but do not explicitly measure bad speculation or retiring, which give important information about potentially unexpected bottlenecks. We explore Lucene’s true microarchitectural bottlenecks using performance counters to give a fuller picture of how optimisations can be tailored to best improve Lucene’s performance.

## 3 BACKGROUND

Lucene stores data in base units called ‘documents’. Documents can be an email or a web page. ‘Terms’ refer to the information stored in documents. [2].

### 3.1 Indexing

#### 3.1.1 The Inverted Index

Using a data structure called an index, Lucene associates terms with documents [2]. More specifically, Lucene uses an inverted

index, which is a data structure consisting of key-value pairs, where the term is the key and the associated ‘postings list’ is the value. These postings lists consist of pointers, called document IDs, to every document that contains the term. The inverted index structure enables rapid search, as Lucene can go directly to documents containing the query terms rather than searching every document for those terms. Postings lists are generally stored in a postings file, and can contain additional meta-data, including the frequency with which the term appears in the document [6], [7], [1]. Table 1 gives an example of an inverted index.

TABLE 1

Inverted index showing how each term (key) in the index is associated with a postings list (value). Each postings list consists of document IDs and term frequency within the associated documents.

| Term      | Document ID | Term Frequency |
|-----------|-------------|----------------|
| delicious | 1, 7, 23    | 9, 4, 6        |
| apple     | 5, 7, 18    | 3, 5, 2        |
| green     | 2           | 7              |

#### 3.1.2 Tokenization

The first step in the indexing process is called ‘tokenisation’. In this step, tokenizers divide text into small indexing units, called ‘tokens’. These tokens can be classified as a specific type of ‘field’, such as ‘author’s name’ or ‘URL’ [2]. The terms in the inverted index are tokenised fields [6].

#### 3.1.3 Analyzers

Lucene gives developers the choice of a diverse range of Analyzers, each of which tokenises text differently. Apache (2023) and Zobel and Moffat (2006) describe how Lucene gives the option to choose Analyzers that [8], [9]:

- Filter out stop (“common”) words, such as ‘the’ and ‘a’, that increase the size of the index without adding value.
- Engage in stemming, where the tokeniser reduces a word to its grammatical root to avoid repetition. This reduces the size of the index without significantly impacting its meaningfulness.
- Engage in synonym expansion, where the term’s synonyms are also evaluated for a better search outcome.
- Convert text to lowercase to decrease search’s case sensitivity.
- Remove character markings to improve search.

When an Analyzer is given a query or data that needs to be indexed, it passes the text to a specific tokeniser. The tokeniser takes the text as an input and returns tokens. For example, Lucene’s Unicode Whitespace Tokenizer divides text into tokens at whitespaces. In contrast, its Letter Tokenizer divides text into tokens at non-letters, producing strings of maximal adjacent letters [8]. These tokens can then be used as input to one or more token filters, which can modify, add to, and/or delete from the input tokens [2]. For instance, Lucene’s Lower-Case Filter changes all input tokens into lower case, its Type Token Filter deletes tokens that are listed in a set of blocked types, and its Stop Filter removes stop words. The Analyser may employ one or more token filters. For example, Lucene’s Simple Analyzer uses the Letter Tokenizer and Lower-Case Filter. The Standard Analyzer is more complex, as it uses the Lower-Case Filter as well as the Standard Tokenizer and Stop Filter. The Standard Analyzer can even recognise URLs and emails [8].

### 3.1.4 Forming the Index from Segments

Rather than having one very large, inverted index, Lucene splits the index into one or more immutable segments, which can be considered ‘mini’ indexes. Documents added to the index are added to a specific segment, which is buffered in memory until it is flushed to disk [10].

When presenting the results of a search, Lucene combines the results from individual segments. Multiple segments are occasionally merged to prevent the searcher from iterating through many segments, which is inefficient. Merges are triggered by some condition, such as having sufficiently large segments. This condition is described in Lucene’s merge policy [10]. The default merge policy is the Tiered Merge Policy, which can merge non-adjacent segments, allowing the most efficient merge to occur. Lucene’s merge scheduler executes merges in a different thread, preventing merges from interfering with the indexing process [11].

When a document is deleted from an index, it is marked as deleted in a specific delete file. Only when segments are merged are these deleted documents removed from the index and their bytes reclaimed. Therefore, sometimes merging reduces the size of the index [10].

## 3.2 Compression

Lucene uses compression techniques to reduce the index’s size by between 85 and 90%. This reduces the amount of memory needed to store the index, increasing efficiency [1]. This report uses the compression techniques described in the Lucene95codec.

Document IDs are compressed using a mixture of delta and frame-of-reference (FOR) encoding. Rather than storing each document’s full ID, delta encoding stores the difference between document IDs. Given that the difference between IDs is often smaller than the IDs themselves, this encoding scheme decreases the bits needed to represent integers. Then, FOR encoding separates the postings lists of integer document IDs into smaller blocks of 128 integers, where integers are represented by the number of bits of the largest integer in the block. Compression efficiency decreases with block size [1]. Lucene further compresses these blocks using bit packing, where multiple integers that can be represented with a small number of bits are packed into one or more bytes. For example, four 2-bit integers can be represented in one byte, saving three bytes of memory [12]. Block size can also be reduced using Patched Frame of Reference (PFOR) encoding, which encodes larger integer values separately at the end of the list [1].

Lucene also uses variable length byte (VLB) encoding, where integers are represented by the minimum number of bytes possible. In VLB encoding, 7 bits per byte are used to represent the integer, with the remaining bit used as a continuation flag. This flag indicates whether the next byte represents a continuation of the current integer or represents the start of a new integer. Lucene uses VLB encoding to represent term positions [1].

Despite its benefits, compression creates additional computation costs when the index needs to be decompressed, such as during search and data retrieval [1].

## 3.3 Searching

Lucene searches the inverted index to find specific term/s. It then presents the documents containing those term/s [2].

### 3.3.1 Queries

When a user enters search terms into the user interface, the string is parsed into a Lucene query. This query consists of terms. Lucene defines two types of term: single words and phrases, which are multiple words contained within double quotes. Boolean operators, such as AND, join terms together into a query, where AND queries present the intersection of the relevant postings lists. The same Analyser that splits the database into terms is used on the query string [8]. For example, the Analyser may transform the terms of the query into lowercase [2].

Frequently used queries are cached in Lucene’s Query Cache, which operates under a ‘least recently used’ (LRU) eviction policy. This policy specifies that, when the Query Cache is full, the least recently used query is evicted to make room for the new query [8]. The cache stores the document IDs returned by the query.

### 3.3.2 Dictionary Search

To access information about the term/s in a query, Lucene decompresses the relevant postings list/s [1]. For single-term queries, search time is proportional to the length of the postings lists across the index’s segments [9]. If the query has multiple terms connected via Boolean operators, Lucene uses set operations to identify relevant documents [1].

To evaluate AND queries, Lucene uses multi-level skip lists. This data structure has a base linked list containing all postings, with each higher level containing fewer postings. Lucene can use the highest level of the skip list to determine the likelihood of the relevant document being contained in the list. This allows Lucene to complete searches without decompressing every postings list, increasing efficiency [1].

### 3.3.3 Scoring Searches

Two widely known search models are the pure Boolean and vector space models. The pure Boolean model searches for documents that are perfect matches for the query, presenting the results in an unordered, unscored list. In contrast, the vector space model represents documents and queries as vectors, calculating the document’s relevance using the distance between the two vectors. Lucene uses a mixture of the two models to score documents based on their relevancy to the search query, with higher scores given to documents with greater similarity to the query [2]. The frequency of the query terms in the document, normalised by the document’s size, influences the document’s score [1].

A scored search returns the top scoring documents, preventing the searcher from inefficiently retrieving every matching document in the index, as users typically only access the first few documents [2].

## 3.4 Top-Down Analysis

### 3.4.1 Execution Pipeline Overview

Modern out-of-order central processing units (CPUs) have an execution pipeline that can be divided into frontend and backend components. The frontend fetches instructions from memory and translates them into micro-operations ( $\mu$ ops), which are then passed to the backend. The backend schedules, executes, and retires  $\mu$ ops, using a combination of the issue queue and re-order buffer to ensure retirement occurs in original program order.  $\mu$ op execution can occur out of program order. Ideally, a modern, superscalar, out-of-order (OOO) 4-issue CPU executes 4  $\mu$ ops per clock cycle. Analysing the behaviour of these  $\mu$ ops gives developers information about a workload’s performance [13].

### 3.4.2 Performance Analysis

Modern OOO CPUs use a variety of techniques to keep their execution pipelines full and thus execute as many instructions per cycle (IPC) as possible. Given that bottlenecks in the execution pipeline can strongly compromise performance, multiple methods are used to identify sources of inefficiency. Developers can identify bottlenecks in a specific workload by profiling microarchitectural events. For example, by finding the number of machine clears, developers can better understand the prevalence of branch mispredictions in the workload. Performance monitoring units (PMUs) are used to complete this profiling [13].

Performance counters can be used to diagnose bottlenecks in a “bottom-up” analysis, where only performance events known to impact performance are measured [14]. However, this method does not measure, and thus detect, bottlenecks not anticipated at design time. In contrast, the “top-down” method divides the execution pipeline into mini-processes in a hierarchical, tree structure, as shown in Figure 1. At each level, relevant and problematic processes are flagged. Descending through the hierarchy allows developers to examine each possible source of inefficiency at a greater level of granularity [13].

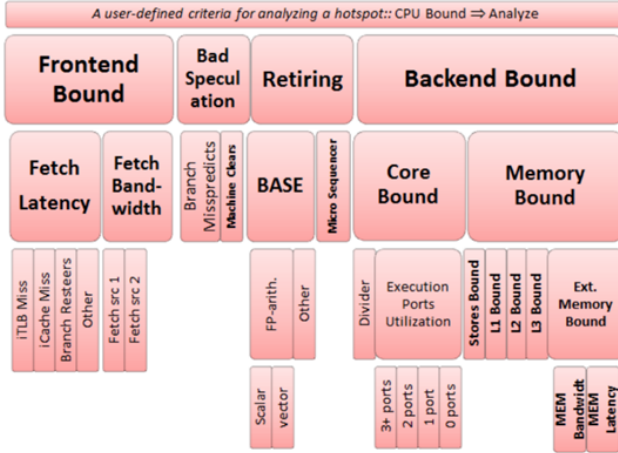


Fig. 1. The Top-Down Performance Analysis Hierarchy [13]

At its highest level, the top-down performance hierarchy divides the pipeline into four domains: frontend bound, bad speculation, retiring, and backend bound. These four domains are used to describe the behaviour of pipeline slots [13]).

### 3.4.3 Pipeline Slot Behaviour

For a processor to achieve maximal throughput,  $\mu$ ops must be issued continuously. Ideally, a  $\mu$ op is issued and retired without stalling in the pipeline, as the higher the proportion of retired slots, the greater the IPC. However, the pipeline slot may need to be cleared due to bad speculation, which occurs when  $\mu$ ops issued after a branch misprediction are flushed.  $\mu$ ops are also stalled by frontend or backend inefficiency, reducing throughput [13].

A  $\mu$ op is frontend bound when it stalls while waiting to be decoded, potentially due to an instruction cache miss. This under supplies the backend [13]. Intel processors use a Complex Instruction Set Computer Architecture [15]. Each cycle, instructions are read into a fetch buffer, where a pre-decoder breaks them down into individual macro-operations. If the decoded instruction length differs from the default length, a stall

penalty may be exacted. Macro-operations are then translated into Reduced Instruction Set Computing  $\mu$ ops [16]. One macro-operation can either be translated into a single  $\mu$ op or multiple  $\mu$ ops. In the decoding pipeline, some decoders can only be used with macro-operations that are translated into a single  $\mu$ op. Bottlenecks can occur if there are many complicated macro-operations, each of which need to be translated into multiple  $\mu$ ops [15]. To address this problem, frequently used  $\mu$ ops can be stored in a  $\mu$ op cache, allowing the process to save time by avoiding the decoding process [16].

When the unavailability of backend resources stalls a  $\mu$ op, the pipeline slot is backend bound. Backend stalls, also known as Execution Stalls, reduce the IPC to below its maximal value. Backend stalls are attributed to either memory or core bound stalls. Memory bound stalls are due to memory subsystem-related lags, such as data cache misses. Core bound stalls are due to execution lags unrelated to memory, such as ALU stalls [13].

Memory bound stalls are particularly important as datasets, and thus the inverted index, grow larger. The resulting memory cost is one of the primary challenges facing search engines, as memory may not be able to scale with dataset size. While DRAM capacity scales by 10% per year, datasets can double in size, leading to large memory delays and backend stalls as information is retrieved from storage [1].

## 4 METHODOLOGY

### 4.1 Hardware

All benchmarking experiments are run on the Ubuntu 20.04 Linux operating system. We conduct the experiments on an Intel® Xeon® Silver 4215R CPU, which operates at 3.20GHz. The system has one socket, with 8 associated cores and 2 threads per core.

### 4.2 Performance Counters

We use Intel’s (2014) top-down analysis framework to investigate the indexing and query evaluation functionalities of the Lucene search engine library (version 10.0.0) when using DRAM [13]. We use the libpfm4 library to implement performance counters and collect data [17]. While these performance counters are chosen using Intel’s 2014 specifications, minor changes in processors and microarchitecture mean there are some slight differences. We present our top-down metrics in Table 2.

We insert performance counters into the code for each of the programs described below. Each program is run five times. We collect performance counter data and convert it into the metrics shown in Table 2. We average the metrics across the five runs to minimise random error.

We tie the thread running the program to a specific CPU. We arbitrarily choose CPU 2. We conduct our benchmarking experiments on that thread.

#### 4.2.1 Lucene Indexing Analysis

To explore the indexing process holistically, we insert performance counters into the IndexFiles.java program, which is taken from Lucene’s demo module and slightly modified [18]. This program implements Lucene95codec compression encodings. After each run of the IndexFiles.java program, we merge the newly created index into two segments using Lucene’s forceMerge.java program. We use these counters to investigate merging’s efficiency. Additionally, we use performance counters to evaluate Lucene’s Unicode Whitespace, Simple, and Standard Analyzers, allowing us to explore the tokenisation and filtering processes [8].

TABLE 2  
Event counters and metrics used in the top-down analysis. [13]

| Event            | LibPFM Performance Counter                                                                            |
|------------------|-------------------------------------------------------------------------------------------------------|
| Clocks           | <code>cpu_clk_unhalted.thread_p</code>                                                                |
| Total Slots      | <code>4 * cpu_clk_unhalted.thread_p</code>                                                            |
| Slots Issued     | <code>uops_issued.any</code>                                                                          |
| Slots Retired    | <code>uops_retired.retire_slots</code>                                                                |
| Fetch Bubbles    | <code>idq_uops_not_delivered.core</code>                                                              |
| Recovery Bubbles | <code>4 * int_misc.recovery_cycles</code>                                                             |
| Metric           | Formula                                                                                               |
| Frontend Bound   | Fetch Bubbles / Total Slots                                                                           |
| Backend Bound    | $1 - \frac{\text{Slots Issued} - \text{Slots Retired} + \text{Recovery Bubbles}}{\text{Total Slots}}$ |
| Bad Speculation  | $\frac{\text{Slots Issued} - \text{Slots Retired} + \text{Recovery Bubbles}}{\text{Total Slots}}$     |
| Retiring         | Slots Retired / Total Slots                                                                           |

#### 4.2.2 Lucene Evaluate Queries Analysis

We search through the newly created index using the Evaluate-Queries.java program, also slightly modified from its original form in Lucene’s demo module [18]. This program searches for queries containing terms present in the dataset at high, medium, or low frequency. We evaluate single-term and multi-term AND queries, with multi-term queries containing terms present in the dataset at the same frequency level. We search through the database for 100,000 single and multi-term low and medium frequency queries, as well as 10,000 single and multi-term high frequency queries. After finding documents that contain the query terms, the documents are scored, with only the top scoring queries presented to the user. We insert performance counters into this program to explore the efficiency of Lucene’s query evaluation [8].

## 5 LUCENE’S INDEXING PROCESS

### 5.1 Creating the Index

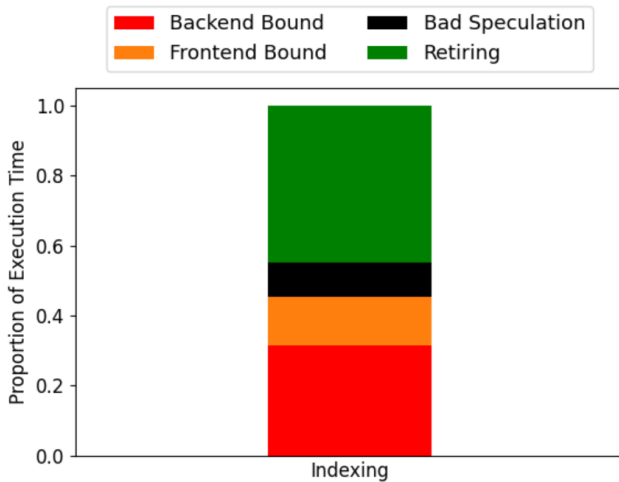


Fig. 2. Top-down analysis of the Index Files program when using DRAM. The metrics are averages from 5 repeats.

During indexing, the highest proportion of execution pipeline slots are retired (45%, see Figure 2). This shows the efficiency of the indexing process. Backend related stalls take up the second largest proportion of execution time (about 31%).

The large proportion of time waiting for backend resources is likely due to the execution pipeline waiting for memory accesses to return data from memory. The program must retrieve relevant information from the dataset to create the inverted index, involving memory accesses beyond the three levels of the data cache. This can force many μops to wait for data in the issue queue, producing backend stalls.

Bad speculation and frontend related stalls are relatively minor, as they affect approximately 10% and 14% of pipeline slots, respectively. The index writer may be the source of this bad speculation. The index writer may predict that a certain document or term is already present in the index. If this prediction is incorrect, the execution pipeline must be flushed and execution restarted from the correct branch, producing bad speculation.

**Indexing Finding.** *The indexing process is relatively efficient, retiring 45% of execution pipeline slots. While bad speculation is the largest source of inefficiency, affecting 31% of pipeline slots, frontend stalls and bad speculation also reduce efficiency. They affect 10% and 14% of pipeline slots respectively.*

The next two sections (‘Analyzers’ and ‘Segment Merging’) break down the indexing process further.

### 5.2 Analyzers

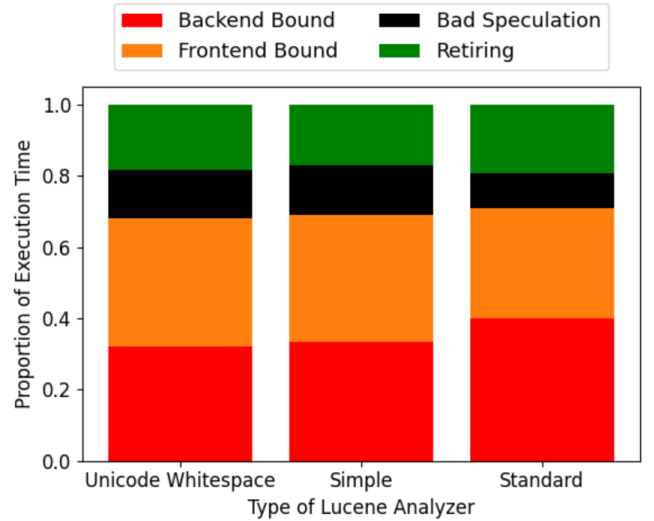


Fig. 3. Top-down analysis of Lucene’s analyzers when using DRAM. The metrics are averages from 5 repeats.

Figure 3 illustrates the top-down analysis of Lucene’s Unicode Whitespace, Simple, and Standard Analyzers.

Stalls due to backend inefficiencies increase with Analyzer complexity. Backend stalls most affect the Standard Analyzer. In contrast, they least affect the Unicode Whitespace Analyzer. While the Standard Analyzer consists of the Lower-Case Filter, Stop Filter, and Standard Tokenizer, the Unicode Whitespace Analyzer only consists of the Whitespace Tokenizer [8]. Potentially, data and tokens passed between all of the Standard Analyzer’s components produces backend stalls, as the analyzer accesses memory multiple times. In contrast, data passes through only one component when using the Unicode Whitespace Analyzer, reducing memory accesses. According to Lucene’s documentation, the analysis process can substantially slow indexing, where more complex Analyzers result in slower indexing [8].

Bad speculation and frontend related stalls affect the three analyzers similarly.

The large proportion of frontend related stalls, between 31% and 36%, may be due to bottlenecks in the decoding process. This type of bottleneck can occur when an instruction's length differs from Intel's default instruction length, or when many of the macro-operations are complex and need to be translated into multiple  $\mu$ ops. The Standard Analyzer appears to have a lower proportion of execution pipeline slots wasted due to frontend stalls than the other two analyzers. This difference may be due to caching of similar instructions used by the Standard Analyzer's multiple filters. However, this difference may not be significant, as the range of values is only about 5%.

Similarly, the range of execution slots wasted due to bad speculation is about 4%, which is likely not significant. The possible reasons for bad speculation are explored more detail in the 'Evaluate Queries' section below.

The proportion of retired slots does not exceed approximately 19% for any of the Analyzers, indicating the process's propensity to stall or suffer from bad speculation. All three Analyzers have relatively similar proportions of retired pipeline slots.

**Analyzers Finding.** *As an Analyzer's complexity increases, the proportion of backend stalls also increases (31% to 36%). In contrast, as the Analyzer's complexity increases, the proportion of frontend stalls decreases (36% to 31%). The high contributions of frontend and backend stalls reduce the Analyzers' efficiency, as seen in the low proportion of retired pipeline slots (less than 20%).*

### 5.3 Segment Merging

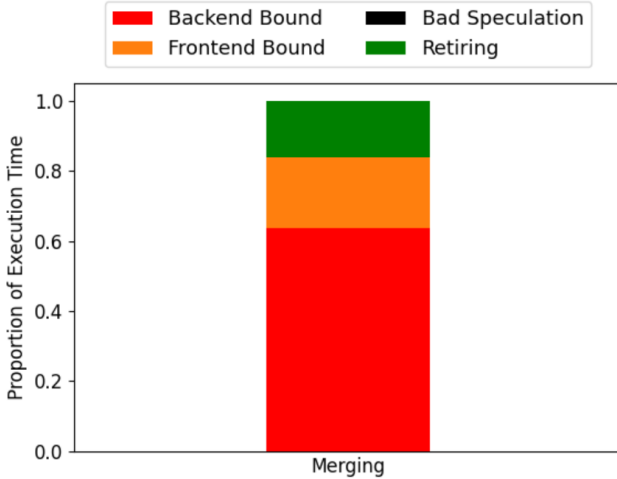


Fig. 4. Top-down analysis of Lucene's Forced Merge operation when using DRAM. The metrics are averages from 5 repeats.

We see that, by far, backend stalls affect the highest proportion of execution pipeline slots (about 64%) during merging. This is expected. When merging, Lucene can require free space in memory up to two times the size of the index, as unmerged segments cannot be deleted until the new, merged segments are complete [8]. According to Lucene's documentation, forced merges are "horribly costly", particularly when the index is merged into a small number of segments, as we do in this report [8]. The backend stalls' negative influence is likely why there is a relatively low proportion of retired pipeline slots.

On Figure 4, we notice that frontend related issues stall about 20% of execution pipeline slots during the forced merge. This could be due to the CPU executing a large number of instructions, such as checking the delete file to determine whether

documents in the index should be allocated into the newly merged segment. Having a large number of instructions can produce the decoder related issues discussed in the previous section.

Between the creation of the index and the forced merge, the index is static, meaning no writes to the index occur. In this static index, program branches related to deleted documents are not used. This may be why bad speculation is negligible (about 1%).

**Forced Merge Finding.** *Merging is a major source of inefficiency in the indexing process, as only 16% of pipeline slots are retired. When merging the static inverted index into two segments, backend delays stall 65% of pipeline slots. Frontend processes also stall 20% of pipeline slots, but bad speculation is negligible (affecting about 1% of pipeline slots).*

## 6 LUCENE QUERY EVALUATION

### 6.1 Searching and Scoring Queries

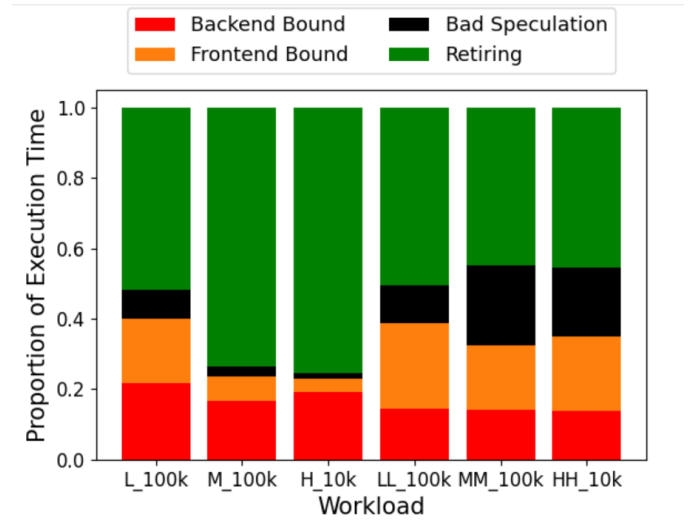


Fig. 5. Top-down metrics for the Evaluate Queries program with DRAM. Metrics are averaged over 5 repeats. Queries contain high (H), medium (M), or low (L) frequency terms. Multiple-term queries are AND queries. A workload evaluated either 100,000 (100k) or 10,000 (10k) queries.

Query evaluation is very efficient for all query types, as a large proportion of execution pipeline slots are retired (45% to 75%). We see can see this efficiency in Figure 5. Single query workloads are particularly efficient, where single-term queries retire approximately 75% of their slots. This may be due to the helpful effects of locality, as high frequency terms are more abundant. Thus, the distance between terms is likely to be smaller than for low frequency terms, increasing the benefits gained from spatial locality.

#### 6.1.1 Single-Term Queries

The effect of bad speculation and frontend stalls on single-term queries decreases as the frequency of the queried term in the database increases. Specifically, between low and high frequency queries, bad speculation decreases from 8% to 2% and frontend related stalls decrease from 18% to 4%.

Despite the high frequency query list containing ten times fewer terms than the low frequency query list, terms in the high frequency list are more likely to be repeated. For example, in the list of 10,000 queries containing high frequency terms, the term

'2009' repeats 97 times. In contrast, in a list of 100,000 queries containing low frequency terms, the term 'asking' repeats only 5 times. Therefore, high frequency query terms are more likely to be in the Query Cache than low frequency terms. If the requested query is in the Query Cache, the CPU can skip the decoding and document ID retrieval processes, thus avoiding expensive decompression processes. These benefits likely reduce the number of frontend, backend, and bad speculation induced stalls, resulting in high frequency single-term queries having the lowest bad speculation and frontend related stalls. This helps explain why the high frequency single-term queries have such a high proportion of retired slots.

Low frequency queries have a higher proportion of backend stalls (22%) than high frequency queries (19%). This may be due to the lower frequency terms being more spread out over the database. Since higher frequency terms are more abundant in the database, it is more likely that spatially or temporally similar locations in memory will be accessed multiple times. This probability decreases for low frequency terms, reducing the helpful effects of locality to speed up memory accesses and potentially resulting in a higher proportion of backend stalls.

### 6.1.2 Multi-Term AND Queries

The proportion of retired execution pipeline slots for multi-term queries (45% to 50%) is less than that of single-term queries (52% to 75%) due to increased bad speculation and frontend related stalls.

Bad speculation affects a higher proportion of pipeline slots for high frequency multi-term queries (19%) than low frequency multi-term queries (11%). Interestingly, this relationship is opposite to that observed for single-term queries. Although Figure 5 appears to show that medium frequency multi-term queries have a much larger proportion of slots wasted due to bad speculation than high frequency queries, the two types of query have very similar proportions of slots wasted (23% and 19% respectively).

There are more possible multi-term queries than single-term. This may reduce the usefulness of the Query Cache and force Lucene to make branch predictions and go to memory. Consequently, multi-term queries have a different relationship to bad speculation than single-term queries.

Search and decompression may cause the high proportion of bad speculation stalls. For example, higher frequency terms are distributed in the index at a greater abundance, increasing the number of skip lists that the searcher must delve into compared to queries with low frequency terms that are present in fewer documents. The decision to delve into versus bypass a skip list may be a source of bad speculation, as high frequency queries have less uniform 'bypass' branches than low frequency queries.

Encoding schemes may also cause bad speculation related stalls. Term frequencies, which can be used to score documents, are compressed by PFOR encoding [1], [19]. However, PFOR encoding is a source of branch misprediction. If the decoder assumes all integers in the compressed list are small, it suffers from branch misprediction when it encounters large integers [19].

Lucene encodes information about term positions using variable length byte (VLB) encoding (Apache, 2023). If the processor incorrectly predicts the value of the continuation flag when decompressing VLB blocks, the processor suffers from branch misprediction [1].

Higher frequency queries need to access more posting lists, term frequencies and term positions than lower frequency queries. Thus, the higher frequency queries must decompress

more data. They consequently suffer from more decompression induced branch mispredictions, potentially explaining why they suffer from a higher proportion of bad speculation than low frequency queries.

Hadjilambrou et al.'s 2019 paper corroborates our hypothesis that decompression is a major source of inefficiency [3]. This increases the validity of our findings.

### 6.1.3 Single-Term versus Multi-Term Queries

A higher proportion of frontend bound stalls occur when evaluating multi-term versus single-term queries (18% to 24% versus 4% to 18%). This may be due to multi-term queries having more complex instructions than single-term queries. Low frequency queries have the highest proportion of execution pipeline slots affected by frontend stalls. This may be because queries are more likely to be repeated in the high frequency query list than the low frequency query list. Thus, the Query Cache is less useful for low frequency queries, creating more work for the decoder and increasing frontend stalls.

Backend processes stall about 14% of pipeline slots during multi-term query evaluation. These stalls are likely due to memory accesses and data cache misses. Backend stalls affect a higher proportion of pipeline slots (up to 22%) during single-term query evaluation. There are likely less search results for multi-term AND queries than single-term queries, suggesting the former require fewer scoring computations. This may reduce core-bound backend stalls, contributing to the lower proportion of backend stalls.

**Evaluate Queries Efficiency Finding.** *Single and multi-term queries are generally efficient, always retiring at least 45% of their pipeline slots. Single versus multi-term queries have different associated inefficiencies. Overall, single-term queries are more efficient than multi-term queries. The former retire 52% to 75% of their pipeline slots while the latter retire 45% to 50% of their pipeline slots.*

**Evaluate Queries Frontend Finding.** *In both single and multi-term queries, frontend related stalls always have the largest effect on low frequency queries (18% and 24%). Stalls in frontend processes have a larger effect on multi-term queries than single-term queries (18% to 24% versus 4% to 18%).*

**Evaluate Queries Backend Finding.** *Backend related stalls affect a higher proportion of pipeline slots during single-term query evaluation than multi-term query evaluation (up to 14% versus up to 22%). For both query types, backend stalls are higher for low frequency queries than high frequency queries (22% versus 19% for single-term and 14% versus 13% for multi-term).*

**Evaluate Queries Bad Speculation Finding.** *Bad speculation is a source of more inefficiency for multi-term queries than single-term queries (11% to 23% versus 2% to 8%). It affects a higher proportion of slots during high frequency multi-term query evaluation (19%) than during low frequency multi-term query evaluation (11%). In contrast, it affects a higher proportion of slots during low frequency single-term query evaluation (8%) than during high frequency single-term query evaluation (2%).*

## 7 THREATS TO VALIDITY

Different search engines have different implementations. For example, they may have different memory management schemes and query scoring algorithms. Consequently, the results of this report may not be applicable to all search engine libraries.

Similarly, we conduct all benchmarking experiments on an Intel machine. These results may not be generalisable to other machines, such as AMD machines.

Unfortunately, our query lists contain repeats of the same query. This may reduce the accuracy of our findings, as the repeated queries may benefit from caching and locality, obscuring potential bottlenecks.

Additionally, in our analysis we only benchmark the behaviour of a limited set of queries. It would be interesting to analyse more complex queries, such as range queries, multi-term OR queries, and multi-term queries that contain terms present at different frequencies in the database.

## 8 RECOMMENDATIONS

In this report, we hypothesise that backend stalls are mainly due to memory accesses. We may be able to reduce this inefficiency by increasing the size of data caches. Limiting expensive memory operations would also minimise backend stalls. For example, using the simplest possible Analyzer may increase performance by decreasing the number of components that need to access the tokenised text from memory. Similarly, we recommend that merging is only conducted when absolutely necessary due to its large memory cost [8]. If the method must be called, it should not be used to merge the index into a small number of segments and sufficient free memory must be made available.

Decompression may cause bad speculation, resulting in stalls and inefficiency. Given that compression is essential to the viability of the index's storage, the ensuing branch mispredictions and pipelines clears may be inevitable. Therefore, it may be useful to have a smaller issue queue to limit waste during inevitable decompression-induced pipeline flushes.

To reduce stalls due to frontend, decoder-related processes, a larger  $\mu\text{op}$  cache could be implemented. Having a greater number of decoders that can translate a complex macro-operation into multiple  $\mu\text{ops}$  may also reduce frontend related stalls.

Different types of query (single versus multi-term, and containing high, medium, versus low frequency terms) suffer from different types of inefficiency. If query management processes are tailored to query type, the efficiency of query search and evaluation may improve.

## 9 FUTURE WORK

This report uses Intel's (2014) top-down analysis framework to present a microarchitectural characterisation of the search engine library Lucene. However, it only uses the first level of the top-down analysis hierarchy. In the future, the second level of the hierarchy could be used to refine our understanding of inefficiencies in Lucene's indexing and query evaluation functionalities. For example, it would be useful to know whether frontend related inefficiency was due to decoder unavailability. This further analysis could generate more specific optimisation strategies.

In this report, we merge a static index. Future work explore how backend bottlenecks are affected if the index being merged contains a large number of deleted documents. These deleted documents would not be carried over to the merged index, reducing the size of the index. It would be interesting to determine whether the increased space in memory available after the merge offsets the memory cost of the merging process.

Additionally, further work could delve into Lucene's open-source code to explore its indexing and query evaluation functionalities at a finer granularity. For instance, analysing decompression operations for different types of queries would shed light on potential inefficiencies, generating more tailored optimisation strategies.

In this analysis, we find that different types of query (single versus multi-term and containing terms found at different frequencies in the database) have different associated bottlenecks. Future work could explore this discrepancy to produce more effective query management strategies. For example, future work could investigate how scoring differences between single and multi-term queries affect core-bound backend stalls.

## 10 CONCLUSIONS

In conclusion, Lucene's indexing and query evaluation functionalities are efficient. About 45% of pipeline slots are retired during the creation of the inverted index. Depending on the query workload, between 45% and 75% of pipeline slots are retired during query evaluation.

Backend related stalls affect the highest proportion of execution pipeline slots during indexing (31%). Frontend processes also reduce indexing efficiency. Frontend and backend processes both significantly reduce the efficiency of Lucene Analyzers, stalling between 30% and 40% of pipeline slots. Their exact effect depends on the Analyzer's complexity. Merging is an inefficient process, stalling 65% of pipeline slots. Multi-term query evaluation is less efficient than single-term query evaluation (45% to 50% of slots retired versus 52% to 75% of slots retired). Frontend, backend, and bad speculation related stalls affect query evaluation to varying degrees, depending on the number of terms in the query as well as the frequency with which the query term/s are present in the database. For at least one query workload, all types of inefficiency stall at least 22% of slots.

Our identification of microarchitectural bottlenecks in Lucene's indexing and query evaluation processes allows us to suggest optimisations that target Lucene's greatest inefficiencies. For example, we identify instruction decoding, memory management, merging, and data decompression as optimisation targets. We also suggest choosing the simplest possible Analyzer and having query specific management strategies.

## REFERENCES

- [1] A. Chilukuri and S. Akram, "Analyzing and improving the scalability of in-memory indices for managed search engines," in *Proceedings of the 2023 ACM SIGPLAN International Symposium on Memory Management*, ser. ISMM 2023. New York, NY, USA: Association for Computing Machinery, 2023, p. 15–29. [Online]. Available: <https://doi.org/10.1145/3591195.3595272>
- [2] M. McCandless, E. Hatcher, and O. Gospodnetić, *Lucene in Action, Second Edition*. Manning Publications, 2010. [Online]. Available: <https://www.manning.com/books/lucene-in-action-second-edition>
- [3] Z. Hadjilambrou, M. Kleanthous, G. Antoniou, A. Portero, and Y. Sazeides, "Comprehensive characterization of an open source document search engine," *ACM Trans. Archit. Code Optim.*, vol. 16, no. 2, may 2019. [Online]. Available: <https://doi.org/10.1145/3320346>
- [4] Y. Su, "Optimization and improvement of lucene document sorting algorithm," in *Frontier Computing*, J. C. Hung, N. Y. Yen, and J.-W. Chang, Eds. Singapore: Springer Singapore, 2020, pp. 1227–1232.
- [5] R. Karam, R. Puri, and S. Bhunia, "Energy-efficient adaptive hardware accelerator for text mining application kernels," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 24, no. 12, pp. 3526–3537, 2016.
- [6] J. Heo, J. Won, Y. Lee, S. Bharuka, J. Jang, T. J. Ham, and J. W. Lee, "Iiu: Specialized architecture for inverted index search," in *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS '20. New York, NY, USA: Association for Computing Machinery, 2020, p. 1233–1245. [Online]. Available: <https://doi.org/10.1145/3373376.3378521>

- [7] N. Lester, A. Moffat, and J. Zobel, "Efficient online index construction for text databases," *ACM Trans. Database Syst.*, vol. 33, no. 3, sep 2008. [Online]. Available: <https://doi.org/10.1145/1386118.1386125>
- [8] T. A. S. Foundation, "Welcome to apache lucene," 2023. [Online]. Available: <https://lucene.apache.org/>
- [9] J. Zobel and A. Moffat, "Inverted files for text search engines," *ACM Comput. Surv.*, vol. 38, no. 2, p. 6-es, jul 2006. [Online]. Available: <https://doi.org/10.1145/1132956.1132959>
- [10] A. Brasetvik, "Elasticsearch from the bottom up, part 1," 2013. [Online]. Available: <https://www.elastic.co/blog/found-elasticsearch-from-the-bottom-up>
- [11] M. McCandless, "Visualising lucene's segment merges," 2011. [Online]. Available: <https://blog.mikemccandless.com/2011/02/visualizing-lucenes-segment-merges.html>
- [12] P. Chang, "Lucene inside out — dealing with integer encoding and compression," 2023. [Online]. Available: <https://towardsdatascience.com/lucene-inside-out-dealing-with-integer-encoding-and-compression-fe28f9dd265d>
- [13] A. Yasin, "A Top-Down method for performance analysis and counters architecture," 03 2014, pp. 35–44.
- [14] S. Eyerhan, L. Eeckhout, T. Karkhanis, and J. E. Smith, "A performance counter architecture for computing accurate cpi components," *SIGOPS Oper. Syst. Rev.*, vol. 40, no. 5, p. 175–184, oct 2006. [Online]. Available: <https://doi.org/10.1145/1168917.1168880>
- [15] M. Hirki, Z. Ou, K. Khan, J. Nurminen, and T. Niemi, "Empirical study of the power consumption of the x86-64 instruction decoder," in *Usenix Workshop on Cool Topics in Sustainable Data Centers (CoolDC'16)*. United States: USENIX Association, 2016, usenix Workshop on Cool Topics in Sustainable Data Center ; Conference date: 19-03-2016 Through 19-03-2016.
- [16] X. Ren, L. Moody, M. Taram, M. Jordan, D. M. Tullsen, and A. Venkat, "I see dead ops: Leaking secrets via intel/amd micro-op caches," in *Proceedings of the 48th Annual International Symposium on Computer Architecture*, ser. ISCA '21. IEEE Press, 2021, p. 361–374. [Online]. Available: <https://doi.org/10.1109/ISCA52012.2021.00036>
- [17] "Libpfm4: a helper library for performance tools using hardware counters." <https://perfmon2.sourceforge.net/>, 2023.
- [18] A. Hasnat, "lucene-benchmark," 2023. [Online]. Available: <https://gitlab.cecs.anu.edu.au/u7300623/lucene-benchmark>
- [19] M. Zukowski, S. Heman, N. Nes, and P. Boncz, "Super-scalar ram-cpu cache compression," in *22nd International Conference on Data Engineering (ICDE'06)*, 2006, pp. 59–59.