Inverted Index (Stage-1)

Diego Muñoz, Sofía Travieso, Judith Portero, Lucía Cruz, Paula Velasco

October 6, 2025

1 Introduction and Goal

This report provides a detailed scientific description of the system developed in the Stage-1 project. Unlike later stages focused on large-scale integration and automation, the primary objective of Stage-1 is to identify the most optimal combination of Datalake and Datamart, the core datalayer, for the project's text search engine. The evaluation focuses on both technical efficiency (ingestion speed, query performance, scalability) and qualitative criteria (organization, transparency, and interpretability of stored data).

To achieve this, the Stage-1 system implements a modular and reproducible framework capable of ingesting, storing, and benchmarking large volumes of textual data. The experiments compare multiple storage backends through standardized benchmarking and structured qualitative analysis.

It includes:

- Data ingestion and structured storage of raw text.
- Multiple data lake and datamart implementations.
- Comparative benchmarking for performance and usability.
- Foundational modules for indexing and search integration.

This methodological approach allows assessing the strengths and limitations of each storage strategy, ultimately determining the most balanced and efficient datalayer architecture for the search engine's future development.

2 Repository Architecture

```
root/
  search/
     indexer.py
     query_engine.py
  shared/
     repository.py
  storage/
     datalake/
        datalake_tria.py
        datalake_sql.py
     datamarts/
         datamart_base.py
```

```
        datamart_shelve.py
        datamart_sqlite.py
  utils/
     API.py
  benchmark.py
  main.py
```

The repository follows a clear separation of concerns: ingestion, persistence, search, and evaluation.

3 Module Documentation

3.1 storage/datalake/

`datalake_tria.py`

Implements a filesystem-based Datalake.

- Saves text under date/hour partitions: `datalake/YYYYMMDD/HH/`.
- Extracts header/body using Gutenberg markers.
- `save_raw()` – writes text files and returns metadata.
- `iter_books()` – iterates over saved book pairs.

`datalake_sql.py`

Implements DatalakeSQL using SQLite.

- Creates table `books(book_id, header, body, ingestion_time)`.
- `save_raw()` inserts or replaces entries.
- `get_book()` and `iter_books()` retrieve content.

3.2 storage/datamarts/

`datamart_base.py`

Abstract base class Datamart defines common interface:

- `upsert_many(rows)`
- `get_by_author(author)`
- `get_by_title(title)`
- `close()`

`datamart_shelve.py`

DatamartShelve – Lightweight object persistence using Python's `shelve`.

- Maintains author and title indices.
- Upserts multiple book records.
- Supports fast lookup and simple persistence.

`datamart_sqlite.py`

DatamartSQLite – Relational datamart using SQLite.

- Table fields: `book_id, title, author, language, header_path, body_path, ingested_at`.
- Uses `ON CONFLICT` upsert logic.
- Indexed by author and title.

3.3 utils/

`API.py`

API client for fetching texts from Project Gutenberg.

- Candidate URLs: `https://www.gutenberg.org/files/{id}/{id}.txt`, etc.
- `fetch_gutenberg_text(book_id)` – tries multiple URLs and returns text or None.
- Uses HTTP headers for polite access.

3.4 Root scripts

`main.py`

Main ingestion script:

- Uses API, Datalake, and DatalakeSQL.
- Splits texts into header/body/footer via regex markers.
- Saves results to both datalakes.
- Example IDs: `[11, 84, 98, 1661, 2701, 1342, 1952, 4300]`.

`benchmark.py`

The benchmark.py module constitutes the core of the performance evaluation in this project. It systematically measures, compares, and visualizes the efficiency of different storage backends under controlled experimental conditions.

- The StorageBenchmark class generates synthetic datasets that emulate text ingestion at varying scales (e.g., from hundreds to thousands of documents). Each dataset is dynamically created in memory to ensure reproducibility and independence from external data sources.
- The benchmark executes standardized write and read operations across four storage layers:
  1. Datalake_File (tria) – hierarchical file-based storage.
  2. Datalake_SQL – relational SQL-based lake using SQLite as backend.
  3. Datamart_Shelve – lightweight key-value Python storage.
  4. Datamart_SQLite – optimized relational analytical store.

  This allows direct and fair performance comparison among heterogeneous storage designs.
- During execution, it measures multiple quantitative metrics: total ingestion time, average I/O throughput (records per second), retrieval latency, and storage footprint. Results are stored in structured tables for further analysis and interpretation.

- The benchmark automatically produces bar plots and tables using Matplotlib and Pandas. These visualizations summarize comparative performance across storage strategies and dataset sizes, allowing a clear interpretation of trade-offs.
- It supports command-line execution with configurable parameters (e.g., dataset size, repetition count, output path), enabling reproducible tests and integration with CI pipelines.
- The script is fully modular, allowing individual benchmarking of each component or a complete evaluation of the entire ingestion-to-query workflow. It can also be extended to incorporate new storage strategies in future iterations.

In summary, benchmark.py serves as the experimental backbone of the project. It provides both quantitative and qualitative insights that guided the final architectural decisions — particularly the adoption of the Datalake_File (tria) for structured and transparent storage, and the Datamart_SQLite for high-performance analytical querying.

4 Dependencies

The system uses:

- `requests` – for HTTP text retrieval.
- `sqlite3` and `shelve` – for local storage.
- `matplotlib` – for plotting benchmarks.
- `re, collections, datetime, pathlib` – standard Python utilities.

5 Testing and Validation

Testing can be performed by:

- Running `python benchmark.py` to validate performance.
- Checking ingestion results in `datalake_sql/books.db`.
- Executing small-scale search queries via `QueryEngine`.

6 Comparative Analysis of Datalakes and Datamarts

6.1 Quantitative Benchmark Results

Table 1 summarizes the performance benchmarks obtained from the system's storage layer. The results clearly show that SQL-based backends significantly outperform file- or object-based approaches. Quantitatively, Datalake_SQL is the fastest data lake implementation, achieving approximately 175× higher throughput than the file-based approach. Among the datamarts, Datamart_SQLite leads with 33,800 items/s, showing a strong advantage for analytical queries and upsert operations.

6.2 Qualitative Comparison

Beyond raw performance, each storage strategy presents distinct qualitative characteristics that justify its use in different contexts.

Component	Total Time (s)	ms/item	Items/s
Datalake_File	0.583	2.92	343
Datalake_SQL	0.033	0.17	5975
Datamart_SQLite	0.006	0.03	33804
Datamart_Shelve	0.375	1.88	533

Table 1: Benchmark results for Datalake and Datamart implementations.

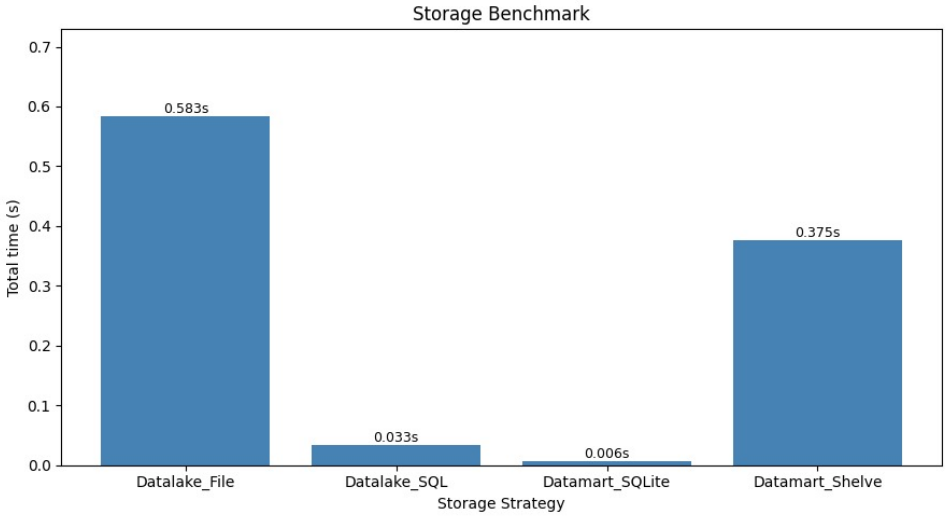

Figure 1: Performance comparison of storage strategies based on total ingestion time. The bar chart confirms that SQL-based solutions (Datalake_SQL and Datamart_SQLite) achieve far lower latency and higher throughput than file-based or Shelve backends.

Datalake_File (tria) This structure organizes text files hierarchically by date and hour partitions (e.g., `datalake/YYYYMMDD/HH/`). Although slower to read or write compared to databases, it offers high transparency, interpretability, and manual accessibility. When datasets grow to thousands of books, the logical folder structure allows users to intuitively locate and verify individual files, which simplifies debugging, provenance tracking, and manual curation.

Datalake_SQL The SQL-based lake prioritizes speed and integration. It stores all content in a relational table, enabling fast ingestion and indexed access. This design is ideal for automated pipelines or when frequent bulk ingestion and queries are required. However, it lacks the visibility of the tria structure since the data is encapsulated within a database.

Datamart_Shelve Shelve-based datamarts are conceptually simple and tightly integrated with Python's native serialization model. They are easy to implement for small or experimental datasets but degrade in performance as the collection scales. Random access and concurrent queries are slower, limiting their usability in production environments.

Datamart_SQLite SQLite provides the best compromise between structural order, performance, and analytical flexibility. It allows relational queries by title, author, or language and efficiently handles thousands of records with minimal overhead. Moreover, it ensures durability and portability — the datamart can be transferred as a single `.db` file, preserving metadata and schema integrity.

6.3 Summary and Justification

The results and qualitative insights justify the following design conclusions:

- Datalake_SQL is the preferred option for ingestion, offering high throughput and easy integration with data pipelines.
- Datamart_SQLite is the optimal analytical layer for querying and exploration due to its relational structure and exceptional speed.
- Datalake_File (tria) remains valuable for human-readable storage, validation, and inspection, particularly in early stages of development or for research reproducibility.

In summary, the chosen architecture leverages the tria-based Datalake for its transparency and structural organization, combined with an SQLite Datamart for analytical efficiency. This hybrid design ensures a balance between interpretability, reliability, and scalability, making it highly suitable for both experimental research and structured text management.

7 Conclusions

The Stage-1 development followed a systematic and comparative methodology aimed at identifying the most suitable data storage approach for scalable and interpretable text management.

Rather than focusing solely on implementation details, the process emphasized experimental evaluation, structured analysis, and methodological justification.

Through iterative benchmarking and qualitative assessment, the study contrasted multiple storage paradigms—file-based, SQL-based, and Python-native solutions—under consistent experimental conditions.

This methodological approach ensured that conclusions were not based on intuition or isolated performance tests, but on a balanced analysis combining quantitative metrics and qualitative reasoning.

The final decision to adopt the tria-based Datalake and the SQLite Datamart reflects this evidence-driven methodology. While the tria structure excels in organization, traceability, and human interpretability, SQLite provides analytical robustness and relational expressiveness.

Together, they form a coherent and adaptable data architecture supported by a transparent and reproducible experimental framework.

Overall, the methodological rigor of Stage-1 establishes a solid foundation for future stages, where scalability, automation, and data integration will be further explored using the same evidence-based design principles.