Stage 1
Building the Data Layer

Search Engine Project

Big Data
Grado en Ciencia e Ingeniería de Datos
Universidad de Las Palmas de Gran Canaria

1 Introduction

The overall goal of Big Data course project is to design and implement a simple **search engine** from scratch, providing students with a practical introduction to the core components of modern data pipelines and Big Data processing. The final system will be capable of ingesting raw data, processing it into efficient structures, and serving basic search queries. At the core of this project lies the **data layer**, which serves as the foundation for all other components of the search engine. Its main purpose is to provide a reliable, scalable, and well-organized environment where raw data can be stored, transformed, and optimized for efficient querying.

This guide focuses exclusively on **Stage 1**, whose objective is to design and implement this data layer, establishing the backbone of the entire search engine. In this phase, students will build a workflow to collect, clean, and organize data, ensuring it is ready for the indexing and querying processes in later stages. The outcome of Stage 1 is a well-structured storage architecture that separates unstructured content from structured, queryable data. The data layer consists of two complementary storage systems:

- **Datalake:** Stores the raw and cleaned book content in an unstructured format, acting as the central repository for data ingestion and processing.
- **Datamarts:** Stores structured, queryable data, including book metadata and the inverted indexes that enable fast and efficient search operations.

By the end of this stage, students will have developed a solid data architecture that mirrors the design principles of real-world Big Data systems. This foundation will enable **Stage 2**, where the crawling, indexing and querying modules will be implemented and integrated into a fully functional search engine using a **microservice architecture in Java**.

2 Data Source

The dataset for this project comes from **Project Gutenberg**, a free digital library of public domain books.

Each book has a unique id and is available as a plain-text (`.txt`). For example, the book with ID `1342` (Pride and Prejudice by Jane Austen) can be downloaded from:

```
https://www.gutenberg.org/cache/epub/1342/pg1342.txt
```

Each book follows a consistent structure:

1. **Header:** Contains metadata about the book such as title, author, release date, language, and licensing notes.
2. **Body:** The actual text of the book.
3. **Footer:** Closing notes and copyright information.

The following script provides an example of how to download a book from Project Gutenberg, detect the markers that delimit the text, and split it into header and body.

```python
import requests
from pathlib import Path

START_MARKER = "*** START OF THE PROJECT GUTENBERG EBOOK"
END_MARKER = "*** END OF THE PROJECT GUTENBERG EBOOK"

def download_book(book_id: int, output_path: str):
    output_path = Path(output_path)
    output_path.mkdir(parents=True, exist_ok=True)

    url = f"https://www.gutenberg.org/cache/epub/{book_id}/pg{book_id}.txt"

    response = requests.get(url)
    response.raise_for_status()
    text = response.text

    if START_MARKER not in text or END_MARKER not in text:
        return False

    header, body_and_footer = text.split(START_MARKER, 1)
    body, footer = body_and_footer.split(END_MARKER, 1)

    # --- Step 3: Save each part ---
    body_path = output_path / f"{book_id}_body.txt"
    header_path = output_path / f"{book_id}_header.txt"

    with open(body_path, "w", encoding="utf-8") as f:
        f.write(body.strip())

    with open(header_path, "w", encoding="utf-8") as f:
        f.write(header.strip())

    return True

# Example of Usage
success = download_book(1342, "data/output")
```

Listing 1: Function to download and split a book

3 Organizing the Datalake

To ensure the ingestion pipeline remains scalable, organized, and easy to maintain, the **datalake** should follow a well-defined directory hierarchy. A recommended approach is to structure files based on the date and hour of ingestion, reflecting exactly when each file was downloaded.

This organization offers several key benefits:

- **Traceability:** Quickly identify when a specific book was ingested, simplifying debugging and audit processes.
- **Incremental processing:** Later stages, such as indexing, can focus only on the most recent folders instead of scanning the entire datalake.
- **Scalability:** Distributing files across multiple directories avoids performance bottlenecks that occur when too many files are stored in a single folder.
- **Compatibility:** This layout mirrors common practices in distributed storage systems such as HDFS or Amazon S3, making it easier to migrate to a large-scale environment in the future.

By following this hierarchy, the datalake remains clean, efficient, and ready to handle growing volumes of data without requiring major structural changes. The recommended layout is:

```
datalake/
    YYYYMMDD/
        HH/
            <BOOK_ID>.body.txt
            <BOOK_ID>.header.txt
```

where *YYYYMMDD* is the date of download (e.g., `20250925`); *HH* is the hour of download in 24-hour format (e.g., `14` for 2 PM); and *<BOOK_ID>* is the file id obtained from Project Gutenberg.

4 Building Datamarts

Once books have been downloaded and processed, the next step is to organize the cleaned data into optimized storage structures called **datamarts**. While the *datalake* serves as a scalable repository for all raw and cleaned text, the *datamart* provides structured, queryable data that will be directly used by the indexing and search modules.
The datamart consists of two main components:

1. **Metadata:** Structured information about each book, extracted from the *header*.
2. **Inverted Index:** Efficient data structures built from the *body*, enabling fast search queries.

This separation ensures that metadata operations (e.g., filtering by author or title) and search operations (e.g., keyword lookups) can be handled independently and efficiently.

4.1 Metadata

The metadata must be parsed directly from the *header* of each downloaded book. By keeping metadata separate from the full text files, the system can quickly filter and organize books without scanning the entire contents of each file. This enables operations such as:

- Filtering books by a specific author, title, or language.
- Quickly locating the path of the corresponding cleaned text file.
- Supporting the indexing stage by supplying structured input data.

The header typically contains descriptive information in a structured, line-by-line format, including fields such as `Title`, `Author`, and `Language`. After extracting these values — for example, using *regular expressions* (regex) to match the relevant lines — the data can be cleaned and normalized. Once extracted, the metadata is stored in a database, such as **SQLite**, to provide a simple and efficient way to query structured information.
An example schema for the `books` table:

```
book_id | title                        | author       | language
5       | Robinson Crusoe              | Daniel Defoe | en
```

Database Engine Comparison

To evaluate the performance of the metadata layer, the same schema should be implemented using different database backends. Recommended categories for testing:

- **Lightweight embedded database: SQLite** is simple to set up and ideal for prototypes or small datasets.
- **Traditional relational database: PostgreSQL** or **MySQL** provide robust indexing, transaction handling, and better support for concurrent access.
- **NoSQL database: MongoDB** or **Redis** offer flexible schema design, high-speed inserts, and horizontal scalability for very large datasets.

By running the same set of operations on these different systems, students can measure the trade-offs between simplicity, speed, and scalability, and justify their final choice of database engine for the project.

Benchmarking Considerations

To ensure the metadata storage layer is efficient and scalable, several experiments should be planned:

1. **Insertion speed:** Measure how quickly metadata for thousands of books can be inserted into the database.
2. **Query performance:** Evaluate the response time for common queries, such as: *Find all books by a specific author*; or *Retrieve the path of a book by its title or ID.*

3. **Scalability tests:** Test how performance changes as the number of books grows from hundreds to tens of thousands.

These benchmarks will help select the most appropriate database engine for later stages of the project, ensuring that the metadata layer can support both the indexing and querying modules efficiently.

4.2 Inverted index

The inverted index can be organized in different ways depending on performance, scalability, and implementation complexity. Below are three illustrative approaches, each with its own advantages and trade-offs. Students are required to implement at least three different strategies, but they are also encouraged to design and experiment with their own custom approaches if they wish. The goal is to benchmark these implementations to understand how the physical organization of the index impacts both query speed and indexing efficiency..

Single Monolithic File

All terms and their posting lists are stored in **one single file**, for example, in a JSON or binary format. The structure maps each term directly to the list of documents where it appears:

```
{
    "adventure": [5, 12, 42],
    "island": [5, 1342],
    "shipwreck": [12, 17]
}
```

This file should be saved in the project under the path:

```
datamarts/inverted_index.json
```

Pros:

- Very simple to implement and maintain.
- Easy to back up and transfer as a single file.

Cons:

- As the index grows, every update may require rewriting a large file.
- Limited scalability for concurrent read/write operations.

NoSQL Database (MongoDB)

In this approach, the inverted index is stored directly in a **NoSQL database**, such as MongoDB, where each term is represented as a document containing its postings list. This design takes advantage of built-in features like indexing, compression, and fast random access.

```
{
    "term": "adventure",
    "postings": [5, 12, 42, 1342]
}
```

Pros:

- High performance for random reads and writes.
- Built-in scalability and support for distributed storage.
- Simplifies concurrent access and incremental updates.
- Query language allows for flexible searches and analytics.

Cons:

- Requires installing and managing an external database system.
- Added complexity compared to simple file-based approaches.
- Network latency can impact performance if not properly configured.

Hierarchical Folder Structure

In this approach, the inverted index is organized as a set of folders, where each **term** has its own individual `.txt` file. The files can be grouped into subdirectories. For example:

```
datamarts/
    inverted_index/
        A/
            adventure.txt
            apple.txt
        B/
            boat.txt
            bridge.txt
        C/
            castle.txt
```

Each file contains the **postings list** for its corresponding term, i.e., the list of `book_id` values (and optionally positions) where the term appears.
Example content of `adventure.txt`:

```
5
12
42
1342
```

Pros:

- Very fine-grained updates: only the file of the affected term is modified.
- Natural fit for distributed file systems or cloud storage environments.
- Easy to track and debug individual terms by inspecting their dedicated files.

Cons:

- A very large number of small files can overwhelm the filesystem, reducing performance.
- Lookup times may increase due to frequent file access operations.
- More complex maintenance compared to monolithic or grouped-file structures.

Benchmarking Considerations

To evaluate the different approaches, students should design experiments to measure:

- **Indexing speed:** Time required to build the inverted index from a given dataset.
- **Query performance:** Average response time for a set of representative search queries.
- **Scalability:** How performance changes as the number of books and terms grows.

These benchmarks will highlight the trade-offs between simplicity, update performance, and scalability, helping guide the selection of the most appropriate design for the final search engine.

5 Control Layer

The **control layer** acts as the *brain* of the data ingestion pipeline. Its main role is to coordinate and supervise the execution of tasks across the different stages of the system, ensuring that the workflow proceeds smoothly, without duplication or data loss. This layer focuses on **tracking the state of the system**. It keeps a record of what has been done, what is currently in progress, and what still needs to be completed. This makes the entire pipeline reliable and auditable.
In this stage, only a **minimal control layer** will be implemented. Its purpose is to test the functionality of the *data layer* by coordinating simple operations such as downloading a book and sending it to the indexing step. More advanced orchestration features, such as parallel processing or complex scheduling, will be addressed in later stages.

5.1 State Tracking

The control layer uses simple **control files** to track progress. Control files are stored in a dedicated directory, such as `control/`.

```
control/
    downloaded_books.txt
    indexed_books.txt
```

Each file contains a list of identifiers (`BOOK_ID`) representing the books at a specific stage:

- `downloaded_books.txt` Lists all books that have been successfully downloaded.
- `indexed_books.txt` Lists all books that have been successfully indexed.

5.2 How it works

The **control layer** acts as the central coordinator of the entire pipeline, deciding which actions need to be performed at each step. The orchestration logic follows a simple decision process:

1. **Check for books ready to be indexed:** If there are books that have already been downloaded and cleaned but not yet indexed, the control layer schedules them for indexing.

2. **Download new books if needed:** If no books are pending for indexing, the system attempts to download a new book from Project Gutenberg. Before downloading, it verifies that the selected `BOOK_ID` has not been downloaded before, preventing duplicates.

3. **Update state tracking:** After each operation, the corresponding control files are updated to reflect the current state of the pipeline.

The following function is a **simple example** of how the control layer can coordinate downloads and indexing tasks. It shows the basic idea of checking which books are ready to be indexed and downloading new ones when needed. Students are free to design their own orchestration logic, for instance by introducing multiple downloaders or parallel indexers, as long as it correctly manages the pipeline and avoids duplicated work.

```python
from pathlib import Path
import random

CONTROL_PATH = Path("control")
DOWNLOADS = CONTROL_PATH / "downloaded_books.txt"
INDEXINGS = CONTROL_PATH / "indexed_books.txt"

TOTAL_BOOKS = 70000

def control_pipeline_step():
    CONTROL_PATH.mkdir(parents=True, exist_ok=True)

    downloaded = set(DOWNLOADS.read_text().splitlines()) if DOWNLOADS.exists() else set()
    indexed = set(INDEXINGS.read_text().splitlines()) if INDEXINGS.exists() else set()

    ready_to_index = downloaded - indexed

    if ready_to_index:
        book_id = ready_to_index.pop()
        print(f"[CONTROL] Scheduling book {book_id} for indexing...")
        # Here you would call the indexer
        with open(INDEXINGS, "a", encoding="utf-8") as f:
            f.write(f"{book_id}\n")
        print(f"[CONTROL] Book {book_id} successfully indexed.")
    else:
        for _ in range(10):  # Retry up to 10 times to find a new book
            candidate_id = str(random.randint(1, TOTAL_BOOKS))
            if candidate_id not in downloaded:
                print(f"[CONTROL] Downloading new book with ID {candidate_id}...")
                # Here you would call the downloader
                with open(DOWNLOADS, "a", encoding="utf-8") as f:
                    f.write(f"{candidate_id}\n")
                print(f"[CONTROL] Book {candidate_id} successfully downloaded.")
                break
```

Listing 2: Example control function to coordinate downloads and indexing

6 Project Delivery Guidelines

Each group must select a unique **group name** that will be used to identify their project. The final deliverable for this stage consists of a single **written report** in PDF format. This document will serve as the official record of your work and must clearly reference the external repository where all the source code is hosted.
This is the only file that must be submitted to the **virtual campus platform**, and **only one member of the group** should upload it on behalf of the entire team. This avoids duplicate submissions and ensures clarity during the evaluation process.

6.1 Required Structure of the Report

The report (`.pdf`) must include the following sections:

1. **Cover page:**
   - Course name and academic year.
   - Project title.
   - Full names and student IDs of all group members.
   - The chosen group name.
   - The URL of the GitHub repository.
2. **Introduction and objectives:** Describe the purpose of the project and the scope of the first stage.
3. **System architecture:** Explain how the pipeline is organized, including the datalake, datamart, and control layer.
4. **Design decisions:** Justify the selected data structures and indexing strategies.
5. **Benchmarks and results:** Present performance metrics and discuss the outcomes.
6. **Conclusions and future improvements.**

6.2 Group Name and Repository Structure

The GitHub repository must follow the exact format:

```
https://github.com/<group_name>/stage_1
```

Example of a valid repository URL:

```
https://github.com/dataexplorers/stage_1
```

The repository must:

- Include a `README.md` with detailed setup and execution instructions.
- Provide a sample dataset so instructors can quickly test the pipeline.
- Use Git history to show the progression of the development work.

7 Evaluation Criteria

The grading will consider both the report and the implementation in the GitHub repository:

- **Report quality** (30%) – clarity, completeness, and organization of the written document.
- **Correctness of the pipeline** (30%) – proper functioning of downloading, indexing, and querying modules.
- **Code quality** (20%) – structure, modularity, and documentation of the source code.
- **Benchmarking and analysis** (20%) – depth of performance experiments and critical discussion of results.

Students will have the opportunity to deliver a short **oral presentation** of their work. This presentation is **optional** and provides an opportunity to gain extra credit or to better demonstrate individual contributions to the project.

- Each student will have a maximum of **4 minutes** to present the work, and will be assigned a specific time slot by the instructor.
- Presentations should focus on key aspects such as design decisions, implementation challenges, performance analysis, and lessons learned.
- Students that choose not to present will not be penalized; their grade will be based entirely on the report and implementation.

The schedule for the presentations will be communicated in advance so that each student knows exactly when they will be presenting.