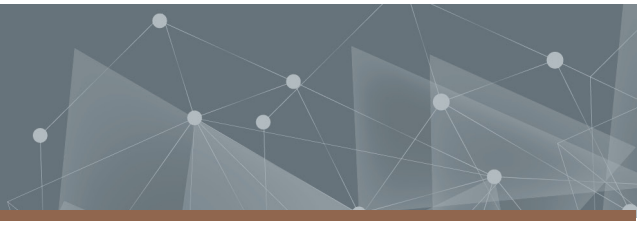
CHALMERS
UNIVERSITY OF TECHNOLOGY



Evaluating the Impact of Compression on Inverted Index Search Engine Performance

Master's Thesis in High-Performance Computer Systems

Hannes Kaulio

Martin Blom

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

CHALMERS UNIVERSITY OF TECHNOLOGY
Gothenburg, Sweden 2025
www.chalmers.se

Master's thesis 2025

Evaluating the Impact of Compression on Inverted Index Search Engine Performance

Hannes Kaulio

Martin Blom

CHALMERS
UNIVERSITY OF TECHNOLOGY

Department of Computer Science and Engineering
Chalmers University of Technology
Gothenburg, Sweden 2025

Evaluating the Impact of Compression on Inverted Index Search Engine Performance
Hannes Kaulio
Martin Blom

Supervisor: Pedro Petersen Moura Trancoso, Computer Science and Engineering
Examiner: Pedro Petersen Moura Trancoso, Computer Science and Engineering
Company Involved: Vesiro
Company Contact: Oskar Hagman

Master's Thesis 2025
Department of Computer Science and Engineering
Chalmers University of Technology
SE-412 96 Gothenburg
Telephone +46 31 772 1000

Typeset in LaTeX
Printed by Chalmers Reproservice
Gothenburg, Sweden 2025

Evaluating the Impact of Compression on Inverted Index Search Engine Performance
HANNES KAULIO
MARTIN BLOM
Department of Computer Science and Engineering
Chalmers University of Technology

Abstract

Efficient compression of inverted indexes is vital for scalable search engines, yet the literature lacks a comprehensive comparison of both modern integer-specific and dictionary-based codecs. This thesis bridges that gap by integrating VarByte, Simple8b, FOR, PFOR/NewPFOR/FastPFOR alongside LZ4, Snappy, and Zstandard into Apache Lucene Core and rigorously benchmarking their impact on compression ratio, indexing throughput, and query latency. Our systematic evaluation uncovers the distinct trade-offs — integer codecs tend to enable faster indexing at the expense of larger footprints, while dictionary schemes offer stronger space savings with moderate latency overhead. Finally, we distill these insights into a lightweight decision-support selection tree that guides practitioners to the optimal codec choice based on their specific application priorities.

Keywords: Inverted Index, Compression Ratio, Search Latency, Query Latency, Index Compression, Information Retrieval

Acknowledgments

We would like to thank everyone at Vesiro for the opportunity to carry out this thesis project and for providing access to the server infrastructure used during testing. A special thanks to Oskar Hagman, our project supervisor at Vesiro, for his consistent support and feedback, and to Felix Naredi for his valuable technical advice and assistance throughout the work. Lastly, we would like to express gratitude to our academic supervisor, Pedro Petersen Moura Trancoso, for his guidance and support throughout the project.

Hannes Kaulio & Martin Blom, Gothenburg, June 2025

List of Acronyms

Below is the list of acronyms that have been used throughout this thesis listed in alphabetical order:

FastPFOR	Fast Patched Frame of Reference
FOR	Frame of Reference
IR Systems	Information Retrieval Systems
JVM	Java Virtual Machine
NewPFOR	New Patched Frame of Reference
PFOR	Patched Frame of Reference
SIMD	Single Instruction Multiple Data
SotA	State of the Art

Contents

List of Figures

List of Tables

1

Introduction

The rapid growth of digital information has led to an increasing reliance on servers to store and manage vast amounts of data. The global data storage market is projected to more than triple in size over the next eight years [16], reflecting the accelerating demand for scalable and efficient data retrieval systems. As data volumes continue to expand, the ability to search through this information efficiently becomes increasingly critical. Search engine efficiency not only determines how quickly relevant information can be retrieved but also impacts resource utilization, including memory, computational power, and energy consumption. These factors are key to ensuring the scalability and sustainability of modern data systems.

A fundamental component of search engines is the index, a data structure that enables fast and efficient document retrieval. Among various indexing methods, the inverted index is widely used due to its effectiveness in handling large-scale text searches by mapping terms to the documents in which they appear. However, as datasets grow exponentially [32], the size of their corresponding indices increases as well. This leads to higher storage demands, increased memory usage, and potentially slower query times, all of which pose challenges to maintaining search performance at scale.

One potential solution to these challenges is compression, which reduces the storage footprint of indices while potentially improving search performance. By compressing indices, more data can be stored in memory, minimizing the need for frequent disk access and improving query response times. Additionally, compression techniques can help reduce operational costs by optimizing storage infrastructure and energy usage, making search engines more efficient and scalable in the face of ever-growing data demands.

Existing research [2, 5, 6, 15, 17, 18, 23, 24, 33, 35, 38] has explored various methods of index compression, including integer compression techniques (e.g. variable-byte encoding and frame-of-reference based techniques) and dictionary-based methods (e.g., Huffman coding and incremental encoding). Understanding how different compression techniques affect search performance and storage efficiency remains a critical area of study, especially as modern applications demand faster and more scalable data retrieval solutions. Having a clear grasp on this is key to further analyzing what techniques hold the most potential for performance gains, as well as knowing which scenarios they might be useful in.

1.1 Aim

The aim of this thesis is to evaluate the impact of different index compression techniques on the performance of inverted index search engines. Specifically, this study seeks to analyze how these compression techniques affect four key performance metrics:

1. **Index Size:** The amount of memory or storage occupied by the index.
2. **Indexing Time:** The time required to build or update the index.
3. **Query Latency:** The time taken to execute a query and return results.
4. **Energy Consumption:** The total energy consumed by the program.

The relative importance of these metrics depends on the use case. For largely static datasets, indexing time has minimal significance since indexing is infrequent. In contrast, for dynamic datasets with frequent updates, indexing speed becomes a critical factor.

In addition to benchmarking the core performance related metrics—compression ratio, search performance and indexing time—we also evaluate the impact on energy consumption from each algorithm. Energy use serves as a fourth performance dimension, offering another layer of insight into the overall efficiency and sustainability of each technique. This is particularly relevant for large-scale deployments where minimizing resource usage and operational cost is a key concern and might be prioritized over pure performance.

1.1.1 Research Questions

This thesis aims to evaluate not only raw performance data but also offer insights into the optimal techniques for different scenarios by addressing the following research questions:

RQ 1: How do different inverted index compression algorithms affect the performance of inverted index search engines in terms of index size, indexing time, query latency and energy consumption?

RQ 2: Can a selection model be designed to select the optimal index compression algorithms based on constraints for index size, indexing time, query latency and energy consumption in inverted index search engines?

1.1.2 Problems

There are several challenges to consider when conducting this research and analyzing its results. One of the most prominent issues is balancing the trade-offs between different compression techniques and their impact on performance metrics. These metrics are often interdependent, meaning improvements in one area may come at

the cost of another. For example, reducing the index size can improve search performance by allowing more data to fit into memory, thereby reducing disk access times. However, the need for decompression introduces computational overhead, which may offset these gains. As a result, optimizing these trade-offs requires careful evaluation, as improvements in one metric might not necessarily translate to an overall performance gain.

Another prominent problem is compatibility, both in terms of the system integration and dataset variability. The research will be conducted using a specific version of Apache Lucene (all details covered in Section 3.2.3), and while the findings may be applicable to similar systems, direct implementation in other versions or distributed environments may require modifications. Additionally, the effectiveness of compression techniques is highly dependent on the characteristics of the dataset. A technique that performs well on one dataset may yield different results on another, raising questions about which technique to use in different circumstances as well as which technique gives the most consistency across diverse data sources and situations.

1.1.3 Limitations

The algorithms considered in this thesis are limited to lossless compression methods that do not alter the core behavior of the search engine or database. This deliberate methodological choice ensures that the search engine continues to produce identical query results as the original implementation. While other approaches involving lossy compression, re-ordering, or static pruning may offer performance gains, they fall outside the scope of this work.

1.2 Ethical Considerations

This thesis project contains a few ethical considerations worth mentioning. Firstly if the emerging results of this research were to be used commercially or in large scale, there is the risk of an algorithm being incorrectly implemented, leading to either faulty behavior and longer latency - or even worse, loss of data. These are all critical malfunctions that could have terrible consequences for any party affected by the system. To mitigate such risks, implementations should include rigorous validation mechanisms, such as unit tests, assertions, and checksum-based integrity checks, to ensure correctness and detect anomalies early in both compression and decompression phases.

Furthermore, index compression and decompression are integral to the core functionality of an IR system, influencing nearly all operations. As a result, changes to the compression system directly impact the overall performance and efficiency of the system. Since system efficiency also affects energy consumption, the choice of compression algorithm can indirectly influence environmental factors such as carbon emissions. From an ethical standpoint, developers and researchers have a responsibility to consider the environmental implications of their design decisions, especially

when deploying systems at scale. Therefore, energy usage becomes a relevant ethical factor, and benchmarking it may provide an additional parameter to consider when selecting or designing compression algorithms.

2
Theory

In this section we first introduce the inverted index and its central role in this research. We then move on to explaining the core idea of index compression and its potential of improving both query latency and efficiency. Thereafter we will cover the current state of the research within this field. Lastly, in Sections 2.3 and 2.4, we cover the theoretical basics of the algorithms used within the two different compression categories.

2.1 Inverted Index

An inverted index is a fundamental data structure used in search engines and information retrieval systems to efficiently locate documents containing specific words or phrases. Unlike traditional document storage, where documents are represented as lists of words, an inverted index reverses this relationship. Instead of listing words per document, it maps each word to a list of documents where that word appears [27]. At its core, an inverted index consists of two main components:

- **Dictionary:** A collection of unique words found in the documents and a postings list corresponding to each word.
- **Postings List:** For every word in the dictionary, the postings list contains:
  - **Document IDs:** The identifiers of all documents where the word appears.
  - **Term Frequency:** The number of times the word appears in each document (optional, used for ranking document when searching).
  - **Positions:** The specific locations of the word within each document (optional, used for advanced search features such as phrase or proximity queries).
  - **Other Metadata:** Additional information, such as document scores, offsets, or payloads, which can enhance search capabilities.

Figure 2.1 shows a visual representation of the structure of an inverted index using 4 made up documents. The example shows how these documents are parsed and indexed into an inverted index. The inverted index in the figure is comprised of a dictionary and postings lists which save the document ID's and positions in this case.

Documents

Document	Text
1	It is said that pigs
2	Pigs like fruit and are
3	Some like it hot, some like it cold
4	It is fruit flies and pigs that

Dictionary

Words	Postings Lists
it	(1;1) (3;3) (3;7) (4;1)
is	(1;2) (4;2)
that	(1;4) (4;7)
pigs	(1;5) (2;1) (4;6)
like	(2;2) (3;2) (3;6)
fruit	(4;3) (2;3)
and	(2;4) (4;5)
are	(2;5)
some	(1;1) (1;5)
like	(2;2) (3;2) (3;6)
hot	(3;4)
cold	(3;8)
flies	(4;4)

Figure 2.1: Visualization of Inverted Index Structure after indexing documents, where the Postings Lists only save Document ID's and Positions

2.2 Inverted Index Compression

Inverted index compression reduces the storage of search engine indexes while potentially improving query performance. The compression targets both major components of the inverted index: the dictionary and the postings lists. Compression provides several key benefits beyond simply saving disk space. Smaller indexes improve cache efficiency by allowing more data to fit in CPU caches, reduce I/O operations by transferring less data between storage and memory.

The selection of compression techniques involves balancing trade-offs between compression ratio, decompression speed, and random access capabilities. Higher compression ratios generally come at the cost of increased decompression time, while some techniques optimize for sequential access at the expense of random access performance.

2.2.1 State of the Art

This section presents the current state of the art in inverted index compression. Broadly, compression methods are divided into two categories: integer-specific coders, which leverage the numeric structure of postings, and dictionary-based (general-purpose) compressors. We review the most widely adopted techniques in both categories, with a focus on their impact on index size, indexing throughput, and query latency.

Integer-Based Compression Techniques Most postings list codecs encode document IDs as differences (deltas) from the previous ID and then apply an integer encoder. Key classes include:

- **Frame-of-Reference (FOR) and block-based codes:** These group sequences of integers into blocks and store each value as an offset from the min-

imum in that block. A fixed number of bits is then used to store the values, based on the largest offset. Word-aligned implementations, such as BP32 and SIMD-BP128, enable high-speed decoding using vector instructions. Studies have shown that FOR-based techniques offer very low query latency due to their fast decoding properties [29].

- **Patched Frame-of-Reference (PFOR):** PFOR extends FOR by handling exception values that do not fit within the fixed bit width [38]. These exceptions are stored separately and later patched in. This approach avoids the compression inefficiency caused by outliers and provides a good balance between compression ratio and decoding speed.
- **SIMD-optimized codecs:** Recent developments include codecs designed for Single Instruction Multiple Data (SIMD) architectures. Examples such as SIMD-BP128 and StreamVByte optimize both throughput and latency by processing multiple integers in parallel [29].

Dictionary-Based Compression Techniques Dictionary compression is often applied to components such as stored fields, term dictionaries, and document metadata. These methods rely on matching repeated byte sequences and replacing them with shorter references. Notable codecs include:

- **LZ4:** A lightweight, high-speed codec derived from the LZ77 algorithm. It is widely used in Lucene and Elasticsearch due to its fast decompression and good balance of speed and compression ratio. It is particularly effective when throughput is more critical than maximum compression.
- **Snappy:** Google's compressor (also LZ77-like). Its speed is similar to LZ4 on some data but typically slower, and its compression ratio is slightly worse. StarRocks documentation ranks Snappy lowest of the common options (Snappy $<$ LZ4 $<$ Zstd $<$ zlib in ratio) [31]. As one study notes, general-purpose codecs like Snappy or LZ4 on inverted lists were found "much slower than PFOR" and often omitted from evaluation [34]. In practice, Snappy is mostly seen in database contexts, not core search engine postings.
- **Zstandard (Zstd):** A modern compression algorithm combining LZ77-style dictionary encoding with fast entropy coding stages like Huffman and FSE. Zstd offers superior compression ratios compared to LZ4 and Snappy, while maintaining acceptable performance. It supports tunable compression levels, allowing users to trade off between speed and compression based on application needs.
- **Sequence-pattern dictionaries (Re-Pair, DINT):** These techniques are designed specifically for compressing postings lists by exploiting repeated patterns of gaps. DINT, for instance, builds a dictionary of common gap sequences and encodes postings as fixed-length references. Such methods have demonstrated competitive compression ratios and fast decoding, though adoption in mainstream systems remains limited.
- **Learned (machine-learned) coders:** A very new direction is to use machine learning to discover compressive transforms. For example, LeCo [25] is a framework that applies learned models to remove serial correlations in a sequence, generalizing classic methods like FOR or delta encoding [29]. In

principle, a learned compressor could automatically adapt to posting patterns, potentially improving on hand-tuned codes. However, such methods are still nascent and have not yet been widely adopted in search engines.

In practice, modern IR systems combine these approaches. For example, Lucene uses FOR or PFOR for postings lists, and LZ4 or Zstd for stored fields [21]. This hybrid strategy allows systems to optimize for different workloads — favoring decoding speed for frequently accessed postings and space efficiency for infrequently accessed text fields.

2.3 Integer Compression

As described in Section 2.1, the postings lists primarily contain integer values such as document IDs, term frequencies, and positions. In this paper, integer compression will refer to the act of representing or storing these lists using less physical memory than the original representation. This is usually done by encoding or modifying the values using techniques based on their expected characteristics, to decrease the number of bits needed to interpret the data. By achieving this we can save physical memory and store a larger portion of the index in caches. The following subsections will present the theoretical basics of the specific integer compression techniques used for integer compression in this paper.

2.3.1 Bit Packing

Bit packing is a technique that optimizes memory usage by storing integers using only the number of bits necessary to represent their values, rather than the standard 32 or 64 bits. This approach is especially beneficial when working with large datasets, as it significantly reduces memory consumption.

However, bit packing introduces computational overhead. Since most processors operate on word-aligned data, accessing packed values requires additional bit-level operations—such as shifting and masking—to pack and unpack individual integers. While this trade-off reduces memory usage, it may slightly impact performance due to the extra processing steps involved.

To illustrate, consider the integers 10, 55, 1005, and 20429. The largest value, 20429, requires 15 bits in binary. Using bit packing, all four integers can be stored using just 15 bits each, instead of the standard 32 or 64 bits. This compact representation leads to considerable memory savings.

Although bit packing adds some computational cost, this can be substantially mitigated using SIMD (Single Instruction, Multiple Data) instructions. Modern CPUs support SIMD operations that enable parallel processing of multiple packed integers in a single cycle. By optimizing packing and unpacking routines with SIMD, the performance penalty from bit manipulation can be significantly reduced [24].

2.3.2 Variable Byte Code

Variable Byte (VB) coding is a simple and widely used technique for compressing sequences of integers, especially in search engines and information retrieval systems. Unlike fixed-width encoding methods, VB coding uses a variable number of bytes to represent each integer, depending on its size.
In VB coding, each byte consists of:

- 1 control bit (the most significant bit).
- 7 payload bits (the remaining bits).

The control bit signals whether the current byte is the last byte for the current integer:

- A control bit of `1` indicates that this is the final byte of the integer.
- A control bit of `0` means that more bytes follow to complete the integer.

The data bits from all bytes are assembled together to recover the original integer.

Example:

Suppose we want to encode the integer 130. Its binary form is:

$$130 = 10000010_2$$

Since 130 requires more than 7 bits, it is split into two bytes:

- *First 7 bits:* `0000010`
- *Remaining bits:* `0000001`

These are then packed into two bytes:

- *First byte:* `00000010` *(control bit 0, more bytes follow)*
- *Second byte:* `10000001` *(control bit 1, last byte)*

Thus, 130 is encoded as two bytes.

2.3.3 Delta Encoding

One way to reduce the memory required to store integers is to use delta encoding. Delta encoding takes a sorted list of integers and stores the differences between each number. The delta is then compressed using some other integer compression technique. Since the deltas will be smaller than the initial values, it will result in a better compression rate. While delta encoding often improves the compression rate, its performance is not optimal when decoding. This is due to the inherent data dependency that exists when adding together the deltas. Since the value of the previous number needs to be known when computing the current one, thus the use of SIMD instructions is not possible to achieve a faster decompression.

2.3.4 Simple Family

The Simple family refers to a group of compression algorithms that encode multiple integers into fixed-size blocks of bits. In these blocks, a variable number of integers

can be packed depending on the maximum size of the integers. The most notable variant is Simple-8b [1], which compresses a sequence of integers into 64-bit words. Each 64-bit word starts with a 4-bit selector field that indicates which compression scheme is used, and the remaining 60 bits are used for actual data storage.

The selector specifies:
- How many integers are packed into the 60 data bits.
- How many bits are allocated per integer.

This design allows the algorithm to adapt flexibly to the distribution of values. If the integers are small (for example, all zeros or ones), many values can fit into a single word. As integers grow larger, fewer can be packed into each word, but compression remains efficient compared to naive storage.

Simple8b is particularly effective when the data contains small deltas or many zeros, which is common after delta encoding. This results in very high compression ratios with minimal computational overhead.

2.3.5 Frame of Reference Encoding (FOR)

FOR encodes blocks of integers of a fixed size. It works by calculating the minimum value (Min_V) and the maximum value (Max_V) in a block of integers. The bit width b required to store the difference $\text{Max}_V - \text{Min}_V$ is then calculated using the formula:

$$b = \lceil \log_2(\text{Max}_V - \text{Min}_V + 1) \rceil$$

Each integer in the block is stored as its difference to Min_V and bit-packed into b bits. The metadata of Min_V and the bit width b is also stored to allow decoding. While FOR encoding is very good at compressing integer blocks where the values are similar to each other, it becomes inefficient if outliers exist. If one value requires more bits than the rest of the values, then all values would need to be encoded using this higher amount of bits.

2.3.6 Patched Frame of Reference Encoding (PFOR)

While Frame of Reference (FOR) works well for integers with similar values, it performs poorly when outliers are present. For example, compressing the numbers 1, 40, 50, 70, and 120 requires just 7 bits per integer with FOR. However, if one of these numbers is replaced with a much larger value, such as 4,294,967,296, then FOR must use 32 bits for every integer, making compression much less efficient.

To solve this problem, Zukowski et al. [38] introduced Patched Frame of Reference (PFOR). The main idea behind PFOR is to compress the majority of integers efficiently using a small bit width, while separately storing a few exception integers that do not fit within the bit width.

The basic process of PFOR works as follows:

- **Block division:** The integers are grouped into fixed-size blocks (commonly 128 or 256 integers per block).
- **Bit width selection:** A bit width b is chosen so that most values in the block fit within b bits. Only a small percentage (e.g., less than 10%) are allowed to overflow.
- **Normal encoding:** Integers that fit within b bits are encoded normally.
- **Exception handling:** Outlier values (exceptions) are stored separately, along with their positions in the block.
- **Decoding:** During decompression, normal integers are decoded first. Then, the decoder patches in the exception values at their stored positions.

By separating exceptions, PFOR achieves better compression than traditional FOR without sacrificing too much decoding speed.

Several variants of PFOR modify how exceptions are handled or how the bit width is chosen:

NewPFOR

NewPFOR builds upon the basic PFOR framework with optimizations targeted at improving decoding speed and compression efficiency, especially in how exceptions are managed. The key distinction lies in how NewPFOR encodes and stores exceptions, favoring a more SIMD-friendly and cache-efficient layout.

- **Exception encoding:** Unlike the original PFOR, which stores full exception values, NewPFOR only stores the high-order bits that do not fit within the chosen bit width b. The low-order b bits of all integers, including exceptions, are stored in the main bit-packed array.
- **Position storage:** The positions of the exceptions within the block are stored in a compact, separate array. This array is often delta-encoded and compressed using a lightweight method such as variable-byte coding to reduce overhead.
- **Reconstruction:** During decoding, the decoder first retrieves the packed normal values, including the truncated exceptions. It then consults the exception position and value arrays to reconstruct the original integers by shifting the high bits left by b and combining them (via bitwise OR) with the stored lower bits.
- **Performance consideration:** NewPFOR is designed to be SIMD-compatible. By isolating exception handling and minimizing branches during decoding, it achieves faster decompression while maintaining competitive compression ratios. The layout ensures that normal values can be processed in bulk using vectorized instructions, with exception correction applied in a secondary pass.

This improved exception mechanism allows NewPFOR to support high-speed decoding, making it well-suited for real-time retrieval systems where throughput is critical.

FastPFOR

FastPFOR further enhances the PFOR family by optimizing for modern processor architectures through aggressive use of SIMD (Single Instruction, Multiple Data) instructions and more structured exception management. While it retains the core idea of separating normal and exceptional values, its design is tailored for high-throughput decoding.

- **Fixed-width encoding:** Like PFOR and NewPFOR, FastPFOR encodes the majority of values in a block using a fixed bit width b. These values are bit-packed contiguously in memory to facilitate fast vectorized decoding.
- **Exception separation:** Values that exceed the representable range of b bits are treated as exceptions. Each exception is divided into two parts:
  - The lower b bits are stored alongside the regular values in the main compressed array.
  - The remaining high-order bits are extracted and stored separately in an exception values array.
- **Position tracking:** The positions of all exceptions within the block are stored in a separate, dedicated array. This array allows the decoder to efficiently locate and correct the placeholder values in the output stream.
- **Decompression process:** During decompression, the main array is decoded first using SIMD-friendly routines. Then, for each exception, the decoder retrieves its position and high bits, reconstructs the original value by shifting the high bits left by b, and combines them with the already decoded lower bits using a bitwise OR operation.
- **SIMD optimization:** A key feature of FastPFOR is that exception processing is designed to minimize branching and enable bulk processing. All exception metadata is aligned and stored in a way that supports prefetching and parallel application, significantly accelerating decompression on modern CPUs.

By isolating exceptions in well-structured, compact arrays and ensuring compatibility with SIMD processing, FastPFOR achieves a favorable balance between compression ratio and decoding speed, outperforming earlier PFOR variants in both throughput and scalability.

2.4 Dictionary Compression

In this paper, dictionary compression will refer to the act of representing or storing the dictionary part of an inverted index using less physical memory than the original representation. As with integer compression in Section 2.3, this is usually achieved by encoding, restructuring or otherwise modifying the bits representing said dictionary. But on the contrary to postings lists, a dictionary contains more than just numerical values. This means that we will have to use other compression techniques for the dictionary. More specifically, techniques based on a fast but simple dictionary-matching stage (the Lempel-Ziv algorithm [37]), which works by finding repetitive characters and storing them as offsets to their first occurrence or an entry from a token dictionary. By also compressing this part of the inverted index

we can save increasingly more physical memory, which leads to a yet smaller index size. The following subsections will present the underlying theory of the compression techniques used for dictionary compression in this paper.

2.4.1 LZ4

LZ4 is a data compression algorithm that aims to provide a good trade-off between compression ratio and (de)compression speed. Unlike other common compression algorithms like Zstd (explained in Section 2.4.3), LZ4 only uses its dictionary-matching stage and does not combine it with an entropy-coding stage. LZ4 compressed data is made up of one or more frames, which in turn contain one or more blocks of data. LZ4 frames are considered independent and can be decompressed independently of each other and blocks may or may not be independent of each other [3].

LZ4 has three defined frame formats: LZ4 frame, skippable frames and legacy frames. LZ4 frames contain compressed data, skippable frames can contain custom user data and the legacy frame format is an older format that is not normally used anymore. Table 2.1 shows the structure of a single LZ4 frame.

Table 2.1: The Structure of a Single LZ4 Frame

`Magic_Number`	4 bytes
`Frame_Descriptor`	3-15 bytes
`Data_Block`	n bytes
`[More Data_Blocks]`	
`End_Mark`	4 bytes
`(Content_Checksum)`	0-4 bytes

The Magic_Number has a value of 0x184D2204. This specific value is used since there is a very low probability of finding it at the beginning of an arbitrary file. The magic number is used as an identifier to confirm that the compressed data is in the LZ4 format, allowing the decompressor to reject formats that it can not decompress early, avoiding wasted processing.

The Frame_Descriptor has a variable size with a minimum of 3 bytes and a maximum of 15 bytes. It is used to provide the decompressor with information needed to decode the data. First it describes what fields are present and then provides them if they are necessary. One of the flags here tells the decompressor if the data blocks are independent or not [10]. Table 2.2 shows the structure of a Frame_Descriptor.

Table 2.2: The Structure of a LZ4 Frame_Descriptor

`FLG`	1 byte
`BD`	1 byte
`(Content_Size)`	0-8 bytes
`(Dictionary_ID)`	0-4 bytes
`HC`	1 byte

After the Magic_Number and Frame_Header comes a number of Data_Block(s). Each frame can contain any number of blocks but must have at least one. Table 2.3 shows the structure of a Data_Block.

Table 2.3: The Structure of a LZ4 Data_Block

`Block_Size`	4 bytes
`Block_Content`	n bytes
`(Block_Checksum)`	0-4 bytes

The highest bit in the Block_Size is 1 if the block is uncompressed and 0 if it's compressed using the LZ4 block format. All other bits are used to represent the size of the Block_Content. The Block_Content is composed of a series of sequences where each sequence is a suite of literals, followed by a match copy operation. A sequence always starts with a one byte token, split into two 4-bit fields. The first field provides the length of the literals that follow the token and the second field represents the number of bytes to copy in the match copy operation. Both fields can represent numbers larger than 15 by adding a variable number of bytes following the field itself in this case. Following the literals after each token is the match copy operation. It is decoded by using the following first two bytes as the offset to copy from (number of bytes backwards from current position), and the second field of the token as the number of bytes to copy. Each Data_Block can end with a Block_Checksum if the Frame_Descriptor has declared so. For each LZ4 frame, the stop of the flow of blocks is denoted with the 32-bit value: 0x00000000 [8].

Lastly the frame can also end with an optional 4 byte Content_Checksum which is also only present if the Frame_Descriptor has declared so. The checksum is the result of the XXH32() hash function [36] digesting the original input data and a seed of zero. The Content_Checksum can be used to validate the result and ensure that all blocks where fully decompressed without any distortion [10].

2.4.2 Snappy

Snappy is a data compression algorithm that aims for high speeds with a reasonable compression ratio. Like LZ4 (see section on LZ4 2.4.1), Snappy uses no entropy-coding stage and relies solely an efficient dictionary-matching stage. Unlike LZ4 and Zstd (see Section 2.4.3 on Zstd) formats, compressed Snappy data can be made up of either on or more frames, or just one stream [19]. For the purposes of this project the framed format is not needed and hence will not be explained. Table 2.4 shows the structure of a Snappy stream.

Table 2.4: The Structure of a Snappy stream

`Uncompressed_Length`	1-4 bytes
`Element`	1-5 bytes
`[More Elements]`	

Table 2.5: The Structure of a single Element in a Snappy stream

`Tag_Byte`	1 byte
`(Additional bytes)`	0-4 bytes

A Snappy stream always starts with the Uncompressed_Length (a maximum value of 2^{32} - 1) stored in a variable byte format(see Section 2.3.2 on variable byte (VB)). This length can be used at the end of the decompression stage to verify that the whole stream was processed, however, it only confirms that the output is of the same size as the original data was and cannot spot any distortions in the data. After the uncompressed length comes the compressed stream itself. It consists of two types of elements: literals and copies. A steam can contain any number of the two element types in any possible order, except that the first element cannot be a copy. All elements start with a one byte Tag_Byte where the lower two bits signify what type of element will follow. A value of (00) signifies a literal element, (01) a copy with 1-byte offset element, (10) a copy with 2-byte offset element and (11) a copy with 4-byte offset element. The interpretation of the upper six tag bits is element-dependent.

Literal elements are literals stored directly in the stream. The literal length is stored in the remaining 6 bits of the Tag_Byte. If the length is larger than 60 a value between 60-63 is used to signify that an additional number (1-4) of bytes is used to describe the length. The literal itself directly follows the length.

A copy element carries a reference back into the previously decompressed data (stream), telling the decompressor what data to reuse in its place. The element contains two encoded values: the offset (number of bytes backwards from current position) and the number of bytes to copy [19].

- (01) Copy with 1-byte offset elements encode their length in the third, forth and fifth bits of the Tag_Byte, and the offset in the remaining bits plus an additional byte after the tag.
- (10) Copy with 2-byte offset elements encode their length in the remaining 6 bits of the Tag_Byte, and the offset in an additional 2 bytes after the tag.
- (11) Copy with 4-byte offset elements also stores it's length in the remaining 6 bits of the Tag_Byte, but uses an additional 4-bytes for the offset.

Unlike the framed formats of LZ4 (Section 2.4.1) and zstd (Section 2.4.3), a Snappy stream has no identifier fields and no checksums to verify correctness at any stage.

2.4.3 Zstandard

Zstandard or "zstd" as it will be referred to from here on out, is a data compression algorithm similar to gzip [13]. It combines its dictionary-matching stage with a large search window and a fast entropy-coding stage, which helps it achieve a better compression ratio at the cost of some speed. Zstd compressed data is made up of one or more frames, which in turn contain one or more blocks of data. In zstd each frame

is independent and can be decompressed independently from any other frames. The decompressed content of multiple concatenated frames is the concatenation of each frame's decompressed content [11].

Zstd has two defined frame formats: Zstd frames and skippable frames. Zstd frames contain compressed data and skippable frames can contain custom user data. Table 2.6 shows the structure of a single zstd frame.

Table 2.6: The Structure of a Single Zstandard Frame

`Magic_Number`	4 bytes
`Frame_Header`	2-14 bytes
`Data_Block`	n bytes
`[More Data_Blocks]`	
`(Content_Checksum)`	4 bytes

The Magic_Number has a value of: 0xFD2FB528. This specific value is used since there is a very low probability of finding it at the beginning of an arbitrary file. The magic number is used as an identifier to confirm that the compressed data is in the zstd format, allowing the decompressor to reject formats that it can not decompress early, avoiding wasted processing.

The Frame_Header has a variable size with a minimum of 2 bytes and a maximum of 14 bytes. It is used to provide the decompressor with information needed to decode the data. First it describes what fields are present and then provides them if they are necessary. Table 2.7 shows the structure of a Frame_Header.

Table 2.7: The Structure of a zstd Frame_Header

`Frame_Header_Descriptor`	1 byte
`(Window_Descriptor)`	0-1 bytes
`(Dictionary_ID)`	0-4 bytes
`(Frame_Content_Size)`	0-8 bytes

After the Magic_Number and Frame_Header comes a number of Data_Block(s). Each frame can contain any number of blocks but must have at least one. Table 2.8 shows the structure of a Data_Block.

Table 2.8: The Structure of a zstd Data_Block

`Block_Header`	3 bytes
`Block_Content`	n bytes

The Block_Header tells the decompressor what Block_Type it is, the Block_Size and if it is the last block. The Block_Type is one of three, a Raw_Block, a RLE_Block or a Compressed_Block. A Raw_Block is simply a block of data that did not compress into a smaller size, it is therefore written in its original uncompressed state. A RLE_Block is a single byte. To be decompressed it needs to be repeated a number of times. For an RLE_Block this number is represented in the Block_Size, since it is already known to always be one byte. A Compressed_Block consists of two sections: a Literals_Section and a Sequences_Section. The results of the two sections are combined to produce the decompressed data. The literals in the Literals_Section can be of the same types as Block_Type, except that for literals, a Compressed_Block is stored using Huffman Encoding (see Section 2.4.3.1), which in turn also has its header compressed using FSE Encoding (see Section 2.4.3.2).

The Sequences_Section contains a succession of sequences, which are comprised of a literal copy command and a match copy command. The literal copy command specifies the Literals_Lengths (number of bytes to copy from the Literals_Section), and the match copy command specifies an Offset and a Length. If there are any literals left after decoding all the sequences they are added to the end of the block. The Literals_Lengths, the Offset and the Length are all stored using one of four types: Predefined_Mode, RLE_Mode, FSE_Compressed_Mode or Repeat_Mode. Predefined_Mode declares that a predefined FSE distribution table is used (see Section 2.4.3.2 on FSE). The RLE_Mode type contains one byte which contains the symbol's value, it is then used for all sequences. FSE_Compressed_Mode declares that standard FSE compression is used, it will then contain a distribution table used for decompression. Lastly Repeat_Mode says to use the last used table in the previous Compressed_Block where the number of sequences is greater than zero. If this is the first block, the table in the dictionary is used [11].

The frame can end with an optional 32-bit Content_Checksum which is only present if the Frame_Header has declared so. The checksum is the result of the XXH64() hash function [36] digesting the original input data and a seed of zero. The Content_Checksum can be used to validate the result and ensure that all blocks were fully decompressed without any distortion [11].

2.4.3.1 Huffman Encoding

Huffman encoding or coding is an entropy encoding method [14] that attempts to compress data by replacing source symbols with a particular type of optimal prefix code called a Huffman code. Huffman encoding on its own is not used for compression of anything, but is used in parts of the zstd (see Section 2.4.3 on zstd) algorithm to improve it's compression ratio. For text, in essence, Huffman coding can be described as replacing character ASCII codes with shorter variable length codes (Huffman codes) [22].

Huffman encoding and decoding works by using a binary tree called a Huffman tree that describes the mapping of Huffman codes to their respective source symbols. To achieve optimal compression, the shortest Huffman codes are assigned to the source

symbols with the highest frequency. For compression, a new Huffman tree can be built in O(n log n) time (where n is the number of symbols) or a predefined tree can be used. In cases where character frequencies are fairly predictable, the Huffman tree can be preconstructed and adjusted during decompression, otherwise the same predefined tree or a provided tree must be used (the provided tree is usually encoded in the beginning of the compressed stream) [22].

2.4.3.2 FSE Encoding

Finite State Entropy (FSE) encoding, like Huffman encoding, is an entropy based encoding method. Compared to Huffman, FSE has a higher accuracy, which lets it achieve a higher compression ratio, especially on symbol sets where the distribution is squeezed. However, sets with less compressible distributions show little difference in compression ratio, with Huffman largely outperforming FSE on both compression and decompression speed [9].

FSE coding is based on Asymmetric Numeral Systems (ANS) theory from Jarek Duda [14], specifically a table-based ANS method called tANS. FSE involves a state that is carried over between symbols and represents the total state of the compressed data so far. This means that decoding must be done in the reverse order of encoding to correctly reconstruct the source data. Encoding and decoding both work by using a table that maps each state and symbol to a new state and bits to output/read [11].

- Encoding involves indexing the current state and symbol into the table, the table then specifies how many bits (of the state) to emit to the compressed stream and the next state to transition to. This process repeats until all symbols are encoded.
- To decode, a lookup is done on the table with the current state. The table then provides the source symbol, the previous encoded state and a number of bits to read from the compressed stream. The source symbol is added to the output and the bits read from the stream are combined with the previous state to provide the next state (encoders previous state). This continues until the initial state (a specific predetermined integer) is reached.

2.5 Summarizing Table

Table 2.9 presents a qualitative summary of the performance characteristics of selected compression algorithms across four key metrics: Compression Ratio (CR), Indexing Time (IT), Query Latency (QL), and Energy Consumption (EC). The fill level of each circle reflects relative performance — a higher fill indicates better performance on that metric.

These qualitative assessments are based on data collected from benchmark studies, technical documentation, and comparative analyses discussed in Section 2.2.1. When exact quantitative data was unavailable or inconsistent across sources, the evaluations relied on commonly observed trends reported in the literature. Key references informing these assessments include: [5, 11, 19, 23, 24, 37].

Table 2.9: Summary of compression algorithm performance across key metrics. Higher fill level indicates better performance. Metrics: Compression Ratio (CR), Indexing Time (IT), Query Latency (QL), Energy Consumption (EC).

Algorithm	**CR**	**IT**	**QL**	**EC**
Integer Compression				
VarByte	◔	◕	◔	◐
Simple8b	◐	◕	◐	◕
FOR	◐	●	●	●
PFOR	◕	●	◕	◕
NewPFOR	◕	◕	◕	◕
FastPFOR	◕	◐	◕	◐
Dictionary Compression				
LZ4	◕	◕	●	◕
Snappy	◐	●	◕	●
Zstd	●	◐	◐	◐

3
Methods

This chapter outlines the methodology used to evaluate the impact of compression algorithms on inverted index performance in Apache Lucene. It describes the integration of both integer compression algorithms and dictionary compression algorithms into Lucene's indexing framework, followed by the design of benchmarking procedures to measure indexing speed, query latency, compression ratio, and energy usage. The chapter also details the hardware and software configurations, the rationale for dataset and query set selection, and the procedures used to ensure the validity and reliability of the results.

3.1 Compression

Implementation of all algorithms will be conducted using an open-source search software called Apache Lucene (sometime referred to as just "Lucene"). Apache Lucene contains a Java library named Lucene Core which provides all of its database functionality such as indexing, searching, hit highlighting and advanced analysis/tokenization [26]. It is this library that will be modified to use the different integer and dictionary compression algorithms previously mentioned.

All algorithms have been chosen based on existing research with performance relating to our four metrics and SotA solutions targeting query performance. All algorithms have been implemented natively in Java into Lucene using the same techniques and without any optimizations that are not present in others.

3.1.1 Integration of Compression and Decompression in Lucene

To study the effects of compression strategies in an information retrieval system, we extended Apache Lucene by integrating support for additional compression algorithms. This involved modifying the source code to allow custom algorithms to be applied during indexing and querying.

3.1.1.1 Compression During Indexing

When documents are indexed in Lucene, they are first tokenized and transformed into postings lists that store mappings between terms and document identifiers, as well as positional and frequency information. Normally, these postings are compressed using Lucene's built-in methods before being stored on disk.

In this project, we modified the indexing logic to apply alternative integer compression to the numeric components of the postings lists. In addition, dictionary compression methods were used to compress the dictionary part of the inverted index. These changes occur at the point where Lucene serializes its internal data structures to storage during indexing.

3.1.1.2 Decompression During Querying

When a query is executed, Lucene loads the relevant segment data from disk and decompresses it before proceeding with search operations like filtering, scoring, and ranking. We inserted decompression steps into this sequence so that Lucene can correctly interpret postings that were previously compressed using our custom methods. These steps take place immediately after disk read operations, ensuring that decompressed data is available for the search engine logic as expected.

3.1.2 Integer Compression

For integer compression, a number of algorithms were chosen to be implemented and benchmarked. The algorithms that were implemented are: VarByte 2.3.2, Simple8b 2.3.4, Frame of Reference (FOR) 2.3.5, Patched Frame of Reference (PFOR) 2.3.6, NewPFOR 2.3.6, FastPFOR 2.3.6

The integers were being compressed in blocks of 128 integers each. All integers given were already delta encoded before being compressed. Some implementation details worth noting are:

Variable Byte (Var Byte) The Var Byte algorithm was implemented using Lucenes VarInt implementation[1].

FOR & PFOR & NEWPFOR & FASTPFOR FOR based techniques were implemented using SIMD bitpacking using Java's Vector library[2].

3.1.3 Dictionary Compression

For dictionary compression, three algorithms were chosen to be implemented and benchmarked. These algorithms are: LZ4 (Section 2.4.1), Snappy (Section 2.4.2) and Zstandard "zstd" (Section 2.4.3). Although existing research concludes that zstd boasts the best compression ratio, SotA solutions seem to point to LZ4 being the objectively best algorithm for IR Systems due to its focus on speed, specifically within decompression [20, 30]. This claim will be further discussed in the results and evaluation Section 4.

[1] `https://lucene.apache.org/core/9_11_1/core/org/apache/lucene/store/DataOutput.html#writeVInt(int)`

[2] https://docs.oracle.com/en/java/javase/17/docs/api/jdk.incubator.vector/jdk/incubator/vector/Vector.html

LZ4 For the LZ4 algorithm we will be using the existing custom version already present in the Lucene Core library[3]. This version is based on the official open source LZ4 project written in C by Yann Collet [7].

Snappy & Zstd For the Snappy and zstd algorithm, we will be using the Java implementations from the Aircompressor project[4] which are Java ports of the original implementations from Google[5] and Facebooks[6] open source projects respectively.

3.2 Benchmarking

The benchmarking process consisted of two distinct phases: *Index Benchmarking* and *Search Benchmarking*. During both phases, time metrics-for indexing time and query latency-are measured by taking timestamps from the JVM's high-resolution time source using the standard Java System class[7] which provides nanosecond precision.

Index Benchmarking The index phase, also called *Indexing*, covers the procedure of processing documents and building the index with the information [required/stated] by the index configuration (see Section 3.2.2). For this phase, a complete benchmark consists of indexing a whole dataset, file by file, adding each file to the index as a Lucene Document. The performance is measured by recording the time it takes for each file in a dataset to be processed, up until the index writer commits all pending changes and closes (releases the ownership of the index files). After this has been completed, we record the final index size by measuring the combined size of all documents comprising the index. The datasets used in this phase are described in Section 3.2.4.

Search Benchmarking This phase covers the performance measuring of the index by testing the searching capabilities and speeds of different queries. For this phase, a complete benchmark consists of executing a whole predefined set of queries against the index and recording the average query latency. This is done by measuring the latency of each query, accumulating the total response time. When the last query has been processed the total response time is divided by the number of queries ran, achieving the average query latency. The predefined sets of queries used during this phase are described in Section 3.2.5.

The following sections will provide more detailed descriptions of the benchmarking procedures and configurations applied throughout all tests.

[3] `https://lucene.apache.org/core/10_1_0/core/org/apache/lucene/util/compress/LZ4.html`

[4] `https://github.com/airlift/aircompressor`

[5] `https://github.com/google/snappy`

[6] `https://github.com/facebook/zstd`

[7] `https://docs.oracle.com/javase/8/docs/api/java/lang/System.html`

3.2.1 Hardware Setup

The benchmark program was executed on a Linux-based system (Debian GNU/Linux 12) equipped with an Intel Xeon CPU @ 2.80GHz. This processor features:

- L1 Cache: 32 KB (instruction) + 32 KB (data) per core
- L2 Cache: 1024 KB per core
- L3 Cache: 66 MB (shared across 2 sockets)

The system was configured with 128 GB of DDR4 RAM. Storage was provided by a 375 GB SSD, with an approximate throughput of 2,400 MB/s for reads and 1,400 MB/s for writes.

3.2.2 Index Configuration

For consistency and to keep all results relevant to each other, the same index configuration will be used across all tests and benchmarks. The index configuration is set to only store document IDs, term positions and frequencies, which are all impacted by either the dictionary or integer compression. In order to only gather data that is relevant to our results, other fields and formats (such as vectorization) used to enable or enhance more advanced features in Apache Lucene, are omitted or unused. A new index is created for every algorithm.

3.2.3 Software Configuration

The experiments were conducted using Apache Lucene 10.1.0[8] with vectorization turned on, otherwise no optimization flags were given. The JVM used was OpenJDK 21[9].

The tests were conducted using a new Lucene process for each algorithm, or combination of algorithms. This is to ensure a clean working environment with nothing cached from any previous testing. The computer had minimal background processes to ensure consistent benchmarking results.

3.2.4 Datasets

There are several popular datasets that are often used within the field of information retrieval. These datasets usually share a lot of characteristics and try to simulate realistic data by being based on public information retrieved from the web. For our research, we chose to work with a dataset called Common Crawl[10] due to its ease of use and retrieval.

A Common Crawl archive consists of unstructured text data, is usually somewhere between 30-140 TiB, and is based on information "crawled" from billions of pages on

[8] `https://lucene.apache.org/core/corenews.html#apache-lucenetm-1010-available`
[9] `https://openjdk.org/projects/jdk/21`
[10] `https://commoncrawl.org/`

the web [12]. In order to work with a feasible amount of data we took samples from the 2025 January archive[11] and the 2016 October archive[12], these samples are used as two separate datasets to provide some variance in content. The datasets will be referred to as *CommonCrawl-2025* and *CommonCrawl-2016* respectively and will be used in the sizes: 10 GB and 100 GB (uncompressed text data) in order to see how different algorithms scale with increasing dataset sizes.

3.2.5 Queries

The type, *k-value* and specific keyword of a query can have a significant impact on the query latency [27]. Therefore, it is crucial that the same set of queries is used consistently when benchmarking different algorithms; otherwise, the results - and by extension, any comparison between them - would be inaccurate.

The k-value refers to the number of documents requested in a query result. As mentioned, this value can have a substantial effect on query latency. To ensure that our benchmarks reflect a broad range of realistic usage scenarios, we incorporate varying n-values. To achieve this without distorting the results of different benchmarks, a predetermined amount of queries will use the values: 10, 100 and 1000 consistently across all benchmarks.

For the specific queries used when benchmarking we choose to create 2 query-sets of 10000 queries each. These query-sets will be referred to as the *Random Words Query-set* and *TREC Million Query-set*.

Random Words Queryset For this queryset we selected proximity queries as the query type for performance evaluation because they require access to document identifiers, term frequencies, and term positions - exercising several parts of the index structure. Query terms were randomly selected from the 10,000 most frequent English words[13], using a fixed random seed to ensure reproducibility.

Each proximity query was constructed as follows:

1. Two randomly chosen words.
2. A proximity constraint of three words (i.e., the two terms must appear within three words of each other).

This design stresses the search system while maintaining controlled and repeatable testing conditions.

TREC Million Queryset This queryset consists of queries taken from the TREC Million Query Track 2009[14]. Queries were picked selectively from the available collection of 40000, featuring 4 different query groups[15] of decreasing priority [4]. For

[11] `https://data.commoncrawl.org/crawl-data/CC-MAIN-2025-05/index.html`

[12] `https://data.commoncrawl.org/crawl-data/CC-MAIN-2016-44/index.html`

[13] `https://www.mit.edu/~ecprice/wordlist.10000`

[14] `https://trec.nist.gov/data/million.query09.html`

[15] Queries where ranked into priority groups by Carterette et al. [4] based on how important they felt it was to test that specific query.

our needs we chose to include the three highest priority groups, adding up to 10,000 queries in total. Queries are ran as either single term- or phrase queries, depending on the number of terms in each query. Figure 3.1 shows the query distribution by group and Figure 3.2 the average term count per respective query group.

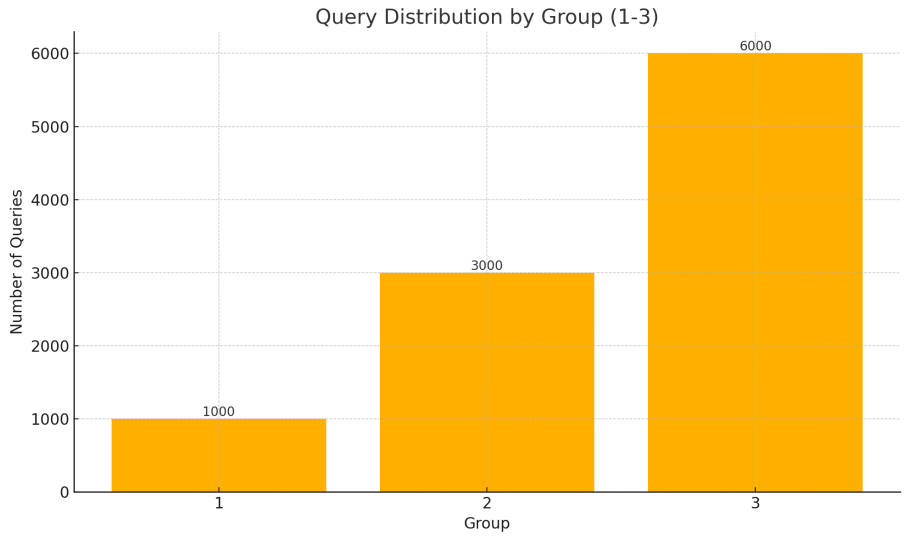


Figure 3.1: Query Distribution by Group for TREC Million Queryset

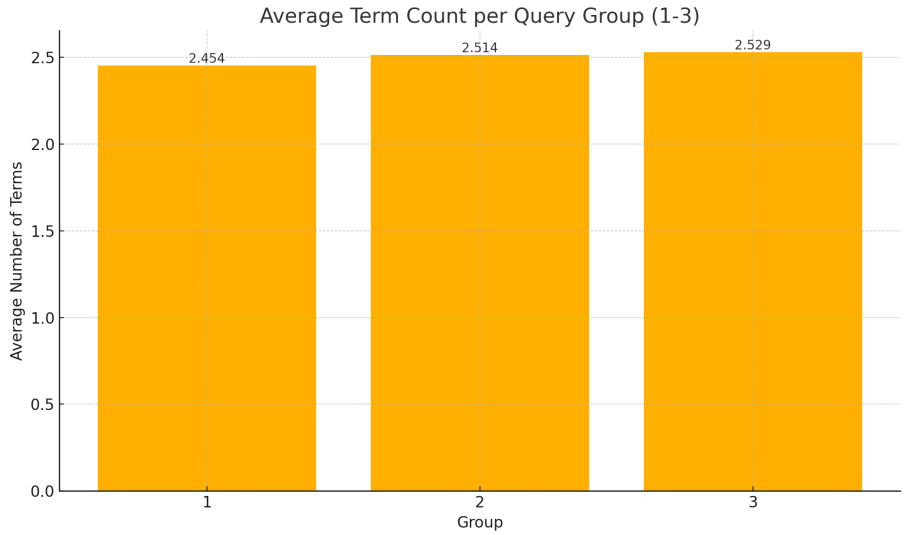


Figure 3.2: Average Term Count per Query Group for TREC Million Queryset

3.2.6 Configuration Summary

A summary of all parameters and their possible values can be seen in table 3.1 below. The purpose of this table is to provide a quick summary of the benchmark configuration parameters used in this project.

Table 3.1: Table showing all parameters of the benchmarking setup together with their possible values.

Parameter	**Value(s)**
Integer Algorithms	FOR, PFOR, Simple8b, VarByte, NewPFOR, FastPFOR
Dictionary Algorithms	LZ4, Snappy, zstd
Datasets	CommonCrawl-2025, CommonCrawl-2016
Dataset Sizes (GB)	10, 100
Querysets	Random Words Query-set, TREC Million Query-set
k-values	1, 10, 100

3.2.7 Energy Consumption

As outlined in Section 1.1, this study includes energy consumption as an additional performance metric, offering insight into the overall efficiency of each compression algorithm. While traditional metrics such as index size, indexing speed, and query latency capture system-level performance, energy usage provides a complementary view — particularly relevant in large-scale or resource-constrained environments. By measuring how much energy is consumed during both indexing and query workloads, we aim to evaluate the practical cost of each technique beyond runtime alone.

3.2.7.1 Measuring Energy Consumption

Energy measurements were performed using JoularJX[16], which captures the JVMs system-wide power draw via OS-level (Intel RAPL drivers on GNU/Linux) energy counters [28]. We repeated each benchmark—indexing and querying—using the same modified Apache Lucene Core engine and workload configurations as described in Section 3.2, ensuring a consistent and realistic workload during measurements. Benchmarks were invoked via a shell wrapper script that launches a fresh JVM and JoularJX instance for each benchmark run—indexing or querying—to avoid carry-over effects and ensure measurement isolation. The version of JoularJX used was 3.0.1.

3.2.7.2 Hardware

Due to limitations of our primary benchmarking server (virtualized environment and lack of driver access), energy tests were conducted on a dedicated desktop PC running Ubuntu 22.04 LTS. The system specifications are:

- CPU: Intel Core i9-11900F @ 2.50 GHz
- RAM: 32 GB DDR4 @ 3200 MHz
- Storage: 1024 GB NVMe SSD @ Read/Write Speed: 2100/1700 MB/s

To reduce measurement noise, we disabled dynamic frequency scaling (Turbo Boost, Cool'n'Quiet), suspended unnecessary background services, and ran each test in an

[16] `https://github.com/joular/joularjx`

isolated runlevel. No external peripherals were connected during energy benchmarking.

3.3 Evaluation

As mentioned in previous sections, the specific dataset and queries used when benchmarking can have a extensive impact on the subsequent results. In order to make accurate and meaningful comparisons it is therefore crucial to only compare results ran on identical benchmark configurations. To achieve this and make the research reproducible, we set up a detailed benchmarking configuration described in Section 3.2.

Evaluation and comparison of techniques will first be carried out individually on integer algorithms and dictionary algorithms in separate sections. Additionally in these sections, each of the key metrics presented in our Aim Section 1.1 are presented and evaluated separately. After evaluating the metrics separately a combined evaluation will be presented where we analyze gains, tradeoffs and drawbacks of different algorithm configurations against each other. The combined evaluation will later be used as the foundation and basis of justifications for the decisions made when constructing the algorithm selection model.

3.4 Algorithm Selection Model

The goal of the algorithm selection model is to provide a clear and practical method for choosing a suitable combination of compression algorithms based on specific performance priorities. To achieve this, we constructed a decision tree designed to guide users by asking simple, sequential questions tied directly to system requirements and workload characteristics.

The structure of the decision tree is grounded in the four key metrics evaluated in this study: index size, indexing speed, query latency and energy consumption. These metrics reflect the most common trade-offs in real-world IR systems — balancing storage, throughput, responsiveness, and operational cost. To ensure the model remains generalizable and intuitive, we arranged the decision points in a strategic order, starting with the factor most likely to constrain system design (index size) and then consider the others in order of typical deployment concerns. Index size was chosen as the first decision point due to its direct correlation with system-level constraints, such as memory capacity, cache locality, and storage limits. Furthermore, results presented later in Section 4 shows a clear distinction in algorithm combinations that prioritize compression, allowing this first branch to lead to an unambiguous recommendation.

If index size is not the overriding concern, the model proceeds to assess whether energy consumption takes precedence over query latency or indexing speed. If energy use is paramount, the model recommends the combination with the lowest

measured joules per workload. Otherwise, the tree proceeds to distinguish whether query latency or indexing speed is most important, and then applies a final check of whether a smaller index remains a secondary consideration. The combination of these decision points covers the practical performance dimensions most relevant to IR deployments, and each path in the tree leads to the compression pair that best aligns with the specified priorities.

The decision tree does not claim to be exhaustive or to guarantee globally optimal selections in all environments. Rather, it reflects the best-performing combinations among those tested in the current work and is intended as a practical aid. Future work could explore dynamic or learned decision trees trained on workload-specific benchmarks to tailor compression strategies to operational metrics.

3.5 Validity of Results

In the following subsections, we examine factors that may affect the validity of our results. We identify potential sources of inaccuracy, discuss parameters that could influence performance outcomes, and explain how these concerns were addressed or mitigated during the study.

3.5.1 Software

When evaluating the effectiveness of compression algorithms for inverted indexes, it is important to recognize that results can vary significantly depending on the software framework used. For example, Apache Lucene may behave quite differently from other systems due to its specific implementation choices. Lucene is written in Java, and its memory management, data structures, and processing patterns may influence compression outcomes in ways that differ from software developed in other languages or environments. Consequently, compression ratios and query performance observed in Lucene may not directly apply to other systems. Therefore, results obtained in one framework should not be generalized without careful consideration of the underlying software architecture.

3.5.2 Hardware

Performance results can vary significantly depending on the hardware on which the system is executed. For example, systems with limited memory may yield substantially different outcomes compared to those with ample memory, particularly for memory-intensive operations such as compression and decompression. Since most information retrieval systems are deployed on server-grade hardware in real-world scenarios, the benchmarks in this study were conducted on a server to more accurately reflect practical usage conditions.

3.5.3 Data- and Querysets

The choice of dataset and query set can have a significant impact on the evaluation of compression algorithms for inverted indexes. Document characteristics such as size, structure and vocabulary distribution influence how well different algorithms perform. Likewise, the query workload—its frequency distribution, length, and type—can affect query latency and compression-related trade-offs. To ensure a robust evaluation, this study uses multiple data- and querysets.

4
Results & Evaluation

In this section, we present the objective results obtained from all integer and dictionary compression algorithm benchmarks, together with a brief analysis of those results. All benchmarks were conducted according to the configuration detailed in Section 3.2. The results are first divided into integer and dictionary compression subsections (Sections 4.1 and 4.2) to assess the algorithms' impacts in isolation. Following these, a combined evaluation is presented, highlighting the best-performing algorithm combinations and the trade-offs observed between them. Within the integer and dictionary sections, we report comprehensive performance results across our four key metrics: index size, indexing speed, query latency and energy consumption. Each metric and algorithm was rigorously assessed using two large-scale datasets—sized 10 GB and 100 GB—to evaluate scalability and operational effectiveness in production-like environments. Additionally, as described in Section 3.2.5, two query sets with different characteristics were used to capture both random and sequential access patterns.

Throughout the section, results are presented based on the 2025 dataset. Subsection 4.3 provides a validation of these findings through supplementary tests conducted on the 2016 dataset. The section concludes with a decision-support model in the form of a compression algorithm selection tree.

4.1 Integer Compression

This section presents the benchmark results of integer compression algorithms in isolation from the dictionary compression algorithms.

4.1.1 Index Size

Table 4.1 presents the size of the inverted index in gigabytes (β) for each tested integer compression algorithm on two CommonCrawl datasets: a 10 GB and a 100 GB subset. The baseline corresponds to the uncompressed index.

Across both datasets, all tested compression schemes significantly reduced the index size relative to the baseline. For the 10 GB dataset, FASTPFOR achieved the best compression, reducing the index size by 52.2%, followed closely by Simple8b and NewPFOR. Similar trends were observed in the 100 GB dataset, where FASTPFOR again yielded the smallest index (31.9 GB), corresponding to a 57.3% reduction

compared to the baseline.

Overall, all methods provided >48% decrease in index size on the smaller dataset and >52% on the larger one, demonstrating the effectiveness of integer compression in reducing the storage footprint of inverted indexes.

Table 4.1: Inverted index size β (in GB) for CommonCrawl-2025 for 10GB and 100GB dataset using the given integer compression algorithm

Name		**CommonCrawl-2025 10 GB**			**CommonCrawl-2025 100 GB**	
		β	%		β	%
Baseline		7.38	–		74.7	–
FOR		3.83	-48.10%		35.2	-52.89%
PFOR		3.70	-49.87%		33.9	-54.59%
Simple8b		3.55	-51.90%		32.1	-57.03%
VarByte		3.67	-50.27%		33.5	-55.21%
NewPFOR		3.60	-51.22%		32.9	-55.96%
FastPFOR		**3.53**	**-52.17%**		**31.9**	**-57.29%**

4.1.2 Indexing Time

Table 4.2 presents the total time (ς) required to construct the inverted index using each compression algorithm, measured in seconds, for both the 10 GB and 100 GB CommonCrawl datasets. The baseline represents the uncompressed indexing process, with a time of 808.32 and 8044.17 seconds, respectively.

The results show that most compression algorithms incur minimal overhead or even slight improvements in indexing time compared to the baseline. For example, FOR and VarByte achieved small reductions in indexing time on the 100 GB dataset, with decreases of 3.3% and 1.4%, respectively. PFOR and NewPFOR also showed modest improvements in indexing speed, despite their more complex compression logic.

FastPFOR was the only algorithm that significantly increased indexing time on the larger dataset, incurring a 6.1% overhead. Conversely, on the 10 GB dataset, all methods exhibited slightly higher indexing times relative to the baseline, with the largest increase being 3.9% for NewPFD. These results suggest that while compression adds some processing cost, it remains efficient and scales well with input size.

Table 4.2: Indexing Time ς (in s) for CommonCrawl-2025 (10 GB and 100 GB)

Name	**CommonCrawl-2025 10 GB**		**CommonCrawl-2025 100 GB**	
	ς	%	ς	%
Baseline	808.32	–	8044.17	–
FOR	**813.35**	**+0.62%**	7775.41	-3.34%
PFOR	830.90	+2.80%	**7687.29**	**-4.44%**
Simple8b	830.24	+2.71%	8106.81	+0.78%
VarByte	818.16	+1.22%	7929.38	-1.43%
NewPFOR	839.78	+3.90%	7814.31	-2.86%
FastPFOR	832.65	+3.01%	8531.44	+6.05%

4.1.3 Query Latency

Table 4.3 presents average query latency measurements (in milliseconds) for the CommonCrawl-2025 100GB index across two query workloads. Our analysis reveals that all compression algorithms introduce performance overhead compared to the uncompressed baseline, though to varying degrees.

In the random-words workload, FOR-based lightweight codecs demonstrate negligible slowdown (FOR: +2.1%, PFOR: +5.9%), while VarByte exhibits substantial performance degradation (+174%). Simple8b and NewPFOR show moderate increases (+60% and +16%, respectively), with FastPFOR falling between these groups (+20.6%).

The TREC-million workload yields smaller performance penalties across all methods: VarByte decreases to +41.8%, Simple8b to +24.2%, and FOR variants cluster around +7-8%. Figure 4.3 illustrates these performance relationships. In absolute terms, FOR-encoded queries require only marginally more time than the baseline (2.28 µs versus 2.23 µs on random queries), whereas VarByte more than triples query processing time.

Table 4.3: Average query time δ (in ms) for CommonCrawl-2025 (100 GB)

Name		CommonCrawl-2025 100 GB				
		Random Words			TREC Million	
		δ (ms)	%		δ (ms)	%
Baseline		2.05	–		5.43	–
FOR		**2.28**	**+11.2%**		4.56	-16.0%
PFOR		2.36	+15.1%		4.57	-15.9%
Simple8b		3.57	+74.2%		5.25	-3.3%
VarByte		6.11	+198.1%		5.99	+10.3%
NewPFOR		2.59	+26.3%		**4.52**	**-16.8%**
FastPFOR		2.69	+31.2%		4.56	-16.0%

4.1.4 Energy Consumption

Table 4.4 shows the total energy consumption β (in kilojoules) of the program when running the CommonCrawl-2025 10 GB and 100 GB benchmarks using the different integer compression algorithms. The baseline represents the energy consumption of the program when using no compression.

Looking at the table we can see that FOR is the clear winner, even outperforming the baseline with a -4.7% consumption on the 100 GB dataset size. It can also be observed that the PFOR family (PFOR, NewPFOR and FastPFOR) has the worst performance in general, incurring a substantial increase in energy consumption on the 10 GB dataset size as compared to the others. However, it is worth noting that all algorithms perform >3% better on the 100 GB dataset size, with only NewPFOR and FastPFOR still inuring an increase of 2.6% and 3.4% respectively.

From this data we can conclude that FOR performs the best in regards to energy consumption over both dataset sizes, and that NewPFOR and FastPFOR perform the worst. If lower energy consumption is a target then FOR should be used, if not, other metrics should dominate the choice of algorithm.

Table 4.4: Energy consumption ϵ (in kilojoules) for CommonCrawl-2025 (10 GB and 100 GB) using integer compression algorithms

Name	**CommonCrawl-2025 10 GB**		**CommonCrawl-2025 100 GB**	
	ϵ	%	ϵ	%
Baseline	62.44	–	688.60	–
FOR	**62.72**	**+0.45%**	**656.18**	**-4.71%**
PFOR	66.16	+5.95%	681.45	-1.04%
Simple8b	63.27	+1.32%	656.35	-4.68%
VarByte	64.04	+2.55%	671.65	-1.44%
NewPFOR	66.87	+7.09%	706.29	+2.56%
FastPFOR	66.79	+6.97%	711.65	+3.35%

4.2 Dictionary Compression

This section presents the benchmark results of the dictionary compression algorithms in isolation from the integer compression algorithms.

4.2.1 Index Size

Table 4.5 presents the size of the inverted index in gigabytes (β) for each tested dictionary compression algorithm on two CommonCrawl datasets: a 10 GB and a 100 GB subset. The baseline corresponds to the uncompressed index using 32-bit integers.

As can be seen in the table, all compression algorithms reduced the inverted index size by a similar amount across the board compared to the baseline. For the smaller dataset sizes such as 10 GB, LZ4 and Snappy perform very closely to each other, with zstd outperforming both slightly at a -3.6% difference in size. Although, a pattern can be observed of LZ4 performing slightly better than Snappy and zstd with growing dataset sizes. In the 100 GB dataset an improvement of 3.0% in LZ4 can be observed, while zstd and Snappy lose some performance at 3.2% and 2.0% respectively.

In conclusion, for dictionary compression, zstd seems to provide the best reduction in the inverted index size with LZ4 coming in a close second for larger datasets, and lastly Snappy coming in a close last place. From this we can conclude that zstd is the best pick if aiming for pure index size reduction. If not, LZ4 and Snappy can then be chosen based on the dataset size or other desired metrics.

Table 4.5: Inverted index size β (in GB) for CommonCrawl-2025 (10 GB and 100 GB) using dictionary compression algorithms

Name	**CommonCrawl-2025 10 GB**		**CommonCrawl-2025 100 GB**	
	β	%	β	%
None	3.64	–	32.97	–
LZ4	3.56	-2.20%	31.99	-2.97%
ZSTD	**3.51**	**-3.57%**	**31.93**	**-3.15%**
Snappy	3.56	-2.20%	32.31	-2.00%

4.2.2 Indexing Time

Table 4.6 presents the total time ς required to construct the inverted index using each compression algorithm, measured in seconds, for both the 10 GB and 100 GB CommonCrawl datasets. The baseline represents the uncompressed indexing process, with a time of 792.05 and 7887.42 seconds, respectively.

From the table we can see that all algorithms incur some overhead compared to the baseline on the 10 GB collection. For all algorithms it can be observed that the overhead shrinks with growing datasets, especially for LZ4 and Snappy whose overhead is decreased by 7.0% and 5.9% respectively on the 100 GB dataset, ultimately leading to a net negative change compared to the baseline. Zstd generally gives the largest overhead compared over all datasets and sizes, this can most likely be attributed to its higher compression goal.

The relative difference in indexing speed between zstd and the two other algorithms is quite large and if the user is concerned or focused on the indexing speed, Snappy or LZ4 should be chosen. If indexing speed is the main focus Snappy is the best contender.

Table 4.6: Indexing time ς (in s) for CommonCrawl-2025 (10 GB and 100 GB) using dictionary compression algorithms

Name	**CommonCrawl-2025 10 GB**		**CommonCrawl-2025 100 GB**	
	ς	%	ς	%
Baseline	792.05	–	7887.42	–
LZ4	841.20	+6.21%	7828.77	-0.74%
ZSTD	890.26	+12.40%	8638.60	+9.53%
Snappy	**805.47**	**+1.69%**	**7558.98**	**-4.17%**

4.2.3 Query Latency

Table 4.7 presents the average query latency (β) measured in nano seconds, observed when running Random Words and TREC Million querysets respectively on the CommonCrawl-2025 100 GB dataset. The baseline represents the query latency observed on the uncompressed index using 32-bit integers. The same benchmark results from the 10 GB dataset size can be seen in the appendix A.2, the general pattern observed was the same change but exaggerated compared to the 100 GB dataset.

Here it can be observed that the only algorithm incurring a large overhead on the query latency is zstd. It can also be observed that both querysets have similar changes compared to the baseline over all algorithms. The algorithm with the largest variance between querysets is Snappy at 7.1%. Moreover it can be observed that algorithms seems to have the largest impact on query latency in the TREC Million queryset.

The clear loser of this metric is zstd who seems to incur a noticeable overhead on all queries and should not be chosen if query latency is of importance. The choice between LZ4 and Snappy is a bit harder with LZ4 winning in the Random Words queryset by 1.8% and Snappy winning in the TREC Millin queryset by 2.3%. Therefore, the expected type of queries should govern the choice between LZ4 and Snappy.

Table 4.7: Average query latency δ (in ns) for Random Words and TREC Million queryset on CommonCrawl-2025 100 GB dataset.

Name		**CommonCrawl-2025 100 GB**				
		Random Words			**TREC Million**	
		δ	%		δ	%
Baseline		2.23	–		4.42	–
LZ4		**2.20**	**-1.31%**		4.22	-4.31%
ZSTD		2.39	+7.19%		4.85	+9.67%
Snappy		2.24	+0.53%		**4.13**	**-6.58%**

4.2.4 Energy Consumption

Table 4.4 presents the total energy consumption β (in kilojoules) of the program when running the CommonCrawl-2025 10 GB and 100 GB benchmarks using the different dictionary compression algorithms. The baseline represents the energy consumption of the program when using no compression.

For the both dataset sizes Snappy is the clear winner excluding the baseline, with an increase in energy consumption of only 3.2% and 0.3% respectively. Over both dataset sizes it is also clear that zstd is the most intensive algorithm consuming the most energy by quite a large margin. Although, the evolution of energy consump-

tion for all algorithms seems to point to a decrease with growing dataset sizes.

According to these results Snappy is the most resource efficient algorithm with the lowest energy consumption over all benchmarks. If energy consumption is of primary concern, Snappy should be chosen.

Table 4.8: Energy consumption ϵ (in kilojoules) for CommonCrawl-2025 in sizes 10 GB and 100 GB using dictionary compression algorithms

Name		**CommonCrawl-2025 10 GB**			**CommonCrawl-2025 100 GB**	
		ϵ	%		ϵ	%
Baseline		60.60	–		640.36	–
LZ4		64.04	+5.67%		664.01	+3.69%
ZSTD		70.32	+16.04%		720.32	+12.49%
Snappy		**62.54**	**+3.20%**		**642.16**	**+0.28%**

4.3 Evaluation on a Second Dataset

To validate the generalizability of our findings, we repeated the full benchmarking process on a second dataset: *CommonCrawl-2016*. This dataset differs from the primary dataset *CommonCrawl-2025* in terms of its time of collection, the distribution of content types, and the structural characteristics of its documents. By evaluating compression performance across two independent collections, we aim to ensure that observed patterns are not artifacts of a single dataset.

The same experimental protocol was applied, including indexing with each compression algorithm, measuring index size, indexing throughput, query latency on two query sets, and energy consumption. The hardware configuration and software stack remained identical to those used in the main experiments to ensure comparability.

Results on the second dataset were largely consistent with those observed on the primary dataset. While absolute performance values varied slightly due to the difference in data, the relative rankings of the algorithms remained stable across all key metrics with some slight differences. This confirms the robustness of the observed trade-offs and supports the general applicability of our conclusions.

A detailed breakdown of the results for the second data set is provided in the tables A.1, A.2.

4.4 Combined Evaluation

This section integrates the results from Sections 4.1 and 4.2 to evaluate the combined effects of integer and dictionary compression algorithms. The goal is to identify optimal combinations based on four core metrics: index size, indexing time, query latency and energy consumption. The full index is comprised of both the postings list (compressed with an integer codec) and dictionary data (compressed using a general-purpose dictionary codec). Therefore, the interaction between the two layers of compression plays a significant role in overall system performance.

Best Combination for Minimal Index Size. To achieve the smallest possible index, **FastPFOR** combined with **Zstandard (zstd)** yields the best results. FastPFOR offers one of the strongest compression ratios for postings lists, reducing index size by 57.3% on the CommonCrawl-2025 100 GB dataset. Zstd, while slower, provides the best dictionary compression, further shrinking the overall index by 3.2%. However, this configuration incurs higher indexing time and increased query latency, especially on real-time queries, due to the overhead introduced by Zstd's more complex decompression pipeline.

Best Combination for Fastest Indexing. If the goal is to minimize indexing time, **FOR** paired with **Snappy** is optimal. FOR and PFOR even outperformed the uncompressed baseline on large datasets by accelerating index build time. Snappy, while slightly less space-efficient than LZ4 or Zstd, provides excellent encoding speed, contributing to a fast end-to-end indexing process. This pairing offers adequate compression (particularly from the integer codec) while achieving the shortest indexing durations across both datasets.

Best Combination for Lowest Query Latency. When minimizing query latency is the primary objective, the combination of **FOR** with **LZ4** or **Snappy** performs best. FOR offers consistently low overhead for decompression due to its simple fixed-width format, and LZ4/Snappy decompress dictionaries with minimal latency. On average, this combination introduces only 2–3% latency overhead over the baseline on both Random Words and TREC queries respectively, making it ideal for latency-sensitive applications such as real-time search systems.

Best Combination for Lowest Energy Consumption. The combination that yields the lowest energy consumption is **FOR** paired with **Snappy**. While FOR offers the lowest compression ratio compression ratio—approximately 4.2% less than FastPFOR—it still delivers excellent performance in both indexing speed and query latency. Snappy, in turn, not only contributes to the lowest overall energy usage but also provides the fastest indexing times, strong query performance, and a respectable compression ratio. Together, they achieve the most energy-efficient configuration by a significant margin, without sacrificing competitiveness across the other key metrics.

Best Balanced Trade-off. For applications requiring a balance between compression, indexing time, query latency and energy consumption **PFOR** paired with

Snappy is recommended. PFOR compresses more aggressively than FOR while maintaining good decoding performance and moderate energy consumption. Snappy complements it with fast dictionary compression and decompression, keeping both indexing and query latency within reasonable bounds, while also boasting the lowest energy consumption. This combination achieves >50% total index size reduction without significantly degrading speed, making it suitable for general-purpose search engine deployments.

Summary. The table below summarizes the best-performing combinations across different metrics:

Table 4.9: Optimal codec combinations for different performance objectives.

Objective	Integer Codec	Dictionary Codec
Minimal Index Size	FastPFOR	Zstd
Lowest Query Latency	FOR	LZ4 / Snappy
Fastest Indexing	FOR / FastPFOR	Snappy
Lowest Energy Consumption	FOR	Snappy
Balanced Performance	NewPFOR	LZ4 / Snappy

4.5 Compression Algorithm Selection Tree

To assist in selecting an appropriate compression strategy for inverted index storage, we propose a decision tree model, illustrated in Figure 4.1. This model guides users in choosing a suitable combination of integer and dictionary compression algorithms based on four core system considerations: index size, query latency, indexing speed, and energy consumption.

The order of questions in the decision tree reflects a practical prioritization of typical system constraints. We begin with index size because storage—whether on disk, in memory, or during transfer—is often a hard limit. If the index can't fit within those boundaries, nothing else matters, so this becomes the first and most critical decision point. After that, we consider energy consumption. While not always a strict constraint, it's an increasingly important factor in many systems, especially those concerned with efficiency or sustainability. However, it tends to be more flexible than storage, so it takes second place. Only after addressing these fundamental constraints do we focus on performance trade-offs like query latency versus indexing speed. These are usually more flexible and can be adjusted by tuning parameters, optimizing algorithms, or scaling resources, making them easier to refine once the harder limits on size and energy are met.

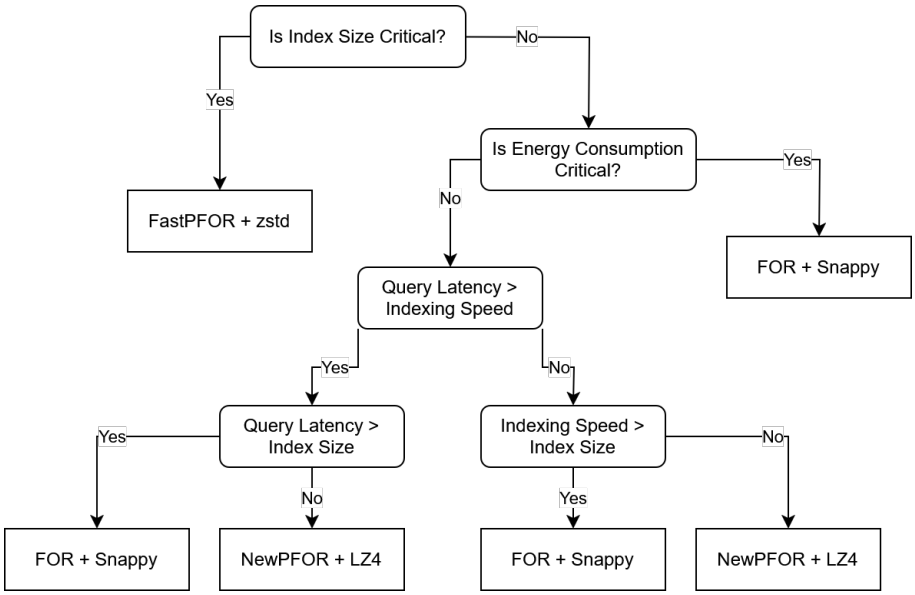


Figure 4.1: Compression Algorithm Selection Tree

1. Is Index Size Critical? The first decision assesses whether minimizing the on-disk and in-memory footprint is the overriding concern. This scenario is common in systems with constrained storage resources or environments where index replication, transfer (network and/or disk), or in-memory caching is a limiting factor.

- **Yes:** → Choose: **FastPFOR + Zstandard (ZSTD)**. FastPFOR offers excellent compression ratios while still enabling fast decompression, and ZSTD further reduces size due to its strong dictionary compression. Ultimately this combination provides an index approximately 6.4% smaller than the average of other combinations while retaining reasonable de and -encoding performance.
- **No:** If storage is not the main concern, further decisions are made based on performance trade-offs.

2. Is Energy Consumption Critical? The next decision assesses whether minimizing energy use is the primary constraint — typical in battery-powered, carbon-budgeted or cost-metered deployments.

- **Yes:** → Choose: **FOR + Snappy**. Both FOR and Snappy offer the lowest energy consumption by a fair margin while still delivering encoding speeds within 1.5% of the top performers and the best query latencies. Together they provide the fastest query latencies and an energy consumption 3.7% lower than average while still while maintaining competitive performance across encoding and compression.
- **No:** If energy consumption is not the main concern either, proceed to the latency vs. throughput branch.

3. Query Latency vs. Indexing Speed. If neither index size nor energy consumption is the key concern, the decision tree now identifies whether query latency

or indexing speed is the dominant performance requirement.

- **Query Latency > Indexing Speed:** This applies to real-time search systems where fast response times are essential. The next and last decision compares the relative importance of query latency versus index size:
  - **Query Latency > Index Size:** Choose: **FOR + Snappy**. This combination delivers the lowest query latency—approximately 30.5% faster than the average—at the cost of a slightly larger index size.
  - **Index Size > Query Latency:** Choose: **NewPFOR + LZ4**. This combination provides a strong balance of size and speed, with approximately 28.9% faster query latencies than average and an index 2.0% smaller than FOR + Snappy.

- **Indexing Speed > Query Latency:** Relevant for systems where frequent indexing or document ingestion occurs, it is crucial to minimize encoding time. The next and last decision compares the relative importance of indexing speed versus index size:
  - **Indexing Speed > Index Size:** Choose: **FOR + Snappy**. This combination provides the fastest decoding and one of the fastest encoding speeds, supporting rapid ingestion and providing on average 4.1% faster indexing time.
  - **Index Size > Indexing Speed:** Choose: **NewPFOR + LZ4**. This pair slightly slows indexing in exchange for better compression, yielding indexing time 2.0% faster than average and an index 2.0% smaller than FOR + Snappy.

Each decision path reflects the practical trade-offs observed in our benchmark results—recommending the algorithm combination that performs the best in regards to the chosen priorities—and is intended to simplify deployment decisions in environments with varying performance and resource constraints.

5
Conclusion

In this thesis, we evaluated a range of integer and dictionary compression algorithms to analyze how they affect four key performance metrics; index size, indexing time, query latency and energy consumption. This was done by implementing the compression algorithms into an IR system called Apache Lucene and rigorously benchmarking them to record their affect on the performance metrics. The data was then used to evaluate what algorithms excel in which categories and how they can be paired to target different performance goals. Finally, we present a simple algorithm selection model to guide users in choosing the right algorithms for their use cases. According to our results, the choice of algorithms can have a high impact on all of the performance metrics and a conscious decision can lead to greatly improved performance in targeted areas while trading off reasonable performance in others.

5.1 Research Questions

In this section we will attempt to answer the research questions posed in the aim of this study by presenting the relevant results and findings from our research.

RQ 1: How do different inverted index compression algorithms affect the performance of inverted index search engines in terms of index size, search time, indexing time and energy consumption?

In answer to RQ1, we found that all tested integer codecs significantly reduce index size (typically >50 %), with modest effect on indexing time (mostly a few percent overhead). Query latency, however, varied widely: FOR-based methods incurred only a few percent overhead, whereas VarByte increased latency dramatically on short queries (up to 2–3× slower). Simple8b/NewPFOR fell in between. As for energy consumption, PFOR and newer versions of it (NewPFOR, FastPFOR) lead to the largest increase in energy consumption by a fair margin. However, it was noted that all algorithms-dictionary algorithms included-performed significantly better in this regard when ran on a larger dataset. For dictionary compression, we observed that zstd achieves the smallest index but incurs the highest CPU cost-performing a larger workload and consequently drawing more energy than others-while LZ4 and Snappy gave good speed and energy consumption with slightly less compression. These results demonstrate that the choice of compression algorithms can dramatically change the performance of these metrics by trading one area for another.

RQ 2: Can a selection model be designed to select the optimal index compression algorithm based on constraints for compression ratio, search time, indexing time and energy consumption in inverted index search engines?

Regarding RQ2, our results suggest simple guidelines for selecting algorithms under different constraints. If search speed is critical, one should prefer fast-decode codecs (e.g. PFOR or FOR for posting lists, and LZ4/Snappy for dictionary), since they keep query latency close to baseline. If storage reduction is the priority, then heavier codecs (Simple8b, NewPFOR, zstd) are optimal, as they maximize compression at the cost of CPU time. Lastly, if energy consumption or indexing time is of essence, more lightweight codecs such as Snappy or FOR are subtile, usually trading off some compression in exchange. In practice, a decision model can be built by ranking algorithms along these axes (as illustrated by the selection flowchart in Figure 4.5). For example, for a predominantly read-only search service with tight memory, one might choose NewPFOR+Zstd; for an application demanding lowest query latency, one might use PFORDelta with LZ4 and so on.

5.2 Future Work

In the following subsections, we highlight several promising directions to extend the study of inverted-index compression in order to achieve further gains in speed, throughput or energy efficiency.

5.2.1 Machine-Learned Compression Models

One promising direction is the integration of machine-learned compression models, such as those exemplified by LeCo [25]. These techniques apply learned transforms to remove statistical redundancies more effectively than traditional codecs. Although still in their infancy and not yet widely adopted in industrial search engines, such approaches have the potential to outperform hand-crafted methods by adapting dynamically to data characteristics and query patterns.

5.2.2 Hybrid or Specialized Dictionary Coders

Another direction worth exploring is the use of hybrid or specialized dictionary coders like Re-Pair and DINT. Re-Pair builds a grammar from frequently occurring symbol pairs, offering high compression at the expense of decoding speed — making it a candidate for archival systems. In contrast, DINT provides very fast decoding by storing common gap patterns in a global dictionary and referencing them via fixed-size codes, showing competitive compression with extremely low decode overhead [29]. Evaluating these approaches under modern workloads could reveal whether they are practical for use in real-time or semi-real-time IR systems.

5.2.3 Advanced Integer Compression Schemes

Beyond traditional block coders (FOR/PFOR) and selector schemes (Simple8b), two promising avenues exist:

- **SIMD-Enhanced Elias–Fano Variants:** Partitioned Elias–Fano (PEF) already combines clustering and random access. Porting PEF to wide-vector instructions (AVX-512) could yield sub-0.5 ns/int decode speeds with near-entropy space.
- **Learned Integer Coders:** Extending learned-model approaches to postings (e.g. using neural networks to predict delta distributions) could further reduce bitrates by capturing long-range correlations — an area ripe for exploration given recent advances in sequence modeling.

5.2.4 Hardware-Accelerated Compression

Moreover, as energy consumption becomes a growing concern in data center and cloud infrastructure, future studies could assess hardware-accelerated compression or compression-aware CPU design. For instance, leveraging SIMD, ARM's NEON extensions, or dedicated decompression hardware (like Intel's QuickAssist or programmable logic blocks in FPGA accelerators) could yield further improvements in energy efficiency and even throughput or speed.

5.2.5 Workload-Aware Adaptive Compression

Finally, dynamic or hybrid strategies—where the system switches codecs based on real-time metrics (e.g., query rate, CPU load, or temperature)—or predicted access patterns (random vs sequential), query volume and document turnover rate, could further improve performance and deliver the best of all worlds. Implementing an adaptive layer-in Lucene or any other IR system-that profiles workload patterns and selects optimal posting and dictionary codecs on the fly would be a compelling next step. These strategies might involve runtime codec switching, multi-tier compression layouts, or integration with learned indexes.

5.3 Final Conclusion

This thesis demonstrates that compression techniques play a crucial role in optimizing the performance of inverted index search engines. By applying both integer and dictionary compression algorithms within Apache Lucene, we quantified their impact across four key metrics: index size, indexing time, query latency, and energy consumption.

Our results show that no single algorithm performs optimally across all dimensions. Instead, performance characteristics vary significantly depending on workload and system priorities. Integer codecs like FOR and PFOR offer low-latency, high-throughput performance, whereas alternatives such as Simple8b and NewPFOR yield higher compression at the cost of speed. Similarly, among dictionary-based

codecs, LZ4 and Snappy provide fast indexing and low energy usage, while Zstd offers superior compression ratios.

By analyzing trade-offs, we developed a decision model to guide codec selection under various performance constraints. This model allows practitioners to balance compression efficiency with operational requirements, such as response time or resource consumption.

Ultimately, the careful selection of compression algorithms enables informed performance tuning of search engines. Our study provides both quantitative benchmarks and practical guidelines to support such decisions, paving the way for scalable, efficient, and adaptable search infrastructure.

Bibliography

[1] Vo Ngoc Anh and Alistair Moffat. Index compression using 64-bit words. *Software: Practice and Experience*, 40(2):131–147, 2010.

[2] Murat Aslanyürek and Altan Mesut. A new method for short text compression. *IEEE Access*, 11:141022–141035, 2023.

[3] Matěj Bartík, Sven Ubik, and Pavel Kubalik. Lz4 compression algorithm on fpga. In *2015 IEEE International Conference on Electronics, Circuits, and Systems (ICECS)*, pages 179–182, 2015.

[4] Ben Carterette, Virgil Pavluy, Hui Fang, and Evangelos Kanoulas. Million query track 2009 overview. 01 2009.

[5] Matteo Catena, Craig Macdonald, and Iadh Ounis. On inverted index compression for search engine efficiency. In Maarten de Rijke, Tom Kenter, Arjen P. de Vries, ChengXiang Zhai, Franciska de Jong, Kira Radinsky, and Katja Hofmann, editors, *Advances in Information Retrieval*, pages 359–371, Cham, 2014. Springer International Publishing.

[6] Rui Chen and Lihao Xu. Snappyr: A new high-speed lossless data compression algorithm. In *2023 Data Compression Conference (DCC)*, pages 334–334, 2023.

[7] Yann. Collet. Lz4 - extremely fast compression. `https://github.com/lz4/lz4`, 2022. GitHub repository, accessed May, 2025.

[8] Yann. Collet. Lz4 block format description. `https://github.com/lz4/lz4/blob/dev/doc/lz4_Block_format.md`, 2022. GitHub repository, accessed May, 2025.

[9] Yann. Collet. New generation entropy coders. `https://github.com/Cyan4973/FiniteStateEntropy`, 2022. GitHub repository, accessed May, 2025.

[10] Yann. Collet. Lz4 frame format description. `https://github.com/lz4/lz4/blob/dev/doc/lz4_Frame_format.md`, 2023. GitHub repository, accessed May, 2025.

[11] Yann Collet and Murray Kucherawy. Zstandard Compression and the 'application/zstd' Media Type. RFC 8878, February 2021.

[12] Common Crawl. Common crawl. https://commoncrawl.org/.

[13] L. Peter Deutsch. GZIP file format specification version 4.3. RFC 1952, May 1996.

[14] Jarek Duda, Khalid Tahboub, Neeraj J. Gadgil, and Edward J. Delp. The use of asymmetric numeral systems as an accurate replacement for huffman coding. In *2015 Picture Coding Symposium (PCS)*, pages 65–69, 2015.

[15] John Yiannis Falk Scholer, Hugh E. Williams and Justin Zobel. Compression of inverted indexes for fast query evaluation. *SIGIR '02: Proceedings of the 25th*

annual international ACM SIGIR conference on Research and development in information retrieval, 2002.

[16] Fortune Business Insights. Data storage market size, share & segmentation by storage medium (direct attached storage, network attached storage, software defined storage and hyper converged storage, cloud storage, storage area network), by enterprise size (mid-size, soho, large enterprises), by industry (bfsi, it and telecommunication, healthcare and life sciences, manufacturing, retail and consumer goods, governments & public sector, media and entertainment, others), by region and global forecast for 2024–2032. Market research report, Fortune Business Insights Pvt. Ltd., 2024.

[17] Veluchamy Glory and Sandanam Domnic. Compressing inverted index using optimal fastpfor. *Journal of Information Processing Vol.23 No.2*, 2015.

[18] Roi Blanco González. *Index Compression for Information Retrieval Systems*. PhD thesis, University of A Coruña, 2008.

[19] Google. Snappy compressed format description. Technical report, GitHub, 2011.

[20] Athira Gopinath and Ravisankar. M. Comparison of lossless data compression techniques. *Proceedings of the Fifth International Conference on Inventive Computation Technologies (ICICT-2020)*, 2020.

[21] Adrien Grand. Store compression in lucene and elasticsearch. *Elastic Blog*, 2015.

[22] David A. Huffman. A method for the construction of minimum-redundancy codes. *Proceedings of the IRE*, 40(9):1098–1101, 1952.

[23] D. Lemire and L. Boytsov. Decoding billions of integers per second through vectorization. *Software: Practice and Experience*, 45(1):1–29, May 2013.

[24] Daniel Lemire, Leonid Boytsov, and Nathan Kurz. Simd compression and the intersection of sorted integers. *Software: Practice and Experience*, 46(6):723–749, April 2015.

[25] Yihao Liu, Xinyu Zeng, and Huanchen Zhang. Leco: Lightweight compression via learning serial correlations. *arXiv preprint arXiv:2306.15374*, 2023.

[26] Apache Lucene. Apache lucene. https://lucene.apache.org.

[27] Ajit Kumar Mahapatra and Sitanath Biswas. Inverted indexes: Types and techniques. *International Journal of Computer Science Issues*, 8, 07 2011.

[28] Adel Noureddine. PowerJoular and JoularJX: Multi-Platform Software Power Monitoring Tools. In *18th International Conference on Intelligent Environments*, Biarritz, France, June 2022.

[29] Giulio Ermanno Pibiri and Rossano Venturini. Techniques for inverted index compression. *ACM Computing Surveys*, 53(6):1–36, December 2020.

[30] S. Biplab Raut. Aocl-compression - a high performance optimized lossless data compression library. In *2023 IEEE High Performance Extreme Computing Conference (HPEC)*, pages 1–7, 2023.

[31] StarRocks. Data compression. *StarRock Docs*.

[32] Petroc Taylor. Volume of data/information created, captured, copied, and consumed worldwide from 2010 to 2023, with forecasts from 2024 to 2028. Technical report, Statista, 2024.

[33] Andrew Trotman. Compression, simd, and postings lists. In *Proceedings of the 19th Australasian Document Computing Symposium*, ADCS '14, page 50–57, New York, NY, USA, 2014. Association for Computing Machinery.

[34] Jianguo Wang, Chunbin Lin, Ruining He, Moojin Chae, Yannis Papakonstantinou, and Steven Swanson. Milc: inverted list compression in memory. *Proc. VLDB Endow.*, 10(8):853–864, April 2017.

[35] Hugh E. Williams and Justin Zobel. Compressing integers for fast file access. *The Computer Journal*, 42(3):193–201, 01 1999.

[36] xxHash Project. xxHash. `https://xxhash.com/`. Accessed May 2025.

[37] J. Ziv and A. Lempel. A universal algorithm for sequential data compression. *IEEE Transactions on Information Theory*, 23(3):337–343, 1977.

[38] M. Zukowski, S. Heman, Niels Nes, and Peter Boncz. Super-scalar ram-cpu cache compression. 05 2006.

A

Appendix 1

Table A.1: Performance of integer compression methods on CommonCrawl-2016-44 dataset (Index size in GB, indexing time in s, query latency in ms, EC = Energy Consumption).

CommonCrawl-2016-44 (10 GB)					
Method	**Index Size (GB)**	**Index Time (s)**	**Random Words (ms)**	**TREC Million (ms)**	**EC (Kj)**
Baseline	7.10	782	0.55	1.25	2.82
FOR	3.61 (-49.2%)	817 (+4.5%)	**0.55** (0.0%)	**1.19** (-4.8%)	**2.80** (-0.7%)
PFOR	3.51 (-50.6%)	832 (+6.4%)	0.57 (+3.6%)	1.21 (-3.2%)	2.90 (+2.8%)
SIMPLE8b	3.42 (-51.8%)	**748** (-4.3%)	0.78 (+41.8%)	1.38 (+10.4%)	2.99 (+6.0%)
VARBYTE	3.55 (-50.0%)	761 (-2.7%)	1.11 (+101.8%)	1.42 (+13.6%)	3.03 (+7.4%)
NEWPFOR	3.48 (-51.0%)	761 (-2.7%)	0.67 (+21.8%)	1.27 (+1.6%)	2.93 (+3.9%)
FASTPFOR	**3.40** (-52.1%)	775 (-0.9%)	0.68 (+23.6%)	1.36 (+8.8%)	3.11 (+10.3%)

Table A.2: Performance of dictionary compression methods on CommonCrawl-2016-44 dataset (Index size in GB, indexing time in seconds, query latency in ms, EC = Energy Consumption).

CommonCrawl-2016-44 (10 GB)					
Name	**Index Size (GB)**	**Indexing Time (s)**	**Random Words (ms)**	**TREC Million (ms)**	**EC (Kj)**
Baseline	3.46	759	0.58	1.20	2.82
LZ4	**3.39** (-2.0%)	782 (+3.0%)	0.57 (-1.7%)	1.22 (+1.7%)	2.77 (-1.8%)
ZSTD	3.34 (-3.5%)	815 (+7.4%)	0.99 (+70.7%)	1.49 (+24.2%)	3.21 (+13.8%)
SNAPPY	3.39 (-2.0%)	**769** (+1.3%)	**0.55** (-5.2%)	**1.21** (+0.8%)	**2.53** (-10.3%)

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING
CHALMERS UNIVERSITY OF TECHNOLOGY
Gothenburg, Sweden
www.chalmers.se

CHALMERS
UNIVERSITY OF TECHNOLOGY