Inverted Index (Stage-1)

DSLJP

06/10/2025

1 Introduction

This report provides a detailed scientific description of the system developed in the *Stage-1* project. The project implements a modular text ingestion and search pipeline using data lakes, datamarts, and a custom query engine, with benchmarking and API integration. Each module is analyzed with emphasis on functionality, purpose, and its role in the architecture. The goal of the Stage-1 system is to implement a complete end-to-end text ingestion, indexing, and search solution built around open digital libraries such as Project Gutenberg. It includes:

- Data ingestion (raw text download and storage).
- Multi-backend data lakes and datamarts.
- Search and ranking engine.
- Benchmarking utilities for storage performance.

The architecture is modular, reproducible, and extensible a foundation for advanced retrieval and analytics systems.

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 search/

`indexer.py`

Implements two core classes for indexing:

- **SimpleInvertedIndex:** Builds a term-frequency inverted index.
  - `index_document(doc_id, text)` – Tokenizes and indexes non-stopword terms.
  - `boolean_or(terms)` / `boolean_and(terms)` – Logical retrieval.
  - `tf_idf_score(terms)` – Weighted retrieval using TF–IDF.
- **PositionalInvertedIndex:** Stores term positions to allow phrase search.
  - `phrase_search(phrase)` – Finds exact phrases across documents.
  - `stats()` – Returns number of docs and unique terms.

3.2 shared/

`repository.py`

Provides document storage and tokenization utilities.

- **STOP_WORDS:** predefined list of ignored terms.
- **tokenize(text)** – cleans and lowercases words, removing stopwords.
- **DocumentRepo:** in-memory document repository.
  - `add(doc_id, text, meta=None)` – store document and metadata.
  - `get(doc_id)` – retrieve text.

3.3 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.

3.4 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.

3.5 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.

3.6 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.

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.

6.2 Qualitative Comparison

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

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.

`bench.png`

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.

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* system successfully demonstrates a modular, reproducible pipeline for large-scale text ingestion and retrieval. Its hybrid architecture—combining file-based and SQL datalakes, analytical datamarts, and flexible search indexing—forms a strong foundation for advanced data engineering and information retrieval research.