Amazon Music Reviews Analysis

Ali Najar (401102701)
Sadegh Mohammadian (401109477)
Mazdak Teymourian (401101495)
Maryam Shiran (400109446)

September 2025

1 Introduction

This project focuses on designing and implementing a complete data pipeline for the ingestion, processing, and analysis of the Amazon Music Reviews dataset.

The solution leverages OpenSearch, an open-source search and analytics platform, as the core technology for indexing, querying, and analyzing large volumes of review data. To support the workflow, the project includes scripts for:

- Data preprocessing: cleaning, transforming, and preparing raw data for ingestion.
- Bulk ingestion: efficiently loading processed data into OpenSearch.
- Querying and analysis: enabling fast retrieval and insights through OpenSearch's query capabilities.
- Performance benchmarking: evaluating system efficiency and response times under different workloads.

For portability and reproducibility, the entire environment is containerized using Docker, ensuring consistent setup across different systems and simplifying deployment.

Step-by-step run instructions appear in the project's GitHub README. In the next sections we'll quickly walk through each item above and answer the agenda questions.

2 Streaming Data Parser

2.1 Overview

The parser, implemented in `parse_stream.py`, processes Amazon music review datasets originally provided in a semi-structured text format. Each review appears as a block of key–value pairs separated by blank lines. However, the raw files contain numerous inconsistencies, including: Irregular spacing, Invisible characters, and Malformed key delimiters.

These inconsistencies make direct analysis unreliable. To address this, the parser employs a robust cleaning and normalization strategy.

2.2 Parsing and Cleaning Strategy

The parser ensures consistency and quality of the data through the following steps:

- **Whitespace normalization:** Excessive and irregular spacing is standardized.
- **Character cleaning:** Zero-width and stray characters are removed.
- **Error tolerance:** Records with malformed delimiters are parsed gracefully without halting execution.

2.3 Data Transformation

Each raw review block is converted into a structured JSONL (JSON Lines) record with standardized fields, including: productId, title, userId, score, summary, text.

In addition, derived fields are computed, such as:

- `helpfulness_up` and `helpfulness_down` (from the raw helpfulness ratio).

Numeric fields (e.g., price, score, timestamps) are cast into appropriate data types. Reviews that are incomplete or invalid (e.g., missing both `text` and `summary`, or containing extremely short content) are filtered out.

2.4 Streaming Architecture

The system is designed for scalability, processing reviews one at a time rather than loading entire files into memory. Output options include:

- **Standard output:** For integration with other command-line tools.
- **Kafka streaming:** For real-time pipelines.

Rate limiting can be applied to control emission speed, which is particularly useful when streaming to downstream systems with fixed throughput expectations.

3 Indexing and Search with OpenSearch

3.1 Technology Selection

Given the dataset's nature—free-form text, ratings, helpfulness ratios, and metadata—the primary requirement was large-scale indexing, aggregation, and full-text search. Traditional relational databases were unsuitable due to limited support for distributed text search.

After evaluating alternatives such as Solr and MongoDB, **OpenSearch** was chosen based on: JSON document storage, Strong full-text search capabilities, Horizontal scalability, Rich aggregation framework, and Compatibility with Elasticsearch APIs.

3.2 Features and Architecture of OpenSearch

OpenSearch organizes data in a cluster-based architecture where indices are composed of shards. Key features include:

- Inverted index for efficient keyword and phrase queries,
- Sharding and replication for scalability and fault tolerance,
- RESTful API for indexing, querying, and management,
- Rich aggregation framework for statistical analysis and token extraction,
- Ingest pipelines for preprocessing and document enrichment.

3.3 Transformation of Raw Data

The raw data, arriving either as JSONL files or Kafka messages, was first validated and then transformed into the **NDJSON format** required by the OpenSearch bulk API. Documents were batched both by count and by payload size to avoid oversized requests. Each record included text fields (review text, summary), numerical fields (score, helpfulness ratio, timestamp), and identifiers (product and user). Custom mappings were defined to enable tokenization, analyzers, and shingle subfields for phrase analysis.

3.4 Handling Streaming Data

Two ingestion modes were supported:

1. **File-based bulk ingestion** for one-time historical backfills.
2. **Kafka-based ingestion** for continuous, real-time data:
   - Consumer groups ensured parallel consumption.
   - Offset management handled fault tolerance.
   - Batching was applied before sending data to OpenSearch.

Both ingestion paths included retry logic with exponential backoff to handle transient network or server errors without data loss.

4 Bulk ingestion

The `bulk_ingest.py` module implements a production-ready, streaming bulk indexer designed to move JSON documents into OpenSearch efficiently and reliably. It supports ingestion from both newline-delimited JSON files (JSONL) and Kafka topics, making it suitable for batch backfills as well as continuous real-time data pipelines.

5 Input Handling

5.1 JSONL Source

- Reads input as a text stream with support for gzip compression and byte-order marks.
- Skips blank lines; malformed JSON entries are logged and ignored without terminating execution.

5.2 Kafka Source

- Uses a consumer group to fetch messages.
- Messages are deserialized into Python objects; only dictionary payloads are processed.

This tolerant input strategy ensures robustness across varied and imperfect data sources.

6 Batching and Buffering

- Implements an NDJSON buffer that generates the exact payload format required by the OpenSearch `_bulk` API.
- Enforces strict thresholds on document count and payload size.
- Automatically flushes when thresholds are reached, preventing oversized HTTP requests.
- Detects pathological cases (e.g., single documents exceeding limits) and surfaces clear error messages for operator action.

7 Transport and Error Handling

- Relies on an `httpx`-based client wrapper with connection pooling, keep-alive, and configurable timeouts.
- Bulk payloads are posted with the correct NDJSON content type.
- Responses are parsed for both HTTP-level and per-item errors.
- Partial failures are logged as warnings.
- Client connections are cleanly closed on shutdown to release sockets.

8 Operational Controls

The tool exposes comprehensive command-line options, including:

- Data source selection (JSONL or Kafka).
- OpenSearch host, target index, and optional ingest pipeline.
- Performance tuning parameters (batch size, maximum payload bytes, timeouts, pool sizes).
- Kafka-specific options (consumer group, brokers, offset reset policy).

The process also emits runtime metrics such as document counts, byte throughput, and batch statistics. A final flush is performed on completion or keyboard interrupt to ensure no data loss.

9 Data Cleaning Challenges

During the ingestion process, several data quality issues were encountered and addressed:

- **Malformed JSON lines:** Incomplete or corrupted records were skipped, with errors logged for review.
- **Missing fields:** Records without summaries or helpfulness ratios were supplemented with defaults (e.g., empty strings, nulls).
- **Encoding anomalies:** Byte order marks (BOM) and stray characters were normalized during stream reading.
- **Stopwords and trivial tokens:** Common words (e.g., *i, you, song, music*) distorted aggregation results and were explicitly excluded in query definitions.

10 Querying and analysis

10.1 Querying Overview

The `run_queries.py` script is designed as a query execution and reporting tool for an OpenSearch (or Elasticsearch-compatible) index of Amazon Music reviews. Its primary purpose is to run a predefined set of analytical queries (q1–q11), summarize the results, and export them in structured formats. The script does not perform ingestion or transformation tasks but provides an interface between the search engine and tabular or human-readable reports.

10.2 Workflow

- **Configuration:** Set via command line (OpenSearch host, target index, queries to execute, and output directory).
- **Execution:** Supports running a single query or all queries sequentially. Each query generates a JSON request body tailored to specific analysis tasks such as keyword retrieval, aggregation, grouping, or helpfulness ratio analysis.
- **Performance:** Utilizes sampler aggregations to manage shard-level load and maintain scalability.
- **Request Handling:** Sends requests to the `_search` endpoint with retry logic and exponential backoff.
- **Output:**
  - Raw JSON responses saved for debugging.
  - Processed summaries exported as CSV files for analytical use.
  - Aggregation results include top terms or buckets.
  - Hit-based queries include structured review details (IDs, scores, helpfulness ratios, timestamps, truncated summaries).
  - Each run concludes with total hit counts for dataset context.

10.3 Query Optimization

To balance accuracy and performance, the following methods were applied:

- Match phrase queries for precise multi-word matching.
- Range and term filters to restrict results (e.g., `score > 4`, `helpfulness_ratio > 0.75`).
- Sampler aggregations to reduce shard processing overhead.
- Exclusion patterns to remove trivial stopwords.
- `track_total_hits = false` to reduce overhead on non-exhaustive queries.
- Warm-up requests in benchmarking runs to stabilize caches and connections.

11 Benchmarking Analysis Report

The benchmarking tool (`bench.py`) provides an asynchronous performance evaluation framework for OpenSearch. Its main objectives are to measure query throughput, latency characteristics, and error resilience under different concurrency levels.

The tool is implemented using the `httpx` asynchronous client, leveraging persistent keep-alive connections and configurable connection pooling to approximate realistic client workloads. It supports both a built-in default query—targeting reviews containing "lyrics" and "vocals" with a minimum score filter—and custom queries supplied from JSON files.

Before timed runs, a series of warm-up requests are executed sequentially. This step stabilizes the connection pool and pre-populates OpenSearch caches, minimizing cold-start effects. Benchmark runs are then carried out for fixed durations at specified concurrency levels. During execution, asynchronous worker tasks continuously issue queries, record request latencies, track successful responses, and count errors. Failures are logged but do not interrupt the overall run.

At completion, the benchmark produces detailed performance metrics:

- Successful and failed request counts
- Achieved throughput (queries per second)
- Average latency, standard deviation, and maximum latency
- Percentile distributions (p50, p90, p95, p99)

Results can be exported in JSON for programmatic analysis or CSV for spreadsheet-based reporting. CSV output is organized by concurrency level, enabling clear comparisons across different workload intensities.

12 Query Benchmark Analysis

We benchmarked a total of eleven representative queries, with execution scripts and results published on GitHub. Figure 1 illustrate the first four queries, comparing baseline execution plans against their optimized counterparts.

Overall, query optimization improves performance at low to moderate concurrency. Throughput gains of roughly 20–25% were observed in this range, and average latency was consistently lower with smoother growth curves. These benefits suggest that optimized queries make more efficient use of system resources under typical workloads.

In summary, optimization is most effective for improving throughput and responsiveness under realistic concurrency, but it does not fully address scaling challenges or long-tail latency.

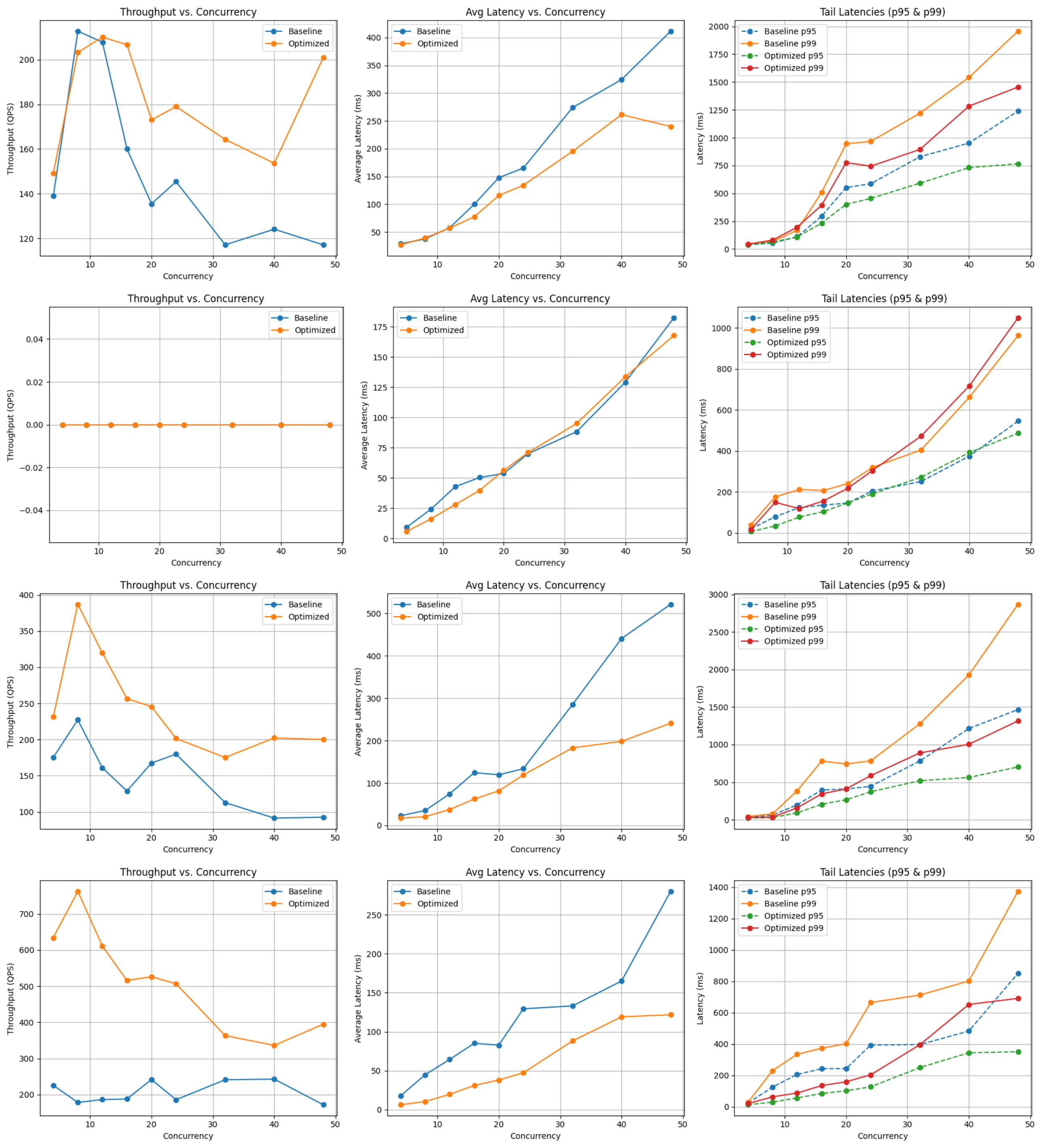

Figure 1: Throughput and latency benchmarks for the first four queries. Optimized queries improve performance at low-to-moderate concurrency but face scalability limits at higher loads. All eleven queries are available on GitHub.