Inverted Index Datamart Structures

Course: Big Data

Academic Year: 2025/2026

Group Name: Microsoft 2

Student Name	Student ID
Enrique Padrón Hernández	45392592E
Jorge Cubero Toribio	44748581B
Sergio Muela Santos	45333475S
Daniel Medina Gonzalez	45619169A

GitHub Repository:
https://github.com/EnriquePDabird/Microsoft-2/tree/main/stage1

Abstract

This project implements an automated data processing pipeline that extracts metadata and text from the Project Gutenberg API, organizes it into a structured datalake, and creates an inverted index in a datamart for efficient search and analysis.

1 Introduction

The goal of this project is to build a scalable pipeline for collecting, storing, and indexing free literary works from the Project Gutenberg API. The system integrates three core layers — a datalake for raw data storage, a datamart for structured access, and a control layer for tracking process status — forming the foundation for future text analytics and search applications.

2 System Architecture

The system follows a modular **three-layer architecture**, designed to ensure data organization, traceability, and scalability. The main layers are the **Datalake**, the **Datamart**, and the **Control Layer**. Each one is implemented through a specific set of Python functions, directories, and file formats.

2.1 Overview of the Pipeline

The entire pipeline is coordinated by the function `controller()`, which manages the data flow from extraction to indexing. The general workflow is as follows:

1. The `main()` function connects to the Project Gutenberg API and retrieves a list of books.
2. The `controller()` function orchestrates the process, calling:
   - `extract_fetch_and_store_books()` to download and store data in the Datalake.
   - `datamart_fill()` to process and index the stored books into the Datamart.
3. The **Control Layer** ensures no book is downloaded or indexed more than once.

Each execution of the pipeline processes one page of books from the API and continues recursively until 5 pages are completed.

2.2 Datalake Layer

The **Datalake** serves as the main repository for raw and semi-processed data. It is organized hierarchically by date and hour, ensuring versioning and chronological structure. The directories are created dynamically using:

```
now = datetime.now()
date_dir = now.strftime('%Y-%m-%d')
time_dir = now.strftime('%H')
full_path = os.path.join('datalake', date_dir, time_dir)
os.makedirs(full_path, exist_ok=True)
```

Each downloaded book produces two files:

- `book_id.header.txt` – contains structured metadata such as title, authors, subjects, and bookshelves.
- `book_id.body.txt` – stores the cleaned text of the book obtained from the API endpoint:

  `https://project-gutenberg-free-books-api1.p.rapidapi.com/books/{book_id}/text`

This organization allows the pipeline to handle multiple executions safely, as each run stores results in a new timestamped directory.

2.3 Datamart Layer

The **Datamart** provides a structured and queryable version of the data for analysis and retrieval. It is implemented inside the `datamart/` directory and includes two main files:

1. `metadata.db` – an SQLite database created by the function `datamart_fill()` that stores book metadata in a relational format:

```
CREATE TABLE IF NOT EXISTS books (
    id INTEGER PRIMARY KEY,
    title TEXT,
    authors TEXT,
    subjects TEXT,
    bookshelves TEXT
);
```

2. `inverted_index.json` – a JSON file containing an inverted index structure, mapping each word to the book IDs where it appears and the number of occurrences:

```
{
    "adventure": {"1234": 5, "5678": 3},
    "love": {"9876": 12, "2345": 8}
}
```

This index is generated using the function `inverted_index_creation()`, which tokenizes the book texts, removes stop words using NLTK, and counts word frequencies with a nested dictionary:

```
inverted_index = defaultdict(lambda: defaultdict(int))
```

2.4 Control Layer

The **Control Layer** ensures that the pipeline maintains state and avoids redundant operations. It creates and manages a directory named `control/` with the following files:

- `downloaded_books.txt` – stores the IDs of books that have been successfully downloaded.
- `indexed_books.txt` – stores the IDs of books that have been processed into the Datamart.

Before downloading or indexing, the system checks whether a book ID already exists in these files:

```
if book_id in downloaded_books and book_id in indexed_books:
    continue
```

This mechanism allows for incremental and restartable processing — if the script is interrupted, it can resume safely without duplicating work.

2.5 Architecture Summary

- **Datalake:** Raw and semi-processed book data, organized by date/time.
- **Datamart:** Structured database (metadata) and search index (inverted index).
- **Control Layer:** State management for process continuity and deduplication.

Together, these components form a scalable ETL (Extract, Transform, Load) pipeline capable of continuously ingesting and indexing large volumes of textual data from the Project Gutenberg API.

3 Design Decisions

3.1 Choice of Data Structures

The pipeline leverages Python's built-in and standard library data structures, selected for their simplicity and efficiency in handling semi-structured data.

3.1.1 JSON for Data Exchange

The Project Gutenberg API responses are in JSON format, which allows direct deserialization and flexible handling in Python:

```
books = json.loads(data)
```

Each book's metadata is stored as a JSON object in files named `book_id.header.txt`. This ensures interoperability between the `datalake` and `datamart` layers and preserves the hierarchical structure of the metadata fields (e.g., authors, subjects, bookshelves).

3.1.2 Dictionaries for Metadata Extraction

Metadata extraction is implemented using dictionaries and list comprehensions for clarity and speed. Example from the `extract_book_info()` function:

```
book_info.append({
    'id': book_id,
    'title': title,
    'authors': [author.get('name') for author in book.get('authors', [])],
    'subjects': book.get('subjects', []),
    'bookshelves': book.get('bookshelves', [])
})
```

This design enables easy extension of the schema if new fields are added to the API in the future.

3.1.3 Default Dictionaries for the Inverted Index

The inverted index is constructed using nested `defaultdict` structures to efficiently count word frequencies:

```
inverted_index = defaultdict(lambda: defaultdict(int))
```

This approach eliminates the need to check for key existence manually and supports constant-time insertion and updates.

3.1.4 Sets for Control Files

Downloaded and indexed book IDs are stored as Python `set()` objects for fast membership testing:

```
if os.path.exists(downloaded_books_path):
    with open(downloaded_books_path, 'r', encoding='utf-8') as f:
        downloaded_books = set(line.strip() for line in f if line.strip())
```

This allows efficient comparison of new and previously processed books, ensuring no duplication.

3.2 Indexing Strategy

The inverted index is the core component of the Datamart. It is designed to support text retrieval by mapping each word to all the books in which it appears, along with its frequency.

3.2.1 Text Cleaning and Tokenization

The text of each book is preprocessed before indexing. The cleaning process includes lowercasing, punctuation removal, and stopword filtering:

```
for word in text.split():
    word = word.lower().strip('.,!?;"()[]{}')
    if word and word not in stop_words:
        inverted_index[word][book_id] += 1
```

Stopwords are provided by the NLTK English stopword corpus:

```
from nltk.corpus import stopwords
stop_words = set(stopwords.words('english'))
```

3.2.2 Storage Format

The resulting inverted index is saved as a JSON file:

```
with open('datamart/inverted_index.json', 'w', encoding='utf-8') as index_file:
    json.dump(inverted_index, index_file, ensure_ascii=False, indent=4)
```

The JSON format provides human readability and compatibility with other tools for later analysis or visualization.

3.3 Control and Fault Tolerance Mechanisms

The control mechanism ensures that the pipeline can resume from the last processed state without losing progress. The design choice of using plain text files for control (`downloaded_books.txt` and `indexed_books.txt`) was made for transparency and simplicity.

Before downloading or indexing a book, the controller checks for its presence in these files:

```
if book_id in downloaded_books and book_id in indexed_books:
    continue
```

This enables incremental updates and fault tolerance — if execution is interrupted, the next run continues processing only pending books.

3.4 Database Design in the Datamart

For structured storage of metadata, a lightweight SQLite database is used. This choice is motivated by:

- No need for a separate server or dependencies.
- ACID compliance for reliable transactions.
- Simple integration with Python's `sqlite3` library.

The schema design is straightforward and matches the metadata fields extracted from the API:

```
CREATE TABLE IF NOT EXISTS books (
    id INTEGER PRIMARY KEY,
    title TEXT,
    authors TEXT,
    subjects TEXT,
    bookshelves TEXT
);
```

Each record corresponds to one book, and authors, subjects, and bookshelves are stored as comma-separated strings to maintain a normalized yet flexible structure.

4 Performance Evaluation

4.1 Benchmarking Methodology

To evaluate the performance of the pipeline, a benchmarking script was developed to measure execution time, memory usage, and CPU utilization during the execution of the `main()` function. The script leverages the `psutil` and `time` libraries to obtain precise system metrics before and after the execution:

```
import time
import psutil
import os
import main

def benchmark():
    start_time = time.time()
    process = psutil.Process(os.getpid())

    mem_before = process.memory_info().rss / (1024 * 1024)
    cpu_before = process.cpu_times()

    main.main()  # Run the pipeline

    mem_after = process.memory_info().rss / (1024 * 1024)
    cpu_after = process.cpu_times()
    end_time = time.time()

    print(f"Execution time: {end_time - start_time:.2f} s")
    print(f"Memory difference: {mem_after - mem_before:.2f} MB")
```

This function captures both user and system CPU times to distinguish between computation and input/output operations. The benchmark was executed on a standard workstation running the full data ingestion and indexing pipeline.

4.2 Results and Analysis

Metric	**Value**
Execution Time	813.22 seconds
Memory Before Execution	219.95 MB
Memory After Execution	219.65 MB
Memory Difference	-0.25 MB
User CPU Time	58.56 seconds
System CPU Time	7.69 seconds
Total CPU Time	66.25 seconds

Table 1: Benchmark results for the full pipeline execution.

The results indicate that the pipeline is **I/O-bound** rather than CPU-bound. While the total wall-clock time was approximately 813 seconds (13.5 minutes), the combined

CPU time was only 66 seconds. This suggests that most of the execution time was spent waiting for network responses from the Project Gutenberg API and performing file I/O operations (reading/writing text files and JSON structures).

Memory consumption remained stable throughout execution, showing a slight reduction of 0.25 MB after completion. This indicates efficient memory management and confirms that the system does not accumulate unnecessary objects in memory during recursive or batch operations.

4.3 Interpretation and Optimization Opportunities

The performance metrics suggest the following optimization directions:

- **Parallelization:** The long execution time could be reduced by implementing concurrent data fetching (e.g., using `asyncio` or `concurrent.futures`) to download and process multiple books simultaneously.
- **Disk I/O Optimization:** Since I/O operations dominate the runtime, compressing or batching file writes in the Datalake and Datamart could reduce overall latency.
- **Caching:** API responses for metadata could be cached locally to avoid redundant requests during repeated runs.

Overall, the benchmark confirms that the pipeline is functionally efficient in terms of resource usage, with modest CPU and memory requirements, and that future performance gains will primarily depend on improving data retrieval parallelism and file handling strategies.

5 Conclusion and Future Improvements

The implemented pipeline successfully automates the extraction, storage, and indexing of literary works from the Project Gutenberg API. By adopting a modular three-layer architecture — comprising the Datalake, Datamart, and Control Layer — the system achieves a robust structure that ensures scalability, traceability, and fault tolerance. Each component plays a distinct role: the Datalake organizes raw data chronologically, the Datamart enables structured querying and analysis, and the Control Layer prevents redundancy while maintaining execution continuity.

Performance benchmarking confirmed the system's efficiency in resource management. Memory consumption remained stable throughout execution, and CPU usage was moderate, indicating effective handling of large text datasets. However, the total execution time revealed that the pipeline is predominantly limited by input/output operations and network latency, rather than computational complexity. This outcome aligns with the design focus on sequential data retrieval and file-based processing.

Despite meeting its functional goals, the project presents opportunities for further enhancement. Implementing asynchronous or parallel processing would significantly reduce execution time by enabling concurrent data downloads and processing tasks. Integrating a scheduling mechanism could automate periodic updates to the dataset, ensuring that the pipeline operates continuously without manual intervention. Likewise, migrating

from SQLite to a more scalable database engine, such as PostgreSQL or Elasticsearch, would allow faster querying and more advanced search functionalities.

In conclusion, the project establishes a solid foundation for large-scale text ingestion and indexing. Its modular and transparent architecture provides an excellent platform for expansion into more advanced big data analytics applications, including natural language processing and machine learning. With targeted optimizations, the system can evolve into a scalable, production-ready framework for digital text management and analysis.