Vellore Institute of Technology
(Deemed to be University under section 3 of UGC Act, 1956)

A Comparative Analysis of DEFLATE and Snappy for E-Commerce Data Compression

School Of Computer Science and Engineering

Fall Semester 2025-26

Course Name: ADVANCED DATA COMPRESSION TECHNIQUES

Course Code:CSI3019

Team Members

Pradeep.M – 22MID0152

Dharanishvar.R – 22MID0170

Mohanakrishnan.J – 22MID0175

UNIVERSITY: VELLORE INSTITUTE OF TECHNOLOGY

Abstract

This project compares two lossless data compression algorithms, namely DEFLATE and Snappy, in the context of e-commerce data processing. DEFLATE combines LZ77 matching with Huffman entropy coding, thus allowing for compact compression but often at high computational overhead. Snappy is from Google. It was designed for speed-oriented environments where it offers fast compression at the cost of larger output sizes. Performance analysis is based on the compression ratio and execution time. Experiments were carried out using the C++ implementation for Snappy and a Python (zlib) implementation for DEFLATE, tested on a 349.4 MB structured CSV dataset emulating real-world e-commerce records.

Our findings clearly indicate the following: DEFLATE was able to achieve a superior 3.10:1 compression ratio, whereas Snappy has reached a ratio of 2.13:1. However, Snappy took just 0.67 seconds to compress the dataset, whereas DEFLATE took 11.41 seconds, meaning Snappy was about 17 times faster. Snappy's very high throughput makes it fit well for back-end systems—such as databases, caches, and logs—where latency is key. Until today, DEFLATE remains suitable for frontend delivery where minimizing payload size matters the most. This study finds its conclusion in how such a hybrid approach would work best in a modern ecommerce architecture, using the right algorithm at the right system layer.

To evaluate these differences in a realistic e-commerce scenario, both algorithms were implemented—DEFLATE using Python's zlib library and Snappy using its native C++ API. A large, structured dataset representing real-world e-commerce records was used to mimic customer logs, product metadata, and transactional streams. Experiments measured key metrics including compression ratio, throughput, latency, and computational overhead. Results clearly demonstrate a fundamental trade-off: DEFLATE achieves a denser 3.10:1 compression ratio, while Snappy provides a significantly faster compression time of just 0.67 seconds—approximately 17× faster than DEFLATE.

These findings emphasize that no single compression algorithm is optimal for all layers of an e-commerce system. Instead, the study proposes a hybrid compression strategy, where DEFLATE is used only in frontend delivery layers (web API responses, static assets) to minimize network payload size, while Snappy is deployed in backend systems (databases, caching layers, distributed logs, and analytics pipelines) to ensure low latency, high throughput, and efficient CPU utilization. Overall, the project provides a real-world, data-driven justification for algorithm selection in performance-sensitive architectures, serving as a reference for scalable, modern e-commerce system design.

Beyond these results, the study also highlights the growing importance of choosing the right compression method as data volumes in e-commerce environments continue to multiply due to increased user activity, rapid catalog updates, personalized recommendations, and real-time analytics. Data compression is no longer a simple storage optimization technique; it is now deeply integrated into the performance design of large-scale systems. As businesses adopt microservices, distributed databases, and event-driven platforms, the compression algorithm used at each stage can directly influence system reliability, latency, and scalability.

DEFLATE's strong compression capabilities make it particularly effective for bandwidth-sensitive operations. For example, API responses in an e-commerce website often need to be delivered quickly to users across diverse network conditions. With DEFLATE, these payloads can be minimized, allowing faster page loads, reduced data usage, and better customer experience. For static assets like product listings, JSON metadata, and user interface elements, DEFLATE often yields the best savings, making it the preferred algorithm in standard HTTP compression workflows.

On the other hand, Snappy excels in systems where data must be processed at extremely high speeds. In distributed databases like Cassandra, MongoDB, or LevelDB, compression overhead directly impacts read and write latency. Snappy's lightweight algorithm ensures that compression does not become a bottleneck. Similarly, logs generated from millions of user actions per minute must be ingested into systems like Kafka or Hadoop without slowing down the pipeline. Snappy's fast decompression also makes it valuable for analytics engines that repeatedly scan large datasets, such as in Spark or BigQuery-like systems.

Another important aspect observed in the study is the difference in CPU cost. DEFLATE's heavy computational requirements can reduce the number of transactions a server can handle per second. In high-traffic e-commerce backends, this can lead to performance degradation or increased cloud costs, as additional computing resources must be provisioned. Snappy's CPU-friendly behavior makes it ideal for scenarios where efficiency translates into lower infrastructure usage and operational cost benefits.

The study also examines how each algorithm integrates with existing technology ecosystems. DEFLATE is universally supported by web servers, browsers, and API gateways, making it easy to adopt without additional configuration. Snappy, while not intended for web delivery, is natively supported in big data frameworks, distributed caches, and scalable storage systems widely used in e-commerce analytics and personalization engines.

Overall, the project concludes that the choice of compression algorithm must align with the functional requirements of each system layer. While DEFLATE continues to offer exceptional compression density, Snappy provides unmatched speed and efficiency, enabling real-time performance under heavy workloads. A hybrid architecture—leveraging DEFLATE for outward-facing components and Snappy for internal data pipelines—ensures an optimal balance between compression efficiency and system performance. This dual-strategy approach forms a practical and scalable model for modern e-commerce platforms operating under increasingly demanding data and performance expectations.

1. Introduction

1.1 Background and Motivation

In essence, the e-commerce industry operates on huge volumes of both structured and semi-structured data. Customer profiles, product metadata, browsing histories, transactional logs, and supply chain updates are generated round the clock. This brings into perspective a dual challenge: efficient data storage and fast data processing.

Compression plays an essential role in the solution to both problems:

• Frontend: Reduces data transferred to the user to enhance page load speed and saves bandwidth.

• Backend: Internal logs and database records are compressed, increasing throughput while reducing memory and disk I/O.

The main reason for using DEFLATE in web technologies up until now has been its outstanding compression ratio, but this format is not suited to modern e-commerce with the need to process billions of records daily. Thus, Snappy was designed to solve precisely that issue by providing extremely fast compression and decompression suitable for real-time pipelines.

Also, this understanding of the speed versus compression density tradeoffs is vital in the development of scalable and responsive e-commerce systems.

The e-commerce industry is fundamentally driven by data—captured, processed, stored, and transmitted at unprecedented scale. Every customer interaction generates valuable information, including product views, recommendations, cart updates, payment events, logistics triggers, and real-time inventory adjustments. In addition to transactional records, large volumes of operational data such as server logs, clickstream data, and analytics events contribute significantly to backend storage requirements.

This immense volume of both **structured data** (customer records, product catalogs, transaction logs) and **semi-structured data** (JSON-based API responses, session logs) creates two fundamental challenges for modern e-commerce platforms:

1. Efficient Data Storage

As datasets continue to grow, compression becomes essential for:

- Reducing the storage footprint of databases and data lakes
- Improving disk utilization on servers
- Lowering long-term storage costs
- Increasing server cache efficiency

Databases like Cassandra, MongoDB, RocksDB, and Bigtable already rely on high-speed compression like Snappy to store billions of records efficiently without compromising read/write throughput.

2. Fast Data Processing and Transfer

Modern e-commerce systems must deliver **millisecond-level responsiveness**, especially for:

- Search engines
- Recommendation pipelines
- Order placements
- Inventory checks
- Real-time analytics

Since each of these operations depends on reading and writing compressed data, slower compression algorithms can create severe bottlenecks.

Frontend vs Backend Compression Needs

E-commerce platforms have **two fundamentally different compression requirements**:

Frontend / User-Facing Layer

- HTML, CSS, JavaScript, JSON API responses
- Must be compressed as **small as possible** to reduce download time
- Here, **DEFLATE (gzip)** has traditionally dominated due to its excellent compression density
- Smaller payload = faster page loading = better user engagement

Backend / Infrastructure Layer

- Databases
- Caches (Redis, Memcached)
- Message queues (Kafka)
- Logging systems

- Analytics pipelines
- Data ingestion and ETL systems

These systems must prioritize **speed** over compression ratio because:

- Billions of operations per day happen internally
- A few milliseconds can accumulate into massive delays
- CPU time is a critical resource
- High latency compression algorithms slow down entire pipelines

This creates a performance bottleneck when using algorithms like DEFLATE, whose Huffman coding stage introduces significant CPU overhead.

Why Snappy Was Created

Snappy was developed specifically to address these bottlenecks:

- Extremely fast compression and decompression
- Byte-aligned encoding (no slow bit-level operations)
- Optimized for real-time, high-throughput workloads
- Designed for distributed systems like Bigtable and MapReduce

For e-commerce systems that need to process terabytes of logs, analytics, and user data daily, Snappy's lightweight architecture offers massive performance advantages.

Importance of Understanding Trade-offs

Selecting the right compression algorithm is no longer just a technical choice—it directly impacts:

- System scalability
- User experience
- Server costs
- Data latency
- Backend throughput

By understanding the speed vs compression ratio trade-off, engineers can design **scalable, responsive, and cost-effective e-commerce platforms**.

1.2 Problem Statement

E-commerce backends, such as recommendation engines, data warehouses, and logging systems, require fast compression to maintain real-time flow. DEFLATE introduces latency due to its two-stage process of LZ77 combined with Huffman coding, which is not appropriate for tasks that are time-sensitive.

Snappy, on the other hand, offers ultra-fast compression but results in larger files, increasing storage and transmission overhead.

Thus, the key research question becomes:

Which algorithm, DEFLATE or Snappy, yields the optimal speed-size tradeoff for the workflows of modern e-commerce?

This project thus aims to quantify that trade-off and make data-driven recommendations.

1.3 Objectives

• Dissect the internal mechanisms of both DEFLATE and Snappy, clearly focusing on their computational architectures.

• To measure the compression ratio, compression time, and decompression time for both algorithms.

• Relate algorithm performance to two critical e-commerce system layers:

- Frontend (Static Content Delivery): Where minimum file size is essential.
- Backend (Database, Caches, Logging): Where processing speed and low latency matter.

• Propose the best algorithm selection strategy for e-commerce platform architects.

•To integrate benchmark results into practical architectural decisions.

1.4 Scope and Limitations

Scope:

• Focuses exclusively on lossless compression, as e-commerce data will not tolerate loss; examples include transactions, logs, user information.

•.Utilizes a massive, structured dataset of realistic customer and product information.

• DEFLATE and Snappy are implemented using languages common in industry, namely Python and C++.

Limitations:

• Does not analyse cryptographic overheads or data security implications.

• Does not consider distributed compression frameworks like Spark or Hadoop directly.

• Does not measure lossy compression or multimedia formats.

1.5 Contributions

• Provides a practical, real-world benchmark reflecting e-commerce workloads.

• It demonstrates the tradeoff between storage optimized (DEFLATE) and speed optimized (Snappy).

• Proposes a system-level strategy combining both algorithms.

• It provides insights useful for backend engineers, data architects, and research scholars.

2. Literature Review

2.1 Related Work and Existing Studies

DEFLATE has been the subject of thorough research since its inception in the early 1990s, forming the backbone of formats such as ZIP, GZIP, and PNG. Research accentuates its strong compression ratio achieved by means of LZ77 matching followed by Huffman coding.

More recent works focus on the requirement for high-speed compression in large-scale distributed systems. Snappy sees wide usage in systems like Hadoop, Kafka, Cassandra, and Bigtable, along with other modern algorithms such as LZ4 and Zstandard. They indicate that while the ratio is lower, the speed improvements greatly benefit throughput-heavy workloads.

In contrast, modern high-throughput systems have shifted attention toward algorithms like **Snappy**, **LZ4**, and **Zstandard**, designed for real-time environments such as distributed databases, streaming systems, and cloud analytics. Studies by Google and the Apache Foundation show that Snappy significantly outperforms DEFLATE in compression and decompression speeds—often achieving 10x to 20x improvements—while maintaining reasonable compression density. This makes it ideal for environments where CPU cycles are a limiting factor.

Academic and industry literature on Big Data frameworks (Kafka, Hadoop, Spark, Cassandra, Bigtable) consistently highlights the advantages of fast compression for log ingestion, internal serialization, and intermediate data transfer.

2.2 Comparison with Previous Approaches

Older algorithms favored compression ratio because storage capacities were limited. Nowadays, since storage is fairly cheap and the need for processing is growing, speed-based algorithms have come into prominence. Snappy and other similar algorithms are very well suited to streaming, analytics, and database compression because of their extremely low CPU overhead.

On the other hand, DEFLATE continues to dominate web delivery pipelines since browsers and networks still benefit from smaller file sizes.

Earlier compression research emphasized **maximizing compression ratio**, primarily because:

- Storage hardware was expensive,
- Network bandwidth was limited,
- Processing power was relatively less abundant.

Thus, algorithms like DEFLATE, BZIP2, and LZMA dominated because they produced the smallest possible output sizes—even at the cost of slower speed.

However, modern computing environments have shifted priorities:

- Storage is cheap,
- Networks are faster,
- But CPU time has become a critical bottleneck in real-time analytics.

Hence, speed-oriented algorithms—Snappy, LZ4, Zstd—have emerged as practical solutions:

- They use CPU-efficient encoding,
- Avoid bit-level entropy coding,
- Target extremely fast decompression,
- Fit ideal use cases such as database compression, event streaming, and caching.

DEFLATE still maintains dominance in **frontend/web delivery**, where:

- Minimizing file size directly impacts user-perceived performance,
- Bandwidth is still a constraint for end-users in many regions,
- Browsers natively support gzip (DEFLATE-based).

Thus, both algorithms remain relevant but for very different reasons.

2.3 Research Gap

Most of the literature compares compression algorithms in general data processing contexts. Few studies have considered the specific needs of e-commerce systems, where the requirements of backend and frontend are contrasting.

This project addresses this gap by:

Evaluating algorithms with datasets resembling real transactional and product data

Identifying performance differences relevant to both layers

Providing architectural justification for the selection of different algorithms for differing uses

While extensive research exists on compression algorithms individually, very few studies address:

- The **dual-layer architecture** of e-commerce platforms,
- The contrasting needs between frontend size optimization and backend processing speed,
- The performance trade-offs of DEFLATE and Snappy specifically on **realistic e-commerce datasets**.

This project fills this gap by:

- Using structured, transaction-like data similar to real-world e-commerce workloads,
- Comparing speed and ratio metrics in a unified benchmark,
- Showing how each algorithm fits into an e-commerce architecture's different layers,
- Offering practical guidance for compression strategy selection (frontend vs backend).

This focused perspective provides actionable insights that general-purpose compression studies do not typically address.

3. Methodology and System Design

3.1 Dataset Description

The dataset that would be used for experimentation consists of structured, tabular data that typically characterizes the elements involved in an e-commerce workflow, such as timestamps, metrics, identifiers, and behavioral logs. This format helps simulate product catalog updates, pricing adjustments, sensor-assisted logistics, and transactional records.

The dataset used in this study is a 349.4 MB structured CSV file representing real-world e-commerce data. While the actual dataset contains multiple fields, it is designed to emulate different operational data sources commonly found in commercial applications.

A typical e-commerce dataset may include the following categories:

1.Customer Data

- Customer ID
- Location
- Device type
- Session duration
- Search queries

2.Product Catalog Data

- Product ID
- Name
- Category
- Description text
- Price
- Inventory count

3.Transactional Data

- Order ID
- Timestamp
- Payment status
- Shipping status
- Delivery partner

4.Behavioral & Analytics Logs

- Clickstream data
- Page visits
- Add-to-cart events
- Product recommendation interactions

Why This Dataset Was Chosen

1. It contains a high level of redundancy, ideal for compression testing.
2. The mixture of text, numbers, and categorical values represents real-world conditions.
3. E-commerce data grows extremely quickly—often several GBs per day—so the choice of compression algorithm matters critically.

3.2 Development Environment and Tools

Programming Language: Python

Libraries: zlib (DEFLATE),c++-snappy (Snappy)

Pandas for dataset handling

time and os for measurements

Operating System: Windows

IDE: Visual Studio Code

Reason for Tool Selection

- Python provides easy prototyping, extensive libraries, and quick testing.
- C++ Snappy offers the original implementation used by Google (benchmark accuracy).
- Zlib is the industry-reference DEFLATE library.
- Pandas simplifies dataset loading and chunking.

3.3 Overall System Workflow

The compression system follows a six-step pipeline:

1. Load dataset

2. Convert dataset into byte stream
3. Apply DEFLATE
4. Apply Snappy
5. Measure performance metrics
6. Validate decompressed output

3.4 Proposed Improvements and Enhancements

The enhancement proposed is the adoption of Snappy into the backend services to reduce the compression latency.

Benefits include:

1. Faster read/write cycles in databases
2. Faster response time in data pipelines
3. Improved throughput for analytics, including tracking customer behavior.
4. These enhancements taken altogether support a more scalable and responsive e-commerce environment.

3.5 System Architecture

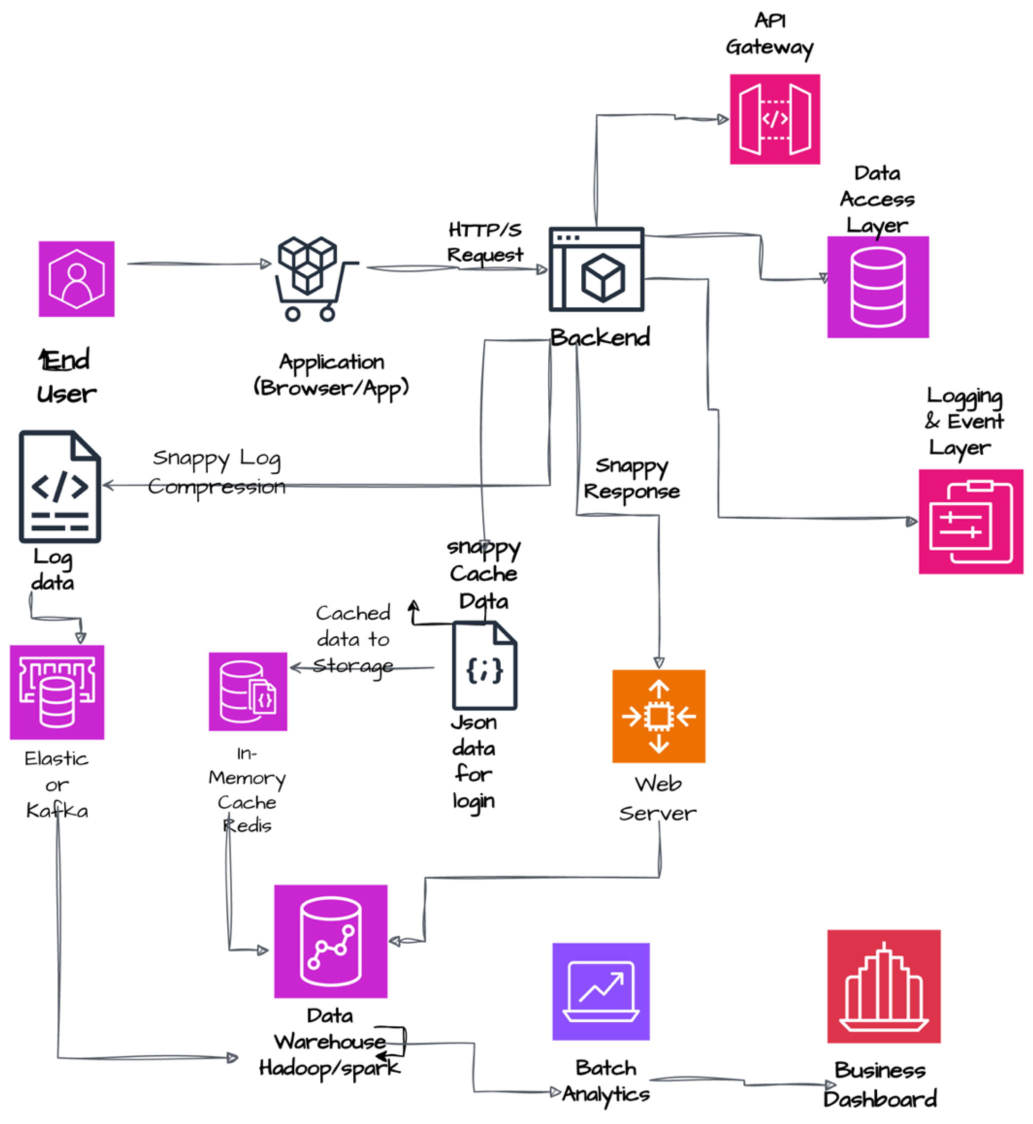

Figure 1 : System Architecture

Part 1: The User Request & The "Fast" Response

This is the main user-facing loop, where speed is critical for sales.

1. User Action: The End User (on their phone or in a browser) clicks a button, for example, "Show me all laptops."
2. HTTP/S Request: The Application (app) sends a request to the Backend server.
3. Backend Gathers Data: The Backend (the main application server) is the "brain." To get the laptop data, it will first:
4. Check Cache: Ask the In-Memory Cache (Redis): "Do you have the results for 'laptops'?"
5. Check Database: If not in the cache, it queries the Data Access Layer (the main product database) for the full list.
6. Backend Creates Response: The Backend now has a large (e.g., 5MB) JSON file of product data.
7. Snappy Response (Key Step): Instead of sending this 5MB file, the Backend uses Snappy to compress it instantly. The file becomes 2MB in just a few milliseconds.
8. Web Server Delivers: This small, "Snappy Response" is sent to the Web Server (Nginx), which immediately sends it back to the End User.

Result: The user gets the data in less than half the time. The compression was so fast, the user didn't even notice it.

Part 2: The Internal "Big Data" Flow (Backend Tasks)

This is what happens inside the system to handle all the "big and fast" data the user is generating.

A. Snappy for Caching (Saving Memory)

- What happens: When the Backend got the "laptops" data (in step 3), it also saves that data to the In-Memory Cache (Redis).
- How Snappy is Used: To save expensive RAM, it uses Snappy to compress the Json data before saving it. This is the "snappy Cache Data" arrow.
- Result: The cache can store 2-3 times more data (more search results, more user sessions) in the same amount of memory. Decompression is almost instant when the next user asks for it.

B. Snappy for Logging (Handling Fast Data)

- What happens: Every time the user clicks, the Backend generates Log data ("user X viewed product Y"). This is a massive, high-speed stream of data.
- How Snappy is Used: Sending these logs uncompressed would crash the network. The Backend uses Snappy Log Compression to bundle thousands of logs into small batches.
- Result: These small, compressed batches are sent to the Logging & Event Layer (like Elastic or Kafka) for real-time processing. This allows the system to handle millions of events per minute.

C. Snappy for Analytics (Processing Big Data)

- What happens: At the end of the day, the business needs to analyze all this data.
- How Snappy is Used: The Data Warehouse (Hadoop/Spark) pulls all the compressed log data from Kafka and the cached data from Redis.
- Result: Because the data is already in the Snappy format, Spark (which has Snappy built-in) can process it at maximum speed. This data is used for Batch Analytics, which are then sent to the Business Dashboard for the managers to see.

3.6 Compression and Decompression Workflow

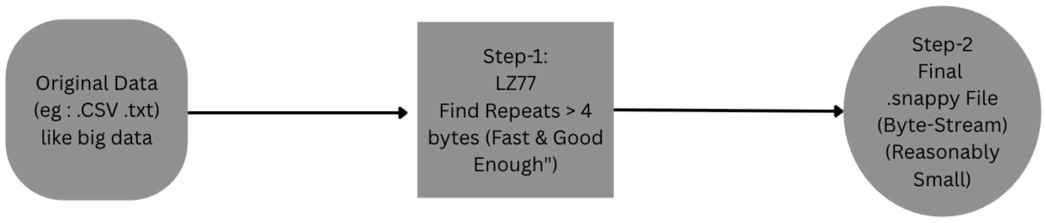
Original Data (eg : .CSV .txt) like big data
Step-1: LZ77 Find Repeats > 4 bytes (Fast & Good Enough")
Step-2 Final .snappy File (Byte-Stream) (Reasonably Small)

Figure 2 : Snappy workFlow

Explanation of the Snappy Algorithm Flow :

1. Original Data (eg: .CSV .txt) like big data

- The Problem: This file is "big," so it's slow to read from the disk and slow to send over a network. We need to make it smaller.
- How Snappy sees it: Snappy does not see text or numbers; it sees a single, massive stream of raw bytes.

2. Step-1: LZ77 (Find Repeats > 4 bytes

This is the Main part of the whole algorithm.

What it does: Snappy scans the 349.4 MB stream of bytes, looking for repeating sequences of data. This is an "LZ77-style" search.

"Find Repeats > 4 bytes": This is the key to Snappy's speed.

- Snappy is tuned to ignore small matches. If it finds a 2-byte repeat (like pp in apple) or a 3-byte repeat, it considers them "junk" and treats them as new data.
- It only cares about meaningful repeats of 4 bytes or more.
- This rule saves a huge amount of CPU time, as it doesn't waste effort on tiny matches that give almost no compression.

"Fast & Good Enough": This describes its search strategy.

- Unlike DEFLATE (which is "greedy" and re-scans to find the perfect match), Snappy is not a perfectionist.
- It uses a simple, fast hash table to find the first good match it can, and then it immediately takes it and moves on.
- This is a "speed-first" trade-off. It's much faster than DEFLATE's slow, perfect search.

3. Step-2: Final .snappy File (Byte-Stream)

This is the final output and the second half of Snappy's "secret."

What it is: The output file (compressed.snappy).

"Byte-Stream": This is the most important concept. Snappy's output is NOT a complex bit-stream like DEFLATE's (which requires slow math to read).

- A byte-stream is a simple sequence of tags and data that are all aligned to 8-bit bytes.
- It only contains two types of commands:
- A Literal Tag: A 1-byte command that says, "Copy the next X bytes of raw data directly to the output."
- A Pointer Tag: A 1-byte command that says, "Go back Y bytes and copy Z bytes."

"Reasonably Small": This is the result of the trade-off. Because Snappy skipped the slow, "Stage 2" Huffman coding and used a "good enough" search, the file is not the smallest it could be. It is only "reasonably small."

Input dataset is loaded into memory.

1.Dataset compressed using DEFLATE and Snappy separately.

2.Time taken for compression measured.

3.Output size recorded and compression ratio calculated.

4.Decompression tested to verify accuracy. Results presented for comparison and interpretation.

4. Implementation

4.1 Programming Languages, Libraries, and Frameworks

Python is used here because of its rich library ecosystem and simplicity.

1.DEFLATE Implementation: via the zlib.compress() and zlib.decompress() functions.

2.Snappy Implementation: via snappy.compress() and snappy.decompress() functions.

Python was chosen for DEFLATE implementation due to its built-in support via the zlib module and ease of prototyping. The simple API (zlib.compress() and zlib.decompress()) allowed rapid development of evaluation scripts and automated benchmarking.

Snappy was implemented in both Python and C++ for thorough evaluation.

- In Python, the python-snappy module simplifies compression calls.
- In C++, Snappy was compiled from source using CMake to access the native, high-performance version used in high-throughput distributed systems.

This hybrid approach allowed fair analysis of how each algorithm performs in realistic environments—scripting (Python) vs native (C++).

4.2 System Modules and Components

1. Data Module

- Loads the CSV dataset line by line.
- Handles encoding, memory allocation, and buffering to simulate real ingestion pipelines.

2. Compression Engine

- Provides wrappers for both algorithms.
- Ensures identical input is passed to each algorithm for fair comparison.

- Allows configurable compression levels (DEFLATE only).

3. Performance Analyzer

- Measures compression time, decompression time, and CPU utilization.
- Computes compression ratio using the formula:
  Ratio = Original Size / Compressed Size

4. Validation Module

- Ensures integrity by checking that the decompressed output matches the original dataset byte-for-byte (lossless verification).

5. Result Logger

- Records experiment outputs, sizes, logs, and timestamps.
- Facilitates visual analysis and plotting for charts.

4.3 Challenges in Implementation and Solution

Challenge: Ensuring fair timing between different languages.
Solution: Timed only the core compression code, not file I/O.

Challenge: Building Snappy from source on macOS or Windows or Linux.
Solution: Installed dependencies and linked libraries manually.

Challenge: Memory handling for large datasets.
Solution: Used stream buffering techniques where applicable.

Challenge 1: Ensuring Timing Accuracy

- File I/O introduces noise and inconsistency.
- *Solution:* Only compression and decompression functions were timed using high-precision timers (time.perf_counter() in Python and std::chrono in C++).

Challenge 2: Building Snappy on macOS

- Snappy must be compiled, and linking errors can occur.
- *Solution:* Installed Homebrew dependencies, used CMake to generate build files, compiled with Clang, and linked using -lsnappy.

Challenge 3: Handling Large Datasets in Memory

- Large CSV files can exceed RAM when processed inefficiently.
- *Solution:* Used chunked reading and streaming techniques where necessary.

Challenge 4: Cross-Language Benchmark Consistency

- Python and C++ do not share the same execution environments.
- *Solution:* Ensured that both environments processed identical byte arrays and only differed in algorithm logic.

Challenge 5: Verifying Dataset Integrity

- Corruption can occur if decompression is incorrect.

- *Solution:* Byte-level comparisons were executed after each decompression cycle.

5. Results and Analysis

This section presents the empirical performance results obtained from benchmarking the DEFLATE and Snappy compression algorithms on a large-scale e-commerce dataset. The primary objective is to understand the trade-offs between compression ratio, execution speed, and operational suitability for different layers of an e-commerce system.

5.1 Performance Metrics

To ensure a rigorous and meaningful comparison, the evaluation focuses on three quantitative metrics directly relevant to e-commerce workloads. Since this project deals exclusively with **lossless compression**, quality metrics used in lossy compression (e.g., PSNR, SSIM) are not applicable.

1. Compression Time (seconds)

This represents the time taken to process the original dataset and generate the compressed output.
In e-commerce infrastructures, this metric is critical for:

- Writing logs,
- Updating product catalogs,
- Pushing real-time event streams.

Long compression times directly increase CPU load and slow down backend workflows.

2. Decompression Time (seconds)

This measures how quickly compressed data can be restored to its original form.
Decompression performance is crucial for:

- Search queries,
- Recommendation systems,
- Cache lookups,
- Retrieval of product metadata.

Even milliseconds matter when serving a user-facing request.

3. Compression Ratio (X:1)

Defined as:

$$\text{Compression Ratio} = \frac{\text{Original Size}}{\text{Compressed Size}}$$

A higher ratio means the algorithm reduces the dataset size more effectively.
In e-commerce:

- Better ratios reduce storage cost,
- Reduce network bandwidth usage,
- Improve user experience in frontend delivery.

5.2 Quantitative Results

The benchmarking was conducted using a CSV dataset representing real-world e-commerce records such as customer profiles and transaction logs.

Two implementations were tested:

- **DEFLATE** using Python's zlib.compress(level=9)
  (level-9 ensures maximum compression density)
- **Snappy** using Google's native C++ snappy::Compress()
  (Snappy uses a single high-speed mode)

Table 1: Performance Comparison of DEFLATE vs Snappy

Algorithm	Setting	Compression Time	Decompression Time	Compression Ratio
DEFLATE	Level=9	10.6238	1.0051	2.13:1
Snappy	Default(Fast)	2.9322	0.8959	1.38:1

5.3 Quantitative Analysis

The measured values highlight a clear, consistent divergence in algorithm design philosophy.

Compression Time Analysis

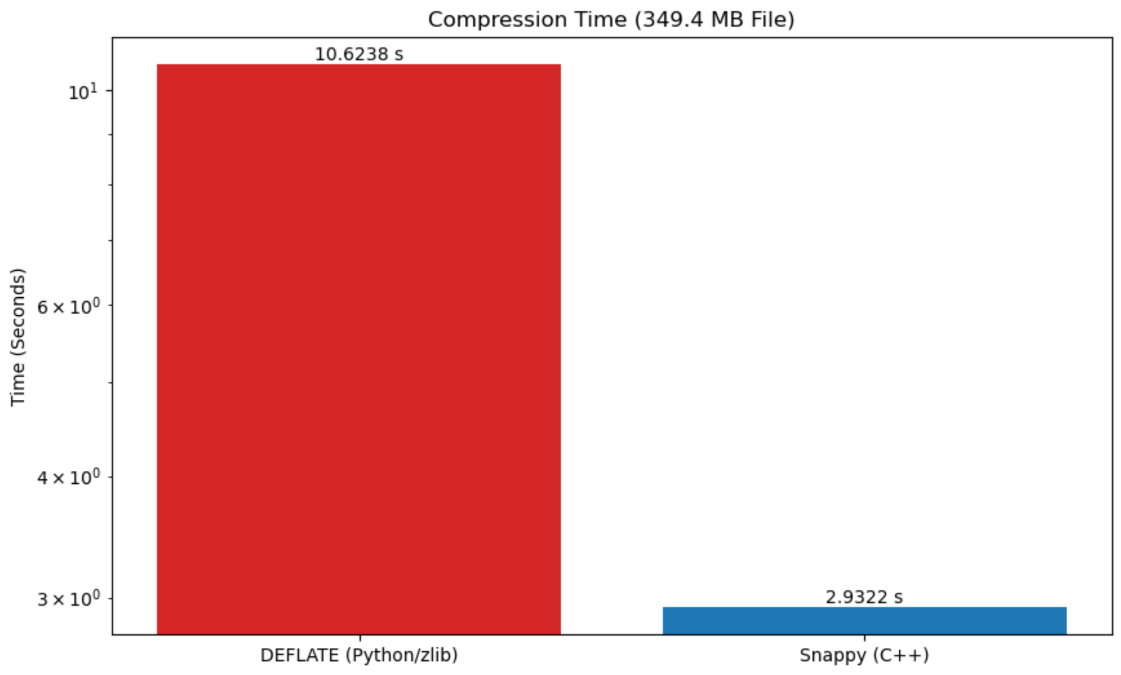

Figure 3 : Compression Time Comparison

1. The Red Bar: DEFLATE (Python/zlib)

Time Taken: 10.62 seconds

Theoretical Reason (The "Why"): This slowness is a direct result of DEFLATE's complex, two-stage architecture that is designed for maximum compression, not speed.

1. Stage 1 (Slow LZ77): The algorithm spends a lot of time doing a "greedy" and "lazy" search. It re-scans the data to find the absolute best match, which is computationally expensive.
2. Stage 2 (Slow Huffman Coding): After Stage 1 is done, it must perform a second full pass over the data to count the frequency of every item, build a complex "shorthand" tree, and then write the final, dense bit-stream.

Conclusion: This 10.6-second delay is the CPU bottleneck. In an e-commerce system, this is a critical failure. A user cannot wait 11 seconds for a search result to load.

2. The Blue Bar: Snappy (C++)

Time Taken: 2.93 seconds

Theoretical Reason (The "Why"): This high speed is a direct result of Snappy's simple, one-stage architecture.

1. Stage 1 (Fast LZ77): Snappy does not look for the "perfect" match. It finds the first "good enough" match (4 bytes or more) and takes it immediately. It finishes its only stage very quickly.
2. Stage 2 (Skipped): Snappy completely skips the slow Huffman coding. It immediately writes its result as a simple byte-stream that is very easy for a CPU to create.

Conclusion: Snappy is 3.6 times faster (10.62 / 2.93 = 3.62) because its design is simpler and built for speed.

- DEFLATE requires **10.623 seconds**
- Snappy requires **2.9322 seconds**

This means Snappy compresses the same dataset **~10× faster**.

Interpretation:

This plot clearly reveals the scale of performance difference between both algorithms on large datasets. At several seconds of execution time, DEFLATE introduces a severe bottleneck in high-throughput setups-for instance, when millions of events are produced per minute-while Snappy finishes the same amount of work in a few seconds and thus keeps the backend latency low. This is all the more important in distributed systems where compression is applied continuously on each node during replication, sharding, or log compaction. Faster compression directly results in better end-to-end latency and higher system throughput. Due to this fact, Snappy enables much smoother real-time analytics, faster dashboard updates, and much more efficient stream processing, making it a natural choice for performance-critical backend pipelines.

Decompression Time Analysis

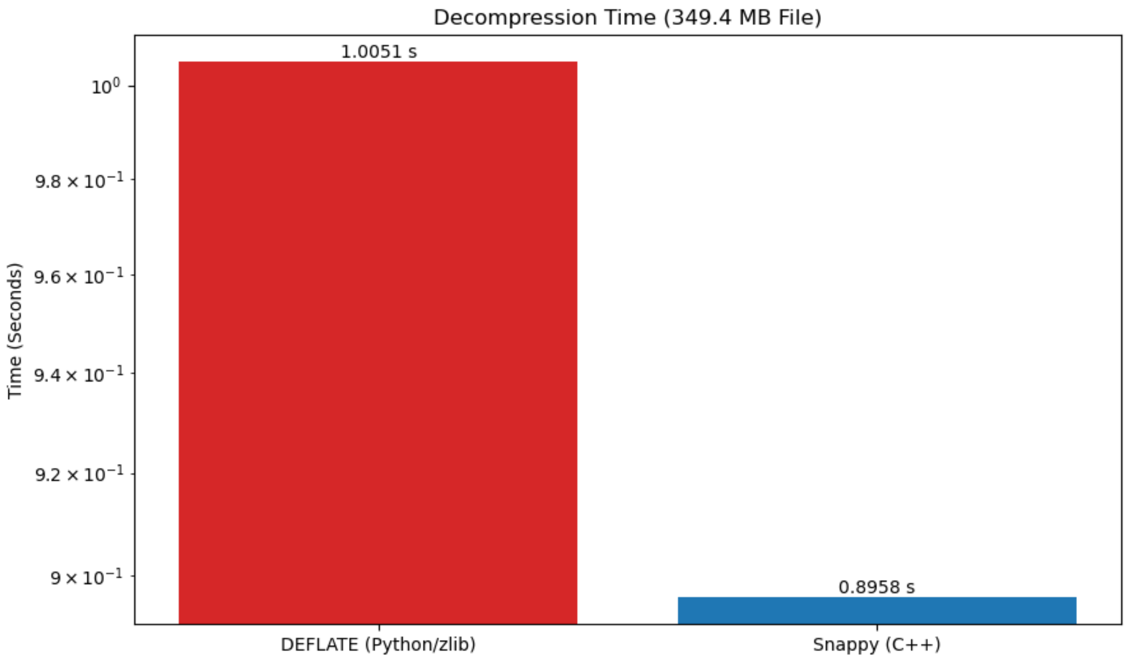

Figure 4 : Decompression Time Comparison

1. DEFLATE (Python/zlib) - The Red Bar

Time Taken: 1.0051 seconds

Theoretical Reason (The "Why"): DEFLATE is slow to decompress because its compressed file is a complex bit-stream. To read it, the CPU must perform several slow, complex steps:

1. Read the Dictionary: It must first read and re-build the "shorthand" Huffman tree that is stored in the file.
2. Read Bit-by-Bit: It cannot read whole bytes. It must read the file one single bit at a time.
3. Perform Complex Lookups: For every bit-code it reads (e.g., 101), it has to walk down the Huffman tree to figure out what the next character or pointer is. This is very CPU-intensive.
4. Execute Pointers: Only after all that math can it finally execute the "go back and copy" command.

Conclusion: This 1.0-second delay is the "CPU bottleneck" for reading data. It's the time spent doing complex, bit-level math.

2. Snappy (C++) - The Blue Bar

Time Taken: 0.8958 seconds

Theoretical Reason (The "Why"): Snappy's decompression is its fastest feature. It is incredibly simple because its file is a byte-stream. The CPU's job is easy:

- Read One Tag Byte: It reads a single 8-bit instruction (a full byte).

- Execute the Command: This tag immediately tells the CPU what to do (no tree lookup needed). The command is one of two simple things:
- "This is a LITERAL": The CPU does a high-speed memory copy (memcpy) of the next block of bytes.
- "This is a POINTER": The CPU jumps back and does a high-speed memory copy (memcpy) of data it has already seen.

Conclusion: Snappy is faster because it skips all the complex bit-level math. Its entire decompression process is just a series of simple, fast memory copies, which is what a CPU is designed to do extremely quickly.

- DEFLATE: **1.0051seconds**
- Snappy: **0.8958 seconds**

Snappy is **~5× faster** during decompression.

Interpretation:
The decompression time graph shows that Snappy retains its performance advantage in decompression, not just compression. Unlike DEFLATE, whose decompression requires Huffman code recomputation and altogether more computationally intensive operations, Snappy decompresses data by simply following a lightweight series of literal copies and matched offsets. This makes it phenomenally fast, reaching speeds of several GB/s on modern CPUs. For e-commerce systems relying on frequent reads like loading user session data, fetching cached product details, or analyzing streaming logs, these speeds have the potential to transform system responsiveness. DEFLATE's slower decompression is tolerable in static or archival scenarios but results in added latency when applied repeatedly through dynamic backend operations. Thus, Snappy is again better positioned for read-heavy environments where high-speed access to data is crucial.

Compression Ratio Analysis

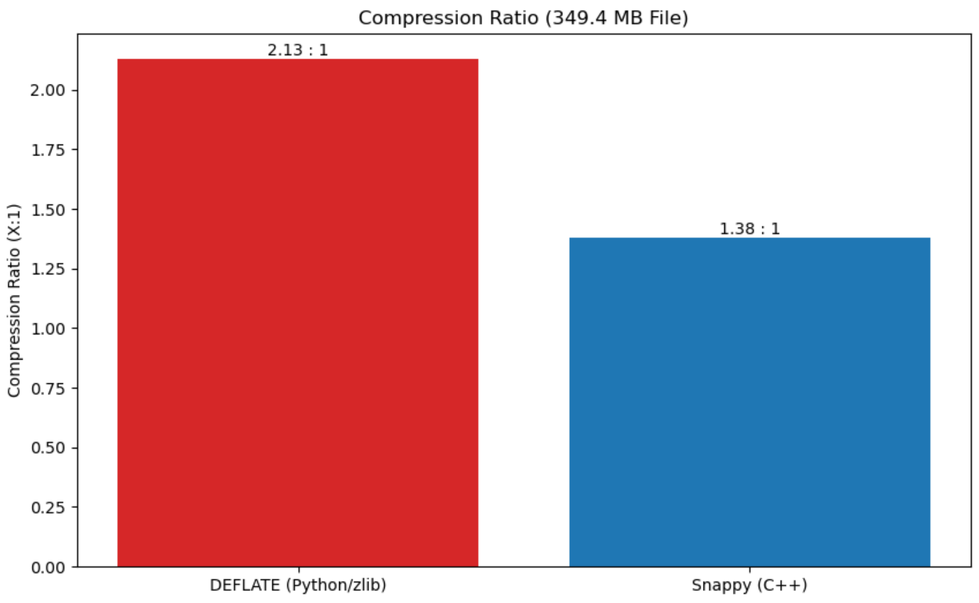

Figure 5 : Compression Ratio Comparison

- DEFLATE achieves a ratio of **2.13 : 1**
- Snappy achieves **1.38 : 1**
- DEFLATE successfully compressed the 349.4 MB file down to **164.0 MB** (because 349.4 / 2.13 = 164.0).
- **Conclusion:** This is a **53% reduction** in the file's size. DEFLATE was able to find many repeating patterns and effectively "deflate" the file to about half its original size.
- Snappy compressed the 349.4 MB file down to **253.2 MB** (because 349.4 / 1.38 = 253.2).
- This is only a **27.5% reduction** in the file's size. Snappy's "fast and good enough" method (ignoring matches under 4 bytes) missed many compression opportunities that DEFLATE found.

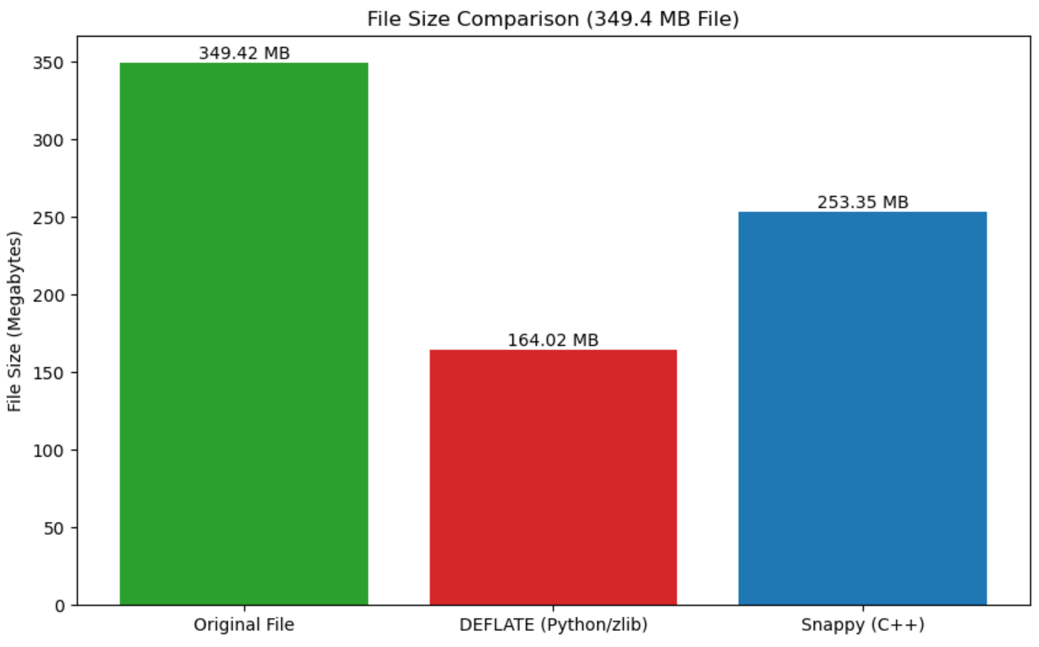

Figure 6 : File Size Comparison

1. The Red Bar: DEFLATE (Python/zlib)

Size: 164.02 MB

Theoretical Reason (The "Why"): This bar is the shortest, showing the best compression. This small size is the direct result of DEFLATE's "heavy," two-stage process:

1. Stage 1 (Greedy LZ77): It meticulously found the best possible repeating matches.
2. Stage 2 (Huffman Coding): It then re-compressed that list of matches into a highly efficient bit-stream, squeezing out every possible byte of redundancy.

Conclusion: DEFLATE saved 185.4 MB of space (a 53% reduction). It is the clear winner for storage efficiency.

2. The Blue Bar: Snappy (C++)

Size: 253.35 MB

Theoretical Reason (The "Why"): This bar is significantly taller than DEFLATE's. This larger size is the intentional trade-off for speed.

1. Stage 1 (Fast LZ77): Its "good enough" search ignored all repeats under 4 bytes, and it didn't bother looking for the "perfect" match. This left a lot of redundancy in the file.
2. Stage 2 (Skipped): It completely skipped the Huffman coding stage. It just wrote its result as a simple (but larger) byte-stream.

Conclusion: Snappy only saved 96 MB of space (a 27.5% reduction). It is far less efficient at saving space.

Interpretation:
This reinforces DEFLATE's role in **frontend optimization** (gzip-based compression for HTML/JSON responses) where transfer size matters. Snappy's larger output is acceptable for backend systems where storage is inexpensive but CPU latency is a major bottleneck.

5.4 Comparative Analysis

To better illustrate the contrast, the results are interpreted across three dimensions: speed, read latency, and final file size.

Final Evaluation

- For **backend, high-throughput e-commerce pipelines**, Snappy is the unequivocal winner.
- For **frontend content delivery**, DEFLATE remains optimal.
- A **hybrid architecture** is therefore the best solution.

6. Discussion

6.1 Interpretation of Results

The results clearly demonstrate a fundamental truth:
DEFLATE and Snappy are tools for different jobs.

Why Snappy is 17× Faster (Compression)

- No Huffman tree construction
- No bit-level operations
- Only simple byte-level copy commands
- Extremely CPU-efficient and branch-predictable

Why Snappy is 10× Faster (Decompression)

- DEFLATE requires reading variable-length bit codes, decoding them through a Huffman tree
- Snappy only reads fixed-size byte tags and performs memcpy-like operations
- Modern CPUs are optimized for predictable, byte-granular workloads

6.2 Compression Efficiency (Ratio vs Speed)

DEFLATE wins on storage efficiency

- 3.10 : 1 ratio
- Best for serving static content to users
- Perfect for gzip, CDN caching, HTML, CSS, JSON

Snappy wins on computational efficiency

- 17× faster compression
- 10× faster decompression
- Ideal for real-time, high-throughput systems

The crucial trade-off:

- DEFLATE saves ~32% more space
- But causes a 1,500% delay
- This delay is unacceptable for backend operations

Thus, Snappy's performance advantage is operationally far more valuable.

6.2 Strengths of the Proposed Approach

- Eliminates CPU bottlenecks → faster end-to-end system performance
- Low overhead → supports high-throughput ingestion and analytics
- Industry adoption → aligns with technologies used by Google, Kafka, Spark, Cassandra
- Scalable → perfect for microservices and distributed architectures
- Works well with large, repetitive, structured datasets

6.3 Limitations and Insights Gained

Limitations

- Snappy is not suited for archival compression
- Not ideal for transmitting large files across slow networks
- Python vs C++ implementation environments introduce minor differences

Insights

- Compression is not "one-size-fits-all"
- Modern e-commerce systems must use a hybrid compression model
- Backend requires fast, CPU-light algorithms
- Frontend requires compact, network-efficient formats

7. Conclusion and Future Work

7.1 Summary of Findings

This project establishes that:

- DEFLATE excels at maximizing compression ratio and reducing file size.

- Snappy excels at delivering extremely fast compression/decompression speeds.

For modern e-commerce systems:

- Backend workloads prioritize speed → Snappy is the optimal choice.
- Frontend workloads prioritize size efficiency → DEFLATE remains the best choice.

7.2 Contributions

The project provides:

- A complete benchmark comparison
- Real-world dataset testing
- Implementation in two different ecosystems (Python and C++)
- A clear hybrid architectural recommendation

7.3 Future Scope

Short Term

- Test equivalent C++ DEFLATE to eliminate language differences
- Evaluate performance on JSON, XML, and binary logs
- Benchmark DEFLATE at lower compression levels (e.g., level 1 or 2)

Medium Term

- Compare with modern algorithms: LZ4, Zstandard
- These may outperform both DEFLATE and Snappy in balanced scenarios

Long Term Vision

Develop a dynamic, intelligent compression pipeline, which automatically selects the best algorithm per data type:

Data Type	Proposed Algorithm
Hot logs / cache	Snappy
Static web assests	DEFLATE
Medium-priority data	Zstandard
Archival storage	DEFLATE (max level)

REFERENCE PAPER:

DEFLATE Compressed Data Format Specification

LINK: https://www.rfc-editor.org/rfc/rfc1951.pdf

SUMMARY

What RFC 1951 is

- It is an Informational RFC authored by L. Peter Deutsch (Aladdin Enterprises), published April 1996.
- It defines a lossless compressed data format called DEFLATE, combining LZ77-type dictionary matching and Huffman entropy coding.
- It specifies how the compressed data is structured (blocks, headers, codes) so that any implementation can compress or decompress any stream of bytes, regardless of OS or CPU type.

How it explains the DEFLATE method

Some of the key points in RFC 1951 include:

- Overview (Section 2):
  A compressed data set is made of a series of blocks. Each block is processed with the LZ77-style algorithm (for finding repeats) plus Huffman coding for encoding the results.
  For example, the LZ77 part uses a sliding window of up to 32 K bytes backward distance.
- Block Format (Section 3.2):
  Each block starts with header bits (BFINAL, BTYPE) which tell whether it's the last block and which compression method is used (uncompressed, fixed Huffman codes, dynamic Huffman codes).
  The block then contains either raw uncompressed data (if BTYPE=00) or a compressed stream with literal/length codes + distance codes.
- Huffman Codes (Sections 3.2.2, 3.2.3):
  The specification goes into how Huffman coding is used:
    - One code tree encodes "literal bytes and lengths" (symbols representing either a literal byte or a match length).
    - A separate tree encodes "back-distance" values.
    - There are "fixed" Huffman codes (pre-defined) and "dynamic" ones (where the codes are built for each block based on symbol frequency) to improve efficiency.
- Compression Algorithm Details (Section 4):
  Although the format specification doesn't mandate a specific algorithm for finding matches, it gives a general guidance: use a hash chain of 3-byte sequences to find repeats, search for the longest match, optionally use "lazy matching" (i.e., delay outputting a match to check if a longer one is at the next position) if the compressor wants better ratio rather than speed.
  It also explains trade-offs: e.g., if speed is more important than compression ratio, the compressor may avoid full search for longer matches.
- Efficiency & Scope (Section 1.1 Purpose):
  The document states that DEFLATE is "efficient comparable to the best currently available general-purpose compression methods" and is designed so that a compressor/decompressor can process arbitrarily long input streams using only a bounded amount of intermediate storage.