Multi-Source Article ETL Pipeline with Data Quality Validation

Author: Giorgio Evola
Course: Data Management Project
Date: February 2026

---

1. Introduction

The goal of this project is to design and implement a reproducible ETL pipeline that ingests technical articles from heterogeneous web sources (GeeksforGeeks and Medium), validates their quality, and integrates them into a unified relational schema.
The focus is on **data quality management**, **schema validation**, and **incremental loading patterns** rather than on advanced analytics or machine learning.

The implementation is fully based on Python and SQLite, with Pandera for declarative data quality checks. Auxiliary components for web scraping and LLM-based enrichment are already designed and implemented as separate modules, but are considered **next steps** rather than part of the core pipeline.

---

2. Project Components and Their Role

The project is organized as a modular Python package under `src/`:

- **`main.py`**
  Entry point of the application.
  Orchestrates the complete ETL pipeline:
  - database initialization (`create_database`, `delete_database`)
  - ingestion of GeeksforGeeks CSV and Medium RSS feeds into staging tables
  - data quality validation and quarantine handling
  - loading of clean records into the dimensional table `dim_articles`
  - basic monitoring by printing samples from `dq_quarantine` and `dim_articles`.

- **`utils/sqlite_db.py`**
  Database access and schema management layer.
  Responsibilities:
  - definition of all table schemas in the `SCHEMAS` dictionary:
    - `stg_gfg_articles` (GFG staging)
    - `stg_medium_articles` (Medium staging)
    - `dim_articles` (unified dimensional table)
    - `dq_quarantine` (data quality quarantine)
  - creation and deletion of the SQLite database (`create_database`, `delete_database`)
  - generic table creation (`create_table`) and dropping (`drop_table`)
  - ingestion of CSV datasets with optional transformations (`ingest_csv_to_db`)

  - helper to run arbitrary SQL queries and return a DataFrame (`query_to_df`)
  - generic DataFrame-to-SQL insert function with chunking (`insert_df_to_db`).

- **`utils/rss.py`**
  Acquisition and transformation of Medium RSS feeds.
  Responsibilities:

  - `RSS_FEEDS`: list of Medium RSS URLs used in the project
  - `get_feed_df(rss_url)`: parse RSS feed with `feedparser` and convert entries to a Pandas DataFrame
  - `transform_medium(df)`: flatten nested RSS structures (e.g., `tags`, `authors`, `title_detail`) into JSON strings, normalize dates to `YYYY-MM-DD`, and deduplicate by `link`/`id_rss`
  - `ingest_rss_to_db(rss, conn, table_name, transform_func)`: orchestrate parsing, transformation, and bulk insertion into `stg_medium_articles`.

- **`utils/dq.py`**
  Data Quality (DQ) and quarantine management module.
  Responsibilities:

  - definition of two Pandera schemas:
    - `SCHEMA_GFG_ARTICLES` for `stg_gfg_articles`
    - `SCHEMA_MEDIUM_ARTICLES` for `stg_medium_articles`
  - `dq_pandera(df, table_name)`: run schema validation in *lazy* mode to collect all errors at once
  - `quarantine_failed_records(...)`: insert failed records into `dq_quarantine` with standardized metadata (source table, primary key, number of columns, error summary)
  - `extract_failed_records_general(errors, df_original, conn, table_name, pk_col)`: split the dataset into clean and failed records, send failed ones to quarantine, and return the clean subset plus the number of quarantined records.

- **`utils/dimensions.py`**
  Dimensional loading and post-validation integration.
  Responsibilities:

  - `mark_records_as_processed(conn, table_name, pk_col, pk_values)`: set `processed = 1` for successfully loaded staging rows
  - `extract_first_tag(tags_json)` / `extract_first_author(authors_json)`: parse Medium JSON arrays and extract the first tag/author
  - `staging_to_dim_articles_gfg(df_clean, conn)`: transform clean GFG records into the `dim_articles` schema and insert them using `insert_df_to_db`, then mark staging records as processed
  - `staging_to_dim_articles_medium(df_clean, conn)`: analogous transformation and load for Medium RSS data
  - `get_dimension_stats(conn)`: compute high-level statistics over `dim_articles` (total records, counts by source, publication date range).

- **`utils/scraping.py`** (Next-step component – not used in the main pipeline)
  Async web scraping of full Medium article content using Playwright and cookies.
  Provides:

    - `fetch_medium_article(...)`: load HTML pages, extract main article content, apply text cleanup with a failsafe mechanism.

- `utils/llm.py` (Next-step component – not used in the main pipeline)
  LLM integration via the OpenAI API.
  Provides:

    - loading of YAML prompt templates (`load_prompt`)
    - generic OpenAI Chat Completions call (`call_openai`)
    - `analyze_article(...)`: end-to-end function that scrapes a Medium article, checks minimum length, and sends content to an LLM for summarization and quality assessment.

These components cooperate to provide a complete ETL and DQ pipeline, with a clear separation between **acquisition**, **validation**, **integration**, and **future enrichment**.

---

3. Research Questions

The project aims to explore the following research questions:

1. **Data Quality across Heterogeneous Sources**
    - What are the most common data quality issues affecting article metadata for GeeksforGeeks vs Medium (e.g., missing fields, invalid URLs, inconsistent dates, malformed JSON)?
    - How frequently do these issues occur in each source?
2. **Integration and Harmonization of Heterogeneous Metadata**
    - Can article metadata from two different platforms (CSV export vs RSS) be normalized into a coherent dimensional schema without heavy manual intervention?
    - Which transformations are required to align identifiers, authors, dates, and categories?

Future work (partially enabled by `scraping.py` and `llm.py`) will extend these questions by analyzing LLM-generated *utility scores* and summaries, and comparing them with traditional engagement metrics when available.

---

4. Datasets and Acquisition Methods

4.1 GeeksforGeeks Articles (Dataset 1)

- **Source**: Public dataset of GeeksforGeeks articles (exported as CSV, originally from Kaggle or similar).
- **File**: `data/GeeksforGeeks_articles.csv`
- **Content** (simplified):
    - `article_id` (or derived from link)
    - `title`
    - `author_id`
    - `last_updated`
    - `link`
    - `category` (topic / tag)

Acquisition & Ingestion Workflow:

1. The CSV file is downloaded manually once and stored under `data/`.
2. `ingest_csv_to_db(...)` in `sqlite_db.py`:
    - reads the CSV into a Pandas DataFrame
    - applies `transform_gfg(df)`:
        - extracts `article_id` from the article `link`
        - parses `last_updated` from human-readable format (e.g., `"02 Jan, 2023"`) to `datetime`
        - generates a normalized date string (`last_updated_str`)
        - selects and renames only the necessary fields for staging
    - ensures the staging table `stg_gfg_articles` exists using the schema in `SCHEMAS`
    - inserts the transformed DataFrame into `stg_gfg_articles`, which includes:
        - `id` (AUTOINCREMENT primary key)
        - `ingested_at` (timestamp)
        - `processed` (boolean flag default `false`)
        - metadata fields (`article_id`, `title`, `author_id`, `last_updated`, `link`, `category`).

4.2 Medium RSS Feeds (Dataset 2)

- **Source**: Medium RSS feeds for relevant tags:
    - `https://medium.com/feed/tag/data-quality`
    - `https://medium.com/feed/tag/data-observability`
    - `https://medium.com/feed/tag/data-governance`
    - `https://medium.com/feed/tag/data-lineage`
    - `https://medium.com/feed/tag/data-engineer`
- **Acquisition & Ingestion Workflow:**

1. For each RSS URL in `RSS_FEEDS`, `ingest_rss_to_db(...)` is called from `main.py`.
2. `get_feed_df(rss_url)` uses `feedparser` to parse the RSS/Atom feed and returns a DataFrame of entries; BOZO exceptions are logged and handled.
3. `transform_medium(df)`:
    - keeps a subset of relevant columns (`title`, `summary`, `link`, `id`, `published`, `updated`, `tags`, `authors`, etc.)
    - renames `id` to `id_rss` to build a stable article identifier
    - serializes nested structures (`title_detail`, `summary_detail`, `tags`, `authors`) as JSON strings
    - normalizes date fields `published` and `updated` to `YYYY-MM-DD`, using `errors='coerce'` and replacing invalid dates with empty strings
    - cleans scalar fields and converts them to strings
    - deduplicates entries by `link` and `id_rss` (keeping the latest row).
4. `insert_df_to_db` inserts the transformed dataset into `stg_medium_articles`, whose schema includes:
    - `id` (AUTOINCREMENT)
    - `ingested_at`, `processed`
    - `title`, `link`, `id_rss`, date fields, and JSON-serialized metadata.

---

5. Process Overview: Acquisition, Storage, Analysis, Cleaning, Integration, Enrichment

5.1 Acquisition

- **Batch source ingestion** from:
  - CSV file for GeeksforGeeks
  - RSS feeds for Medium.
- All raw inputs are transformed to fit the staging schemas while preserving as much metadata as possible.

5.2 Storage

- **Database Engine**: SQLite, via `sqlite3` library.
- **Logical Layers**:
  - **Staging layer**:
    - `stg_gfg_articles`
    - `stg_medium_articles`
      These tables store raw, minimally transformed data with a `processed` boolean indicating whether a record has already passed the ETL pipeline.
  - **Dimensional layer**:
    - `dim_articles`
      Unified representation of article metadata across sources:
    - `id` (surrogate key)
    - `article_id` (business key, unique)
    - `source_platform` (`GFG` or `Medium`)
    - `title`, `author`, `pub_date`, `link`, `category`
    - `is_valid`, `created_at`, `updated_at`.
  - **DQ layer**:
    - `dq_quarantine`
      Contains records that failed validation, with metadata about errors.

5.3 Exploratory Analysis (Lightweight)

The project does not focus on heavy exploratory analysis, but several simple queries and prints are used for inspection:

- head of staging tables after ingestion
- head of `dim_articles` and `dq_quarantine` after the ETL run
- `get_dimension_stats(conn)` to compute total counts and date ranges.

These tools are sufficient for validating that the pipeline behaves as expected and for inspecting typical validation failures.

5.4 Cleaning and Data Quality Validation

Central idea:
All cleaning and validation rules are expressed as *declarative Pandera schemas* in `dq.py`. This avoids scattering manual checks across the code and makes the pipeline easier to extend and reason about.

5.4.1 GFG Schema: `SCHEMA_GFG_ARTICLES`

Key rules:

- `article_id`:
  - required string
  - cannot be empty
- `title`:
  - required string
  - minimum length 6, maximum length 200
- `author_id`:
  - nullable string
  - must match regex `^[a-zA-Z0-9_-]+$` if not null
- `last_updated`:
  - nullable string
  - must match pattern `YYYY-MM-DD HH:MM:SS` (normalized in ingestion)
- `link`:
  - nullable string
  - must contain a valid HTTP/HTTPS URL.

5.4.2 Medium Schema: `SCHEMA_MEDIUM_ARTICLES`

Key rules:

- `id_rss`:
  - required non-empty string
- `title`:
  - required string length between 3 and 500
- `title_detail`, `summary_detail`:
  - nullable strings
  - when present, must look like JSON (`{` or `[` prefix)
- `summary`:
  - length ≤ 5000 characters
- `link`:
  - contains a valid HTTP/HTTPS URL
- `published`, `updated`:
  - nullable strings in format `YYYY-MM-DD HH:MM:SS` (normalized in ingestion)
- `tags`, `authors`:
  - nullable strings
  - when present, must start with `[` to represent a JSON array.

5.4.3 Validation Flow

1. From `main.py`, for each source:
   - select unprocessed rows from the staging table (`processed = false`)
   - pass the resulting DataFrame and table name to `dq_pandera(df, table_name)`.
2. `dq_pandera` applies the correct schema and returns:
   - `None` if validation passes

- a `SchemaErrors` object with all failures if any check fails.

3. `extract_failed_records_general`:
   - ensures `dq_quarantine` exists (via `SCHEMAS` and `create_table`)
   - extracts unique failing row indices from `errors.failure_cases`
   - builds a DataFrame of failed records and sends it to `quarantine_failed_records`, which inserts them into `dq_quarantine` with standardized columns:
     - `source_table`, `pk_column_name`, `pk_value`
     - `total_columns`
     - `validation_error` (aggregated list of failing columns per record).
   - returns the cleaned DataFrame (all failed rows removed) and the number of quarantined records.

This architecture guarantees that **no invalid record reaches the dimensional table**, while preserving detailed diagnostics for offline inspection.

5.5 Integration into the Dimensional Model

Integration is handled by `staging_to_dim_articles_gfg` and `staging_to_dim_articles_medium` in `dimensions.py`.

5.5.1 GFG → `dim_articles`

- Input: `df_clean` from `extract_failed_records_general` for `stg_gfg_articles`.
- Transformations:
  - `article_id` → `article_id`
  - `title` → `title`
  - `author_id` → `author`
  - `last_updated` (datetime) → `pub_date` as `YYYY-MM-DD`
  - `link` → `link`
  - `category` → `category`
  - `source_platform` is set to `'GFG'`
  - `is_valid` is set to `1`.
- Persistence:
  - a DataFrame with the transformed columns is sent to `insert_df_to_db(dim_df, 'dim_articles', conn)`
  - after successful insertion, `mark_records_as_processed` updates `stg_gfg_articles.processed` to `1` for the corresponding staging `id` values.

5.5.2 Medium → `dim_articles`

- Input: `df_clean` for `stg_medium_articles`.
- Transformations:
  - `id_rss` → `article_id`
  - `title` → `title`
  - `authors` (JSON array) → `author` via `extract_first_author`
  - `published` (string `YYYY-MM-DD`) → `pub_date`
  - `link` → `link`
  - `tags` (JSON array) → `category` via `extract_first_tag`
  - `source_platform` set to `'Medium'`

    - `is_valid` set to `1`.
- Persistence and flagging:
    - DataFrame inserted into `dim_articles`
    - `stg_medium_articles.processed` set to `1` for the staging `id` values.

Although the current implementation relies on `INSERT ... TO_SQL` rather than explicit `ON CONFLICT` SQL, uniqueness is enforced by the `article_id` unique constraint in `dim_articles`. In practice, the ETL design assumes *immutable* article identifiers per source; rerunning the pipeline from a clean database avoids duplicates.

5.6 Enrichment (Future Work)

The project includes **prototype components** for content enrichment:

- `scraping.fetch_medium_article(...)` can retrieve the full text of a Medium article, bypassing paywalls with user-provided cookies.
- `llm.analyze_article(...)` can send this content to an LLM (OpenAI) using YAML-configured prompts, obtaining:
    - a summary
    - qualitative assessments (e.g., depth, clarity, originality)
    - a possible *utility score*.

These functionalities are intentionally decoupled from the main ETL so that the current project remains reproducible without requiring external APIs or browser automation. They also define a clear roadmap for future research questions on AI-based content evaluation.

---

6. Software Architecture

6.1 Architectural Style

The project follows a **layered architecture** with strong separation of concerns:

1. **Data Access Layer (`sqlite_db.py`)**
    - Responsible for physical storage, schema creation, and generic I/O.
    - Maintains a dictionary of table schemas for consistent table creation.
2. **Ingestion Layer (`sqlite_db.py`, `rss.py`)**
    - Source-specific logic to read external datasets and transform them into staging format.
    - Uses Pandas for CSV and `feedparser` for RSS, plus helper transformations.
3. **Data Quality Layer (`dq.py`)**
    - Centralized declarative validation with Pandera.
    - Manages quarantine, error tracking, and cleansing (split into valid and invalid subsets).
4. **Integration Layer (`dimensions.py`)**
    - Handles transformations from staging schemas to the `dim_articles` dimensional schema.
    - Implements processed flagging and basic statistics.

5. **Orchestration Layer (`main.py`)**
    - Defines the ETL workflow, step order, and logging messages.
    - Currently orchestrated via a simple Python script, but easily portable to a scheduler or workflow manager (e.g., cron, Airflow) in the future.
6. **Enrichment Layer (Prototype) (`scraping.py`, `llm.py`)**
    - Not used in the main run, but designed to plug into the pipeline to enrich `dim_articles` with full text and LLM-based features (summary, utility score, topics).

6.2 Key Technical Choices and Rationale

- **SQLite as the Database Engine**
    - Lightweight, file-based, and sufficient for a single-user academic project.
    - Bundled with Python, minimizing external dependencies and installation burden.
    - Supports SQL and standard integrity constraints (UNIQUE, CHECK, NOT NULL).
- **Pandas for Data Manipulation**
    - De facto standard for tabular data handling in Python.
    - Facilitates CSV and RSS ingestion, transformation, and conversion to/from SQL.
- **Pandera for Data Quality**
    - Allows declarative schema definitions similar to database DDL but at the DataFrame level.
    - Supports *lazy* validation to collect all errors in one pass, improving observability and debugging.
    - Keeps validation logic close to the data model and independent from ingestion code.
- **Processed Flag Pattern for Idempotence**
    - The boolean `processed` field in staging tables allows incremental and idempotent ETL runs:
        - only records with `processed = false` are picked up;
        - after successful loading, records are marked as processed;
        - quarantined records remain unprocessed for later inspection or correction.
    - This pattern simulates a simple form of change data capture and simplifies error recovery.
- **Quarantine Table for Failed Records**
    - Instead of discarding invalid data, the pipeline captures all failures in `dq_quarantine` with detailed error metadata.
    - Enables reproducible analyses of DQ issues and supports potential remediation workflows.
- **Modularity for Future Extensions**
    - LLM and scraping are separate modules, allowing the instructor or future developers to run the core ETL without external services but still providing a clear path for enrichment stages.

---

7. Conclusions, Improvements, and Future Developments

7.1 Conclusions

- The project successfully implements a **multi-source ETL pipeline** that consolidates GeeksforGeeks and Medium article metadata into a unified dimensional schema.
- **Data quality** is treated as a first-class concern, with explicit Pandera schemas, lazy validation, and a quarantine mechanism that prevents dirty data from polluting downstream tables.
- The architectural design is **modular and reproducible**:
  - fully Python-based;
  - minimal external dependencies;
  - clear orchestration through `main.py` and explicit database schemas.

7.2 Limitations

- The ETL is currently **batch-oriented** and run from scratch (the database is deleted at the start of `main.py`), which simplifies reproducibility but does not reflect incremental production workloads.
- Data quality metrics (e.g., number of quarantined records by error type, completeness scores) are not yet exposed through a dedicated reporting or dashboard layer; they must be derived manually via SQL queries or Python.
- LLM-based enrichment and full-text scraping are present as code but not integrated into the main pipeline, mainly to keep the project easily runnable without external services.

7.3 Possible Improvements and Future Developments

1. **Incremental ETL without Full Reset**
   - Remove the initial `delete_database` in `main.py` for non-demo runs.
   - Use the `processed` flag to support daily or hourly incremental loads.
   - Add scheduling (e.g., cron) and logging to text files instead of only console output.
2. **Content Enrichment and LLM-based Scoring**
   - Connect `scraping.fetch_medium_article` and `llm.analyze_article` to `dim_articles`:
     - store full article text in a new table (e.g., `article_content`)
     - store summary and utility score in `dim_articles` or a related `article_enrichment` table.
   - Design research questions about correlation between LLM-based scores and native engagement metrics (if available).
3. **Migration to a More Scalable DBMS**
   - Replace SQLite with PostgreSQL to support higher concurrency, larger volumes, and more complex indexing strategies.
   - Adapt `sqlite_db.py` to use SQLAlchemy for DB-agnostic code.
4. **Testing**
   - Implement unit tests for:
     - `transform_gfg`, `transform_medium`
     - Pandera schemas and `extract_failed_records_general`
     - `staging_to_dim_articles_*`.
   - Add integration tests that run the entire pipeline on a small sample.

5. **User-facing Interfaces**
    - Implement a small dashboard (e.g., Streamlit) to:
        - browse `dim_articles`
        - inspect quarantine records
        - visualize basic statistics (articles per source, per category, over time).

---

8. Reproducibility Notes

- All paths used in the code assume the project root layout presented in the *operational guide*.