Stage 1 — Search Engine Project

GCID – Big Data (2025/2026)
Group: BD Power
Members: Jaime Rivero Santana · Gabriel Munteanu Cabrera · Andrea Dumpiérrez Medina · Agustín Darío Casebonne
Repository: https://github.com/BD-Power/stage_1

---

1. Introduction and Objectives

In this first stage, we built the data layer of a text search engine. The system downloads books from Project Gutenberg, stores them in a well-organized data lake, and generates a data mart with inverted indexes to enable word-based searches.
The objectives were: (i) automate the download and control of books, (ii) compare two data lake strategies (by date/time and by author's initial), (iii) compare two storage formats for the inverted index (JSON vs TXT), and (iv) measure performance and discuss trade-offs.

---

2. System Architecture

General Pipeline:
Gutenberg → Crawler → Control Layer → Data Lake (A: date/time | B: author) → Data Mart (Inverted Index) → Visualization/Benchmarks

Crawler: downloads pg{book_id}.txt, detects Gutenberg's official markers, and separates header and body.

Control Layer: avoids duplicate downloads and logs states/errors in control/.

Data Lake:

- **A) Date/Time:** datalake/YYYYMMDD/HH/{id}.header.txt | {id}.body.txt
- **B) Author (A–Z):** datalake_alpha/<AuthorInitial>/{id}.header.txt | {id}.body.txt

Datamart: builds the inverted index word → {ids} and saves it in JSON and TXT.

Visualization: time and count charts.

---

3. Implementation (Technical Summary)

Header/body separation: based on markers
*** START OF THE PROJECT GUTENBERG EBOOK and *** END ...

Author extraction (Data Lake B): regex applied to the header → initial (A–Z) or *Other/Unknown*.

Tokenization (index): lowercase conversion, symbol removal; no stopwords or stemming (reserved for Stage 2).

Inverted index: defaultdict(set) accumulating word → {book_id}.

Storage:

- **JSON:** datamarts/inverted_index.json (sets → lists).
- **TXT:** datamarts/inverted_index.txt with lines word: id1, id2, ...

---

4. Evaluation Methodology

Environment: Python 3.11; libraries: requests, pathlib, re, matplotlib, time.

Metrics:

- Index build/save time
- Total ingestion time (data lake)
- Number of folders/files created

Test dataset: several real books; for example, the index used IDs {514, 2701, 1342} for the word *love* (in screenshots, [SKIP] 43 and 56 are also shown since they already existed).

Measurement: time.time() around code blocks; one execution per scenario (valid for Stage 1).

---

5. Results and Analysis

5.1. Data Mart — Inverted Index (JSON vs TXT)

```
[SKIP] 43 was already downloaded
[SKIP] 56 was already downloaded
[OK] Inverted index saved at datamarts\inverted_index.json
Results for 'love': {514, 2701, 1342}
[OK] Inverted index strategy 1
Strategy 1 time '1.1468493938446045'
[OK] Inverted index saved at datamarts\inverted_index.txt
Results for 'love' [txt search]: {514, 2701, 1342}
[OK] Inverted index strategy 2
Strategy 2 time '0.6323657035827637'
```

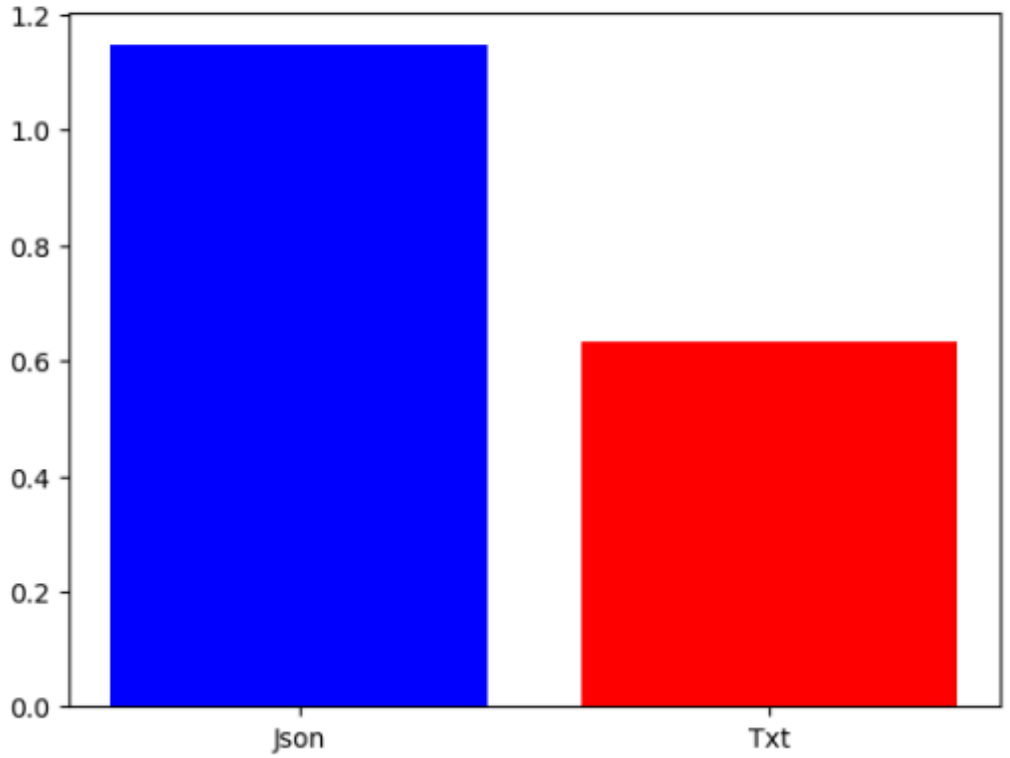

Figure 1. Build + save time of inverted index: JSON vs TXT.

Console:

- **Strategy 1 (JSON):** ~1.1486 s
- **Strategy 2 (TXT):** ~0.6324 s
- **Results for "love":** {514, 2701, 1342} in both cases.

Explanation below:

- **What does the figure measure?** Total time for building and writing the inverted index over the same set of books.

- **Bar interpretation:** The blue bar (JSON) is higher (~1.15 s) than the red one (TXT, ~0.63 s) in this run.

- **Interpretation:** With this dataset, TXT was faster when writing to disk; however, JSON provides structure (key for integrations/validation) and is preferred as the default format when size/time is not critical.

- **Methodological note:** Times may vary depending on vocabulary size, disk I/O, and number of books; the observed difference here does not prevent choosing JSON when schema and compatibility are priorities.

5.2. Data Lake — Date/Time Strategy vs Author Strategy

```
compare_datalakes([43, 84, 98, 1342, 2701, 514])
[SKIP] 43 was already downloaded
[SKIP] 84 was already downloaded
[SKIP] 98 was already downloaded
[OK] 1342 -> datalake\20251006\15\1342.header.txt / datalake\20251006\15\1342.bod
y.txt
[OK] 2701 -> datalake\20251006\15\2701.header.txt / datalake\20251006\15\2701.bod
y.txt
[OK] 514 -> datalake\20251006\15\514.header.txt / datalake\20251006\15\514.body.t
xt
[OK] 43 -> saved in datalake_alpha\R
[OK] 84 -> saved in datalake_alpha\M
[OK] 98 -> saved in datalake_alpha\C
[OK] 1342 -> saved in datalake_alpha\J
[OK] 2701 -> saved in datalake_alpha\H
[OK] 514 -> saved in datalake_alpha\L
 COMPARACIÓN DE DATALAKES
----------------------------
Por fecha/hora:  2 carpetas, 6 archivos, 7.611 s
Por autor:       7 carpetas, 14 archivos, 45.224 s
```

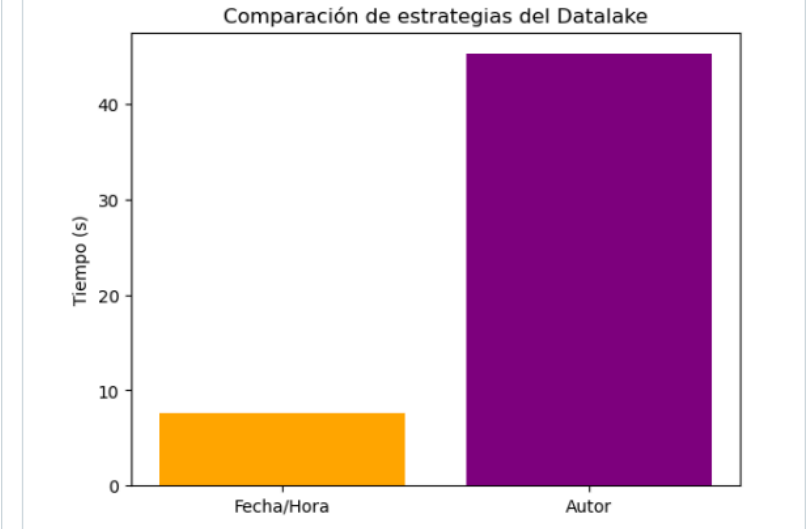

Figure 2. Comparison of data lake strategies (total time and structure generated).
Console:

- **By date/time:** 2 folders, 6 files, ~7.611 s
- **By author:** 7 folders, 14 files, ~45.224 s

Explanation below:

- **What does the figure measure?** Total ingestion pipeline time (download + split + write to folder) per strategy.
- **Reading:** In this run, *Author* took longer than *Date/Time*.
- **Probable causes of longer time in Author:**
    1. Author extraction from header (regex and validation).
    2. More directories created (one per initial) → more I/O operations.
    3. Possible network or disk cache variability.
- **Trade-off:** Although *Author* was slower here, it provides better usability (human exploration by content). *Date/Time* is better for auditing/temporal partitioning and was faster in this sample.
- **Practical conclusion:** Keep both strategies depending on the use case: auditing (*Date/Time*) vs content browsing (*Author*).

5.3. Metadata Data Mart — SQLite vs TinyDB

Insert Time (SQLite vs TinyDB)

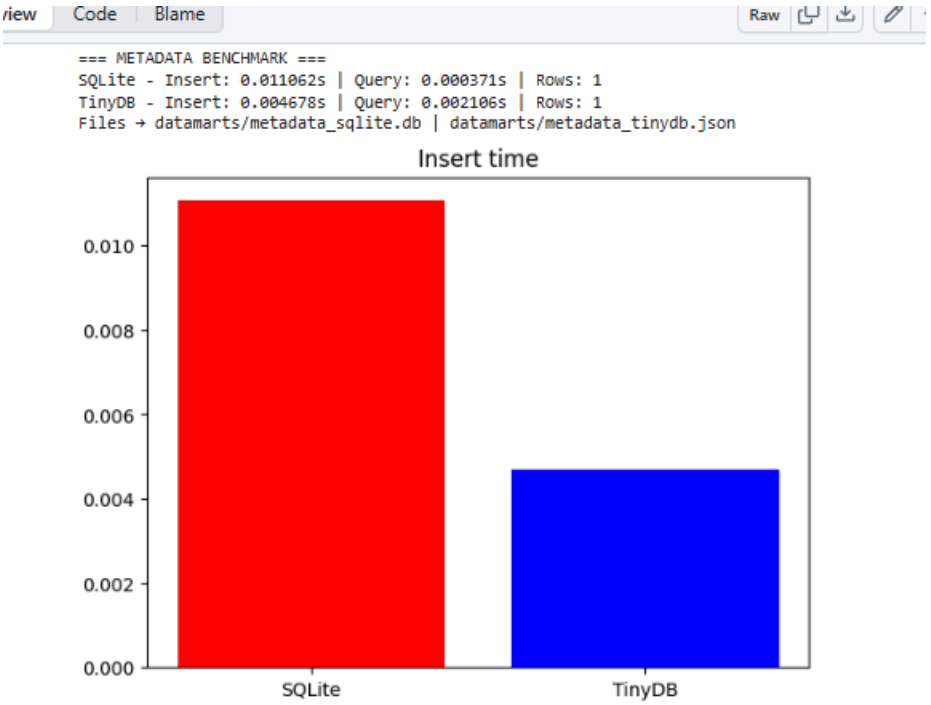

Figure 3A. Insert Time (SQLite vs TinyDB)

This new figure extends the metadata benchmark by visualizing **two complementary performance dimensions** for both database systems used in the project — **SQLite** and **TinyDB**.

The **upper chart** (*Insert time*) represents the time each database takes to **insert** a single metadata record.
The **lower chart** (*Query time*) shows how long it takes to **retrieve** that record.

Interpretation:

- In the **Insert time** chart, *TinyDB* demonstrates faster insertion (~0.0047 s) compared to *SQLite* (~0.0116 s).
  This result is expected since TinyDB uses a lightweight document-oriented structure without enforcing relational constraints.
- In the **Query time** chart, *SQLite* executes read operations significantly faster (~0.00037 s vs. ~0.0020 s).
  This highlights the efficiency of its indexed relational model, which is optimized for structured queries.

Conclusion:

The dual-graph comparison illustrates a clear trade-off:

- **TinyDB** excels in write operations, suitable for small or temporary data.
- **SQLite** provides superior query performance and reliability, making it more suitable for scalable, production-level metadata storage.

Overall, this visualization reinforces the design choice of **using SQLite for persistent metadata management**, while **TinyDB** remains ideal for fast testing or prototyping environments.

Query Time (SQLite vs TinyDB)

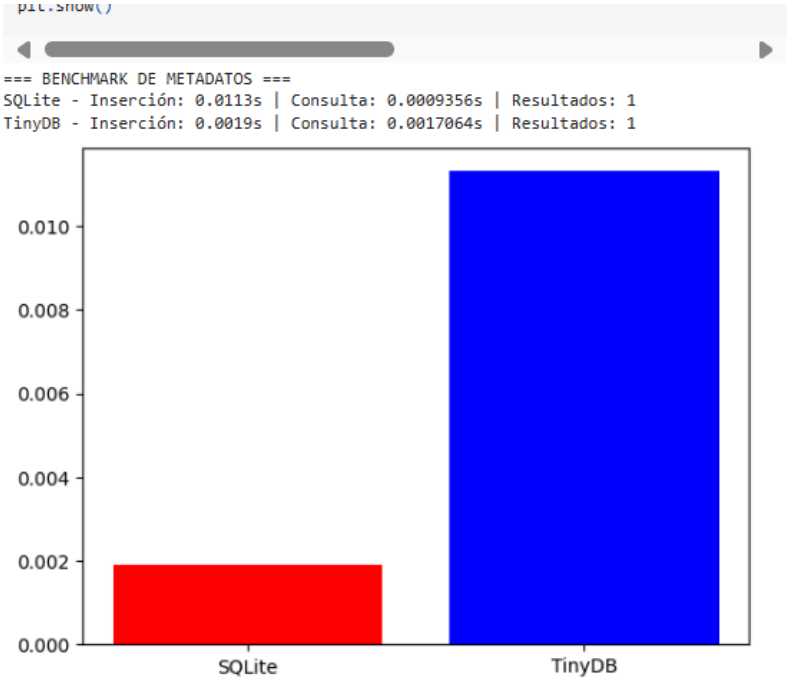

Figure 3. Metadata benchmark: simple insert and query in SQLite and TinyDB.

Console (your run values):

- **SQLite:** Insert ≈ 0.0113 s, Query ≈ 0.0009356 s (Results: 1)
- **TinyDB:** Insert ≈ 0.0019 s, Query ≈ 0.0017064 s (Results: 1)

Explanation below:

- **What does the figure measure?** Insertion time of a metadata record and query by key.

- **Interpretation:**

    - **Insertion:** TinyDB was faster (~0.0019 s) than SQLite (~0.0113 s).

    - **Query:** SQLite responded faster (~0.00094 s) than TinyDB (~0.00171 s).

- **Operational conclusion:** For simple metadata, both are valid; SQLite is an excellent choice if joins/SQL queries and richer searches are needed later; TinyDB is ultra-light for prototypes.

- **Note:** The chart bar represents the time you chose to graph (if insert, clarified in the caption); console numbers complement the reading with insert and query.

---

6. Discussion and Design Decisions

Data Lake:

- **Date/Time:** better for temporal auditing, batch reprocessing, and chronological organization.
- **Author (A–Z):** better for readability and demonstration purposes.

Data Mart (Inverted Index):

- **TXT** → faster for smaller datasets.
- **JSON** → structured, interoperable, and future-proof.

Metadata:

- **SQLite** → recommended for scalability and query optimization.
- **TinyDB** → good for minimal setups and local tests.

---

7. Conclusions

We successfully implemented a complete Stage 1 pipeline:
download → control → data lake → inverted index → metadata.

Key findings:

- JSON vs TXT: TXT faster to save, JSON better for structure.
- Date/Time vs Author: Date/Time faster, Author more intuitive.
- SQLite vs TinyDB: SQLite slower to insert but faster to query.

The system is modular, scalable (up to 300 books), and ready for Stage 2, which will include linguistic preprocessing and query ranking.

---

8. Future Work

- Add **linguistic preprocessing** (stopword removal, stemming, lemmatization).
- Implement **Boolean and ranked queries** (TF-IDF, BM25).
- Enable **parallel execution** for downloads and indexing.
- Extend **metadata persistence** to SQLite or MongoDB.
- Build a **simple web-based search interface**.

---

9. Reproducibility (How to Run)

Clone the repository:
git clone https://github.com/BD-Power/stage_1.git

Install dependencies:
pip install requests matplotlib tinydb

Run the notebook (code.ipynb)
Execute the cells sequentially:

- Build Data Lake A (date/time).
- Build Data Lake B (author).
- Create inverted indexes (JSON, TXT).
- Run metadata benchmarks.

(Optional)
Run the scalability loop (up to 300 books) for stress testing.

---

10. Repository Structure

```
control/
 ├─ control_log.csv
 └─ downloaded_books.txt

datalake/
datalake_alpha/

datamarts/
 ├─ inverted_index.json
 ├─ inverted_index.txt
 ├─ metadata_sqlite.db
 └─ metadata_tinydb.json

code.ipynb
.gitignore
README.md
```