Master in Artificial Intelligence

Project Report

Dimensional Modelling and ETL using MongoDB

Course: Data Engineering

Academic Year: 2025 – 2026

Students:

Gianluca Lascaro
Filippo Morbidelli
Elio Maria Trincia

GitHub Repository:

https://github.com/byluca/ct-medical-images

Report for the data engineering project (lab practice 3)

Contents

1 Introduction

This work was done aiming to apply the dimensional model in order to analyze medical images. Starting from a Kaggle dataset of DICOM files, an ETL pipeline was designed and implemented in Python. The pipeline extracts the relevant metadata from every DICOM file, transforms (cleans, normalizes, generates surrogate keys) it, and then loads it into a NoSQL data warehouse in MongoDB, in a star schema. Another topic within this scope is the conversion of DICOM images to the more accessible file format, the JPEG format, for easier analysis and viewing.

How to Run

This section describes the setup and execution of the whole ETL pipelines as stated by the guide.

- **Prerequisites:** Python, MongoDB Community Edition (running on `localhost:27017`), and MongoDB Compass.
- **Folder Structure:** The project needs a folder `data/dicom_dir/` for the input `.dcm` files and builds a folder `data/jpeg_images/` to store the converted image output.
- **Install Dependencies:** Install dependencies by running `pip install -r requirements.txt`. The major libraries used are `pydicom`, `pymongo` and `Pillow`.
- **Test Connection:** Run `python test_connection.py` to check the connection to the MongoDB server,the script sends a ping command to confirm.
- **Run Pipeline:** Run the main script `python main_etl.py`,it will process all found `.dcm` files, populate the `dicom_dw` database in MongoDB and generate the JPEG files.

2 Methodology and Implementation Choices

The pipeline was developed based on the star schema model and implemented with specific design decisions to ensure both data integrity and efficiency.

2.1 Data Model: Star Schema

The project follows the star schema model described in the guide, this structure is particularly suitable for analytical purposes because it clearly separates data into two main components:

- **Dimension Tables (Dimensions):** Collections that store descriptive and contextual information, such as `dim_patient` and `dim_station`.

- **Fact Tables (Facts):** A central collection, `fact_study`, that contains quantitative measures and the surrogate keys linking it to the dimensions.

This structure illustrated in Figure 1 simplifies analytical queries and makes data exploration more efficient.

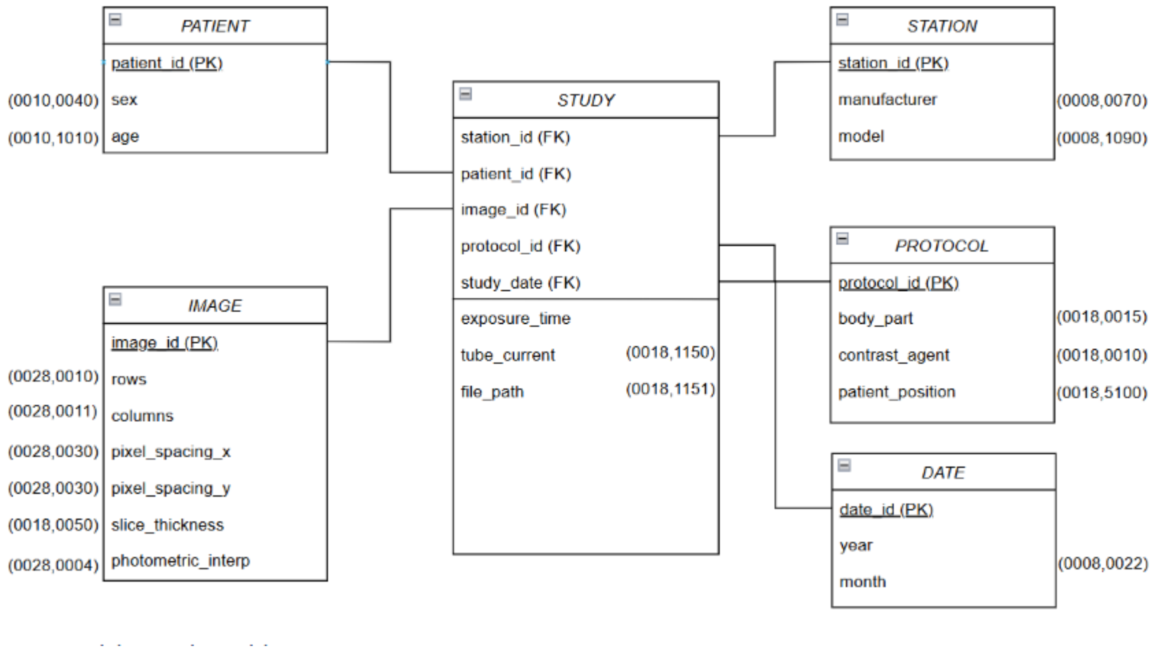


Figure 1: Star schema of the DICOM Data Warehouse (from lab guide).

2.2 Data Processing: ETL Pipeline

The core of this project is the data pipeline implemented in main-etl.py, which orchestrates the entire process of reading, transforming, and loading the data through a robust file-by-file processing loop designed to isolate failures so that if a single DICOM file is corrupt or missing critical data the pipeline logs the error and continues to the next file, ensuring that one bad file does not halt the entire batch, while the transformation logic is encapsulated in a separate etl-utils.py module to promote code reusability and maintainability following the standard ETL workflow .

2.2.1 Extract

The extraction process starts in `main_etl.py`. We first define where the data lives using a global constant `DATA_DIR` which points to the relative path `"data/dicom_dir/"`. Sticking to a relative path was a good design choice because it ensures the pipeline is portable and adheres to the project guideline of avoiding absolute paths.

To find the actual data, the `glob` library scans this directory and simply returns a list of all files ending in `*.dcm`. With this file list in hand, the main loop begins processing them one by one.

A major focus for this phase was robustness, so we designed the logic to be resilient at two different levels:

- **File-Level Resilience:** We anticipated that some files might be completely corrupt. To handle this, the entire processing logic for each single file is wrapped inside a `try...except Exception as e` block. It means that if `pydicom.dcmread` fails on a bad file or any other unexpected error occurs the pipeline doesn't crash. It simply logs the error for that file and continues to the next one allowing the job to run unsupervised over the entire dataset.
- **Tag-Level Resilience:** Even if a file opens correctly it might be missing specific metadata tagsm, to prevent this from halting the pipeline we implemented a custom helper function

`get_safe_value`. This function acts as a simple safety layer when we ask for a tag, it attempts to read it and if the tag is missing (`data_element is None`) it just returns a default value (like "Unknown") instead of raising a program-stopping error.

2.2.2 Transform

This is the most important phase, where the raw extracted metadata is conformed to the business logic of the data warehouse. This process involves generating keys, normalizing values, and converting image formats.

Design Choice: Surrogate Keys A core principle of this data warehouse is the use of surrogate keys (SK) over natural keys, we chose not to use the DICOM `PatientID` as the primary key for the patient dimension because natural keys from source systems can be unreliable; they might be changed, reused, or contain duplicates. Surrogate keys are stable, internally managed identifiers that we control ensuring the long term integrity of the dimensional model. Our implementation, the `surrogate_key` function in `etl_utils.py` creates a deterministic MD5 hash from a dictionary of values. To guarantee determinism it first sorts the dictionary by key before serializing it to a JSON string, this ensures that the same set of attributes will always produce the same surrogate key.

Design Choice: Dimension Management To prevent data duplication in the dimension tables we implemented a "find or create" logic as required by the lab guide. This was necessary because the same entity will appear in thousands of different DICOM files but it must exist as only a single unique document in its respective dimension collection. Our `get_or_create` function orchestrates this: it first calls `surrogate_key` to get the entity's deterministic key. It then performs an efficient `collection.find_one()` query to see if a document with that key already exists. If the document is found the function immediately returns the existing key and if it isn't found does the function create the new document insert it with `collection.insert_one()` and return the new key. Back in `main_etl.py`, this returned SK is captured and stored for the final fact table load.

Data Cleaning and Normalization Functions A list of specific helper functions was developed in `etl_utils.py` to enforce the data quality rules defined in the lab guide:

- `format_age`: This function processes the tag, it handles the 'Y' suffix in strings like '061Y' by stripping it and converting the value to an integer. This transformation is important because it changes the data type from a string to a number enabling numeric analysis.
- `normalize_contrast_agent`: This function standardizes the tag, the logic is specifically designed to catch various "empty" states: like `None`, empty strings or strings with a length of 1 or less.
- `normalize_pixel_spacing`: This function implements a cardinality reduction technique, the tag provides high-precision float values to make grouping and analysis possible. The `normalize_pixel_spacing` function uses `numpy` to find the closest value in a predefined list of "bins".`main_etl.py` also correctly handles that this DICOM tag is a list, extracting the Y and X spacing from their respective list indices.
- **Date Transformation:** Done inline in `main_etl.py`, the Acquisition Date string is parsed into a Python `datetime` object allowing for the easy extraction of `year` and `month` for the `dim_date` dimension.

Image Conversion The `dicom_to_jpeg` function processes the raw pixel data into a standardized preview image,this step is necessary because DICOM pixel data is not directly viewable in most applications. Converting to a 256x256 JPEG provides a uniform, lightweight, and universally supported format suitable for previews. The function itself is more than a simple conversion; it first reads the `pixel_array` and checks if `RescaleSlope` and `RescaleIntercept` tags exist. If they do, it applies them to the pixel data which is a critical step to convert the raw stored values into meaningful units. Only after this step does it normalize the pixels to a 0-255 range. It also includes logic to skip multi-frame images and uses the Pillow library to resize and save the file as grayscale .

2.2.3 Load

The final phase loads the transformed data into the MongoDB data warehouse,this process is intentionally bifurcated to ensure data integrity.

The dimension tables are loaded during the transform step, this is handled implicitly by the `get_or_create` function, which inserts a new dimension record only at the moment a new unseen surrogate key is generated.

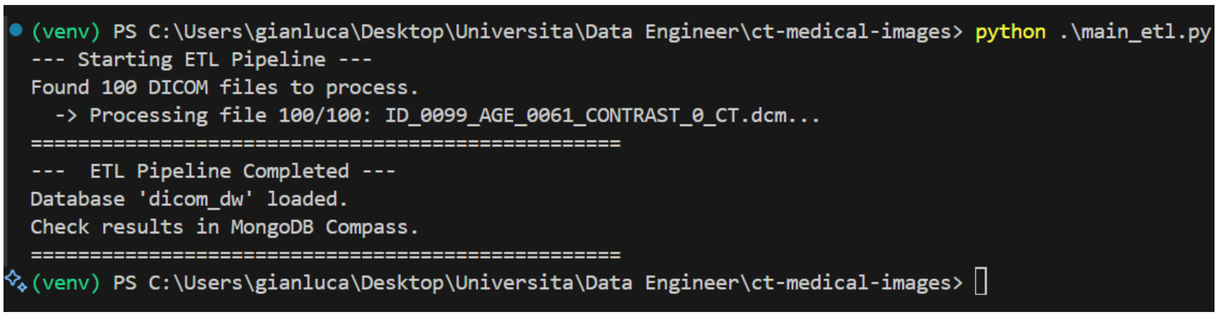
The fact table, however is loaded explicitly at the very end of the main loop in `main_etl.py`. After all five surrogate keys (for patient, station, protocol, image, and date) have been retrieved or created a `fact_study` document is assembled. This document contains only these keys and the relevant measures (like `exposure_time` and `tube_current`).

This fact-loading step is protected by an integrity check, a key design choice in `main_etl.py` is the conditional insertion: the fact document is saved to the database only if the `dicom_to_jpeg` conversion was successful. This prevents the creation of orphaned fact records that would point to a non-existent image file ensuring the atomicity and reliability of each entry in the fact table.

3 Results Analysis

The ETL pipeline was executed as described in the "How to Run" section, Figure 2 shows the terminal output upon completion, confirming that 100 DICOM files were processed and the `dicom_dw` database was successfully loaded.

With the data warehouse populated the database was analyzed using MongoDB Compass. The created collections and several analysis queries are presented below.



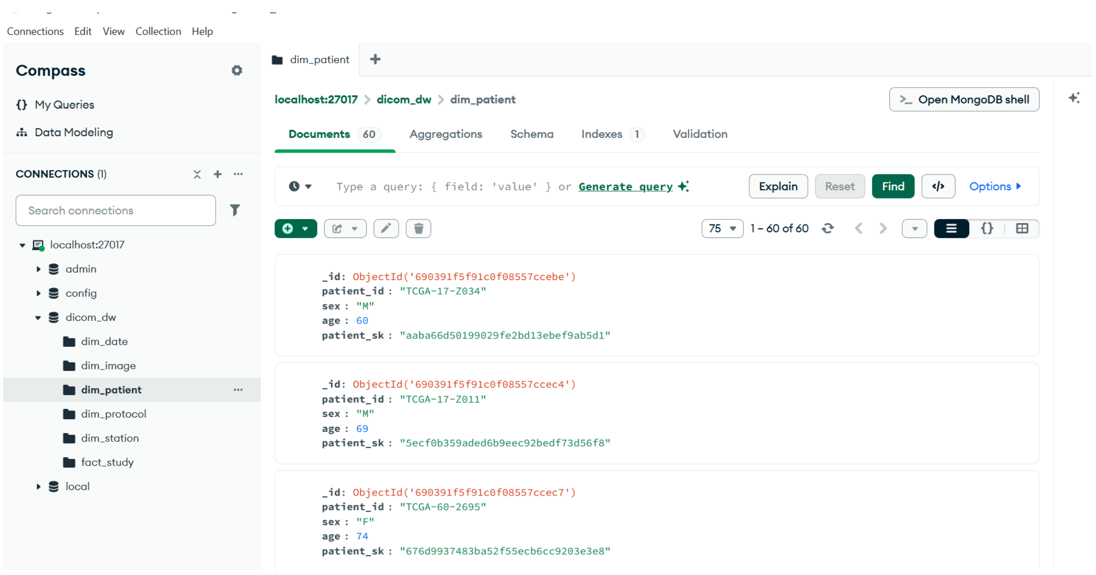
```
(venv) PS C:\Users\gianluca\Desktop\Universita\Data Engineer\ct-medical-images> python .\main_etl.py
--- Starting ETL Pipeline ---
Found 100 DICOM files to process.
  -> Processing file 100/100: ID_0099_AGE_0061_CONTRAST_0_CT.dcm...
==================================================
---  ETL Pipeline Completed ---
Database 'dicom_dw' loaded.
Check results in MongoDB Compass.
==================================================
(venv) PS C:\Users\gianluca\Desktop\Universita\Data Engineer\ct-medical-images> 
```

Figure 2: Terminal output showing the successful completion of the ETL pipeline.

3.1 Visualizing the Collections in MongoDB

As a result of the ETL pipeline, 6 collections were created in the `dicom_dw` database.

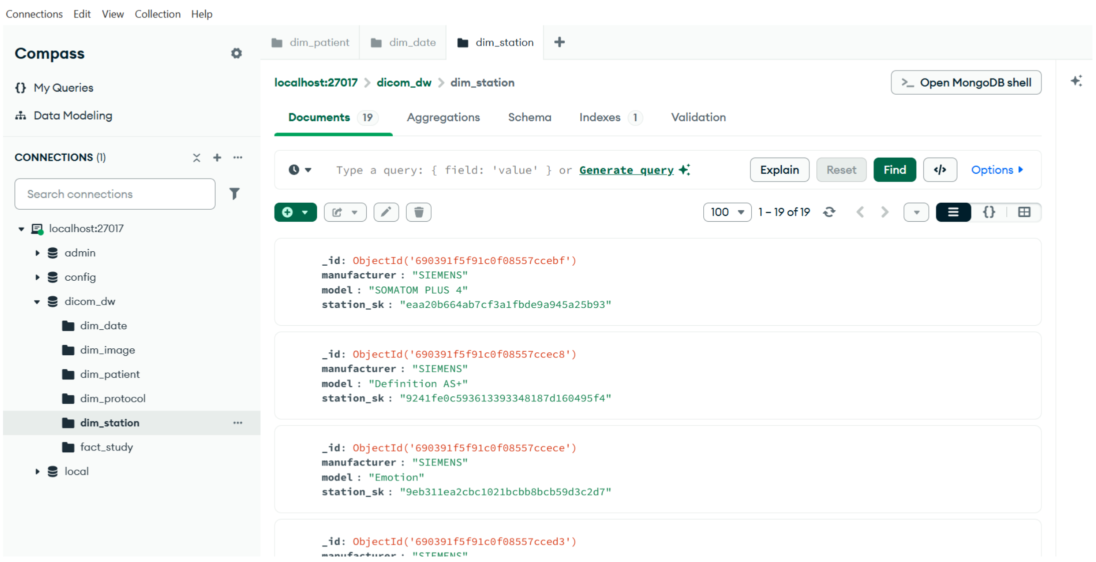
- **dim_patient:** Contains patient demographic attributes (sex, age), an example is visible in Figure 3.



```
_id: ObjectId('690391f5f91c0f08557ccebe')
patient_id : "TCGA-17-Z034"
sex : "M"
age : 60
patient_sk : "aaba66d50199029fe2bd13ebef9ab5d1"

_id: ObjectId('690391f5f91c0f08557ccec4')
patient_id : "TCGA-17-Z011"
sex : "M"
age : 69
patient_sk : "5ecf0b359aded6b9eec92bedf73d56f8"

_id: ObjectId('690391f5f91c0f08557ccec7')
patient_id : "TCGA-60-2695"
sex : "F"
age : 74
patient_sk : "676d9937483ba52f55ecb6cc9203e3e8"
```

Figure 3: Screenshot of `dim_patient` in MongoDB Compass.

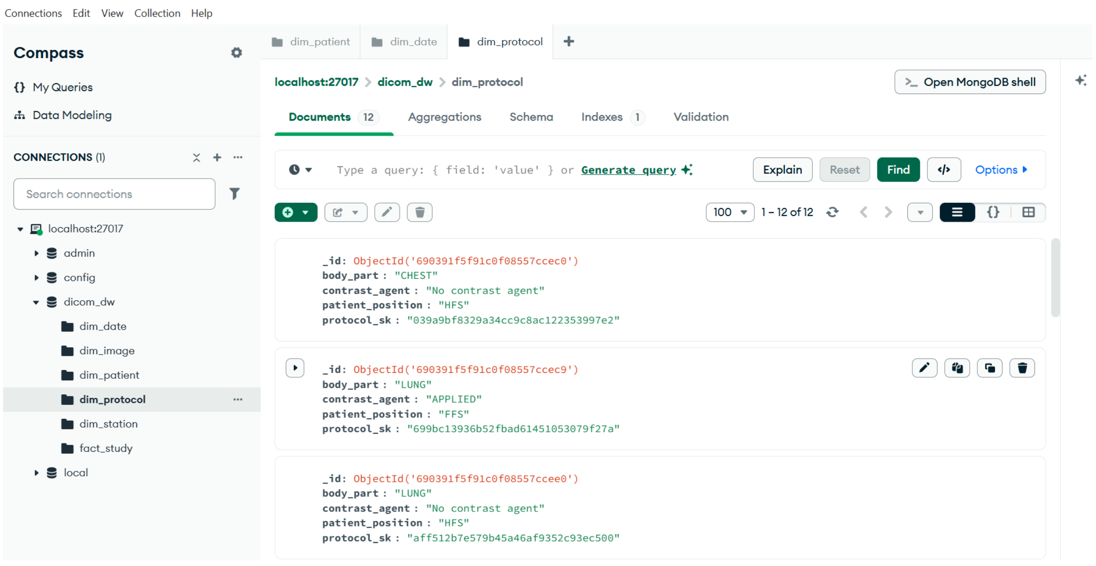
- **dim_station:** Contains information about the machinery (manufacturer, model), an example is visible in Figure 4.



```
_id: ObjectId('690391f5f91c0f08557ccebf')
manufacturer : "SIEMENS"
model : "SOMATOM PLUS 4"
station_sk : "eaa20b664ab7cf3a1fbde9a945a25b93"

_id: ObjectId('690391f5f91c0f08557ccec8')
manufacturer : "SIEMENS"
model : "Definition AS+"
station_sk : "9241fe0c593613393348187d160495f4"

_id: ObjectId('690391f5f91c0f08557ccece')
manufacturer : "SIEMENS"
model : "Emotion"
station_sk : "9eb311ea2cbc1021bcbb8bcb59d3c2d7"

_id: ObjectId('690391f5f91c0f08557cced3')
manufacturer : "SIEMENS"
```

Figure 4: Screenshot of `dim_station` in MongoDB Compass.

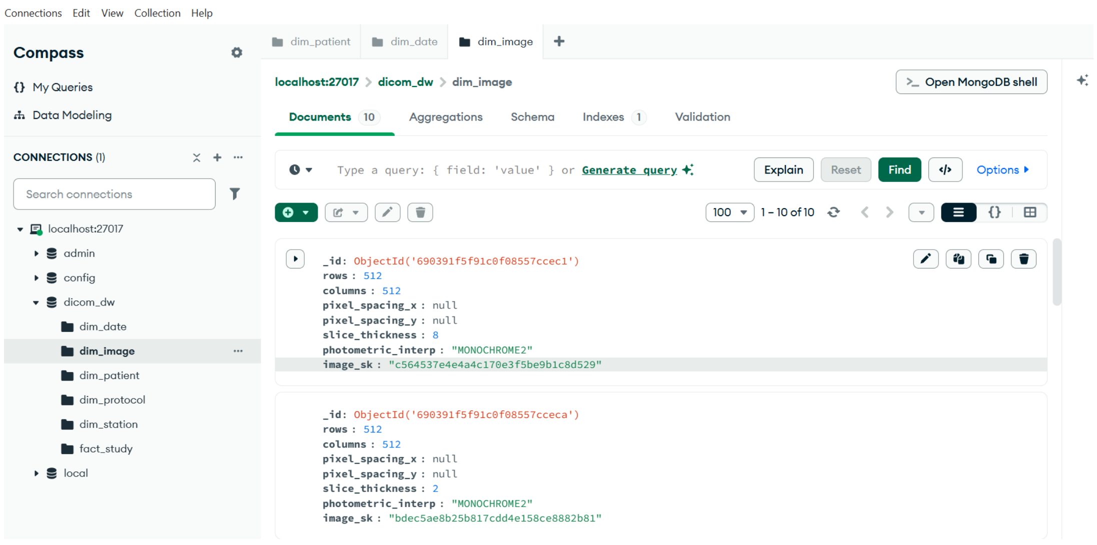
- **dim_protocol:** Describes the acquisition protocol (body part, contrast agent), an example is visible in Figure 5.



```
_id: ObjectId('690391f5f91c0f08557ccec0')
body_part : "CHEST"
contrast_agent : "No contrast agent"
patient_position : "HFS"
protocol_sk : "039a9bf8329a34cc9c8ac122353997e2"

_id: ObjectId('690391f5f91c0f08557ccec9')
body_part : "LUNG"
contrast_agent : "APPLIED"
patient_position : "FFS"
protocol_sk : "699bc13936b52fbad61451053079f27a"

_id: ObjectId('690391f5f91c0f08557ccee0')
body_part : "LUNG"
contrast_agent : "No contrast agent"
patient_position : "HFS"
protocol_sk : "aff512b7e579b45a46af9352c93ec500"
```

Figure 5: Screenshot of `dim_protocol` in MongoDB Compass.

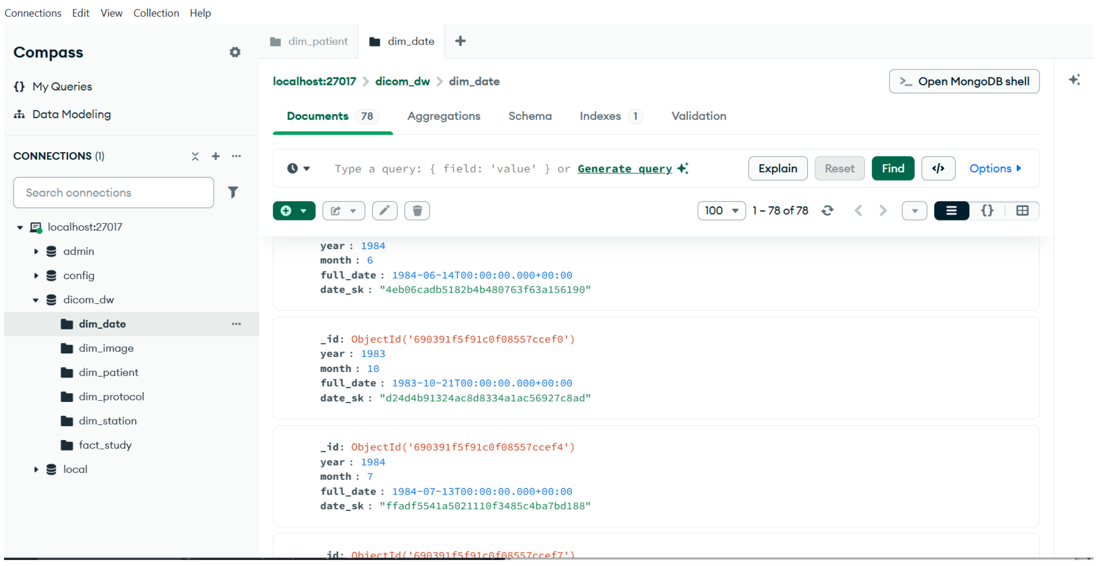
- **dim_image:** Contains technical image metadata (resolution, slice thickness), an example is visible in Figure 6.



```
_id: ObjectId('690391f5f91c0f08557ccec1')
rows : 512
columns : 512
pixel_spacing_x : null
pixel_spacing_y : null
slice_thickness : 8
photometric_interp : "MONOCHROME2"
image_sk : "c564537e4e4a4c170e3f5be9b1c8d529"

_id: ObjectId('690391f5f91c0f08557cceca')
rows : 512
columns : 512
pixel_spacing_x : null
pixel_spacing_y : null
slice_thickness : 2
photometric_interp : "MONOCHROME2"
image_sk : "bdec5ae8b25b817cdd4e158ce8882b81"
```

Figure 6: Screenshot of `dim_image` in MongoDB Compass.

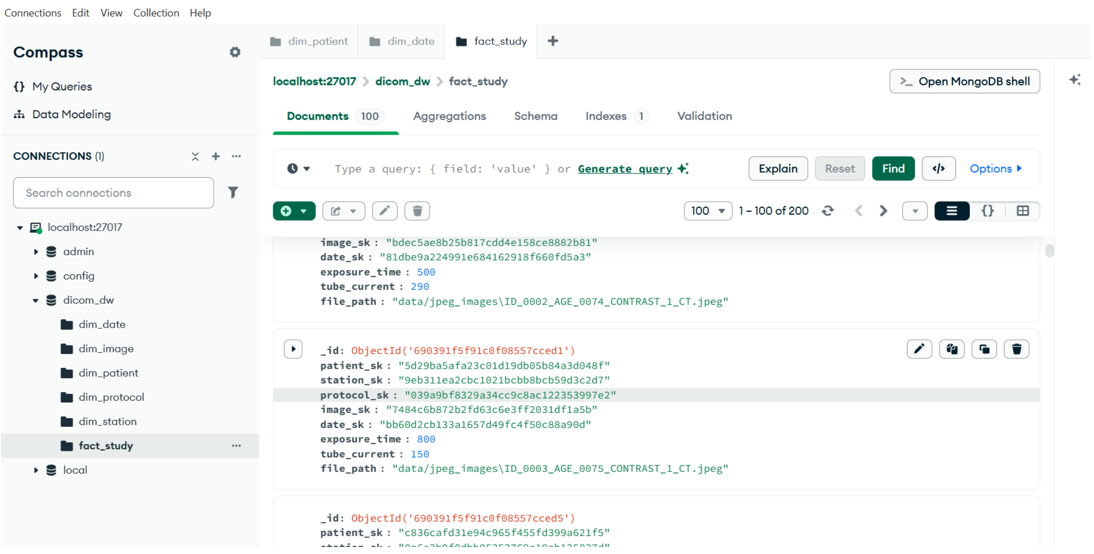
- **dim_date:** Time dimension for grouping studies by date (year, month), an example is visible in Figure 7.



```
year : 1984
month : 6
full_date : 1984-06-14T00:00:00.000+00:00
date_sk : "4eb06cadb5182b4b480763f63a156190"

_id: ObjectId('690391f5f91c0f08557ccef0')
year : 1983
month : 10
full_date : 1983-10-21T00:00:00.000+00:00
date_sk : "d24d4b91324ac8d8334a1ac56927c8ad"

_id: ObjectId('690391f5f91c0f08557ccef4')
year : 1984
month : 7
full_date : 1984-07-13T00:00:00.000+00:00
date_sk : "ffadf5541a5021110f3485c4ba7bd188"

_id: ObjectId('690391f5f91c0f08557ccef7')
```

Figure 7: Screenshot of `dim_date` in MongoDB Compass.

- **fact_study:** The central fact table, which links all dimensions and stores the facts, an example is visible in Figure 8.



```
image_sk : "bdec5ae8b25b817cdd4e158ce8882b81"
date_sk : "81dbe9a224991e684162918f660fd5a3"
exposure_time : 500
tube_current : 290
file_path : "data/jpeg_images\ID_0002_AGE_0074_CONTRAST_1_CT.jpeg"

_id: ObjectId('690391f5f91c0f08557cced1')
patient_sk : "5d29ba5afa23c01d19db05b84a3d048f"
station_sk : "9eb311ea2cbc1021bcbb8bcb59d3c2d7"
protocol_sk : "039a9bf8329a34cc9c8ac122353997e2"
image_sk : "7484c6b872b2fd63c6e3ff2031df1a5b"
date_sk : "bb60d2cb133a1657d49fc4f50c88a90d"
exposure_time : 800
tube_current : 150
file_path : "data/jpeg_images\ID_0003_AGE_0075_CONTRAST_1_CT.jpeg"

_id: ObjectId('690391f5f91c0f08557cced5')
patient_sk : "c836cafd31e94c965f455fd399a621f5"
```

Figure 8: Screenshot of `fact_study` in MongoDB Compass.

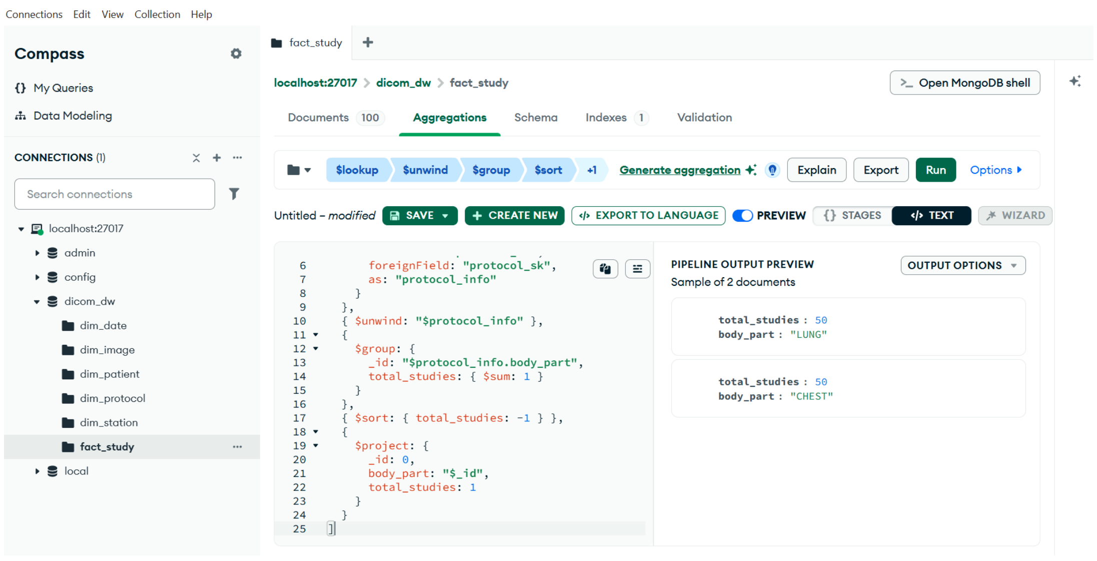
3.2 Data Warehouse Queries

By using the dimensional model and MongoDB's Aggregation Framework we can answer analytical questions.

3.2.1 Analysis 1: Scan Count by Body Part

Question: How many scans were performed for each body part?
This query joins `fact_study` with `dim_protocol` using the `$lookup` operator. Next it groups (`$group`) the results by `protocol.body_part` and counts (`$sum: 1`) the items in each group

and finally it sorts (`$sort`) the results.



```
      foreignField: "protocol_sk",
      as: "protocol_info"
    }
  },
  { $unwind: "$protocol_info" },
  {
    $group: {
      _id: "$protocol_info.body_part",
      total_studies: { $sum: 1 }
    }
  },
  { $sort: { total_studies: -1 } },
  {
    $project: {
      _id: 0,
      body_part: "$_id",
      total_studies: 1
    }
  }
]
```

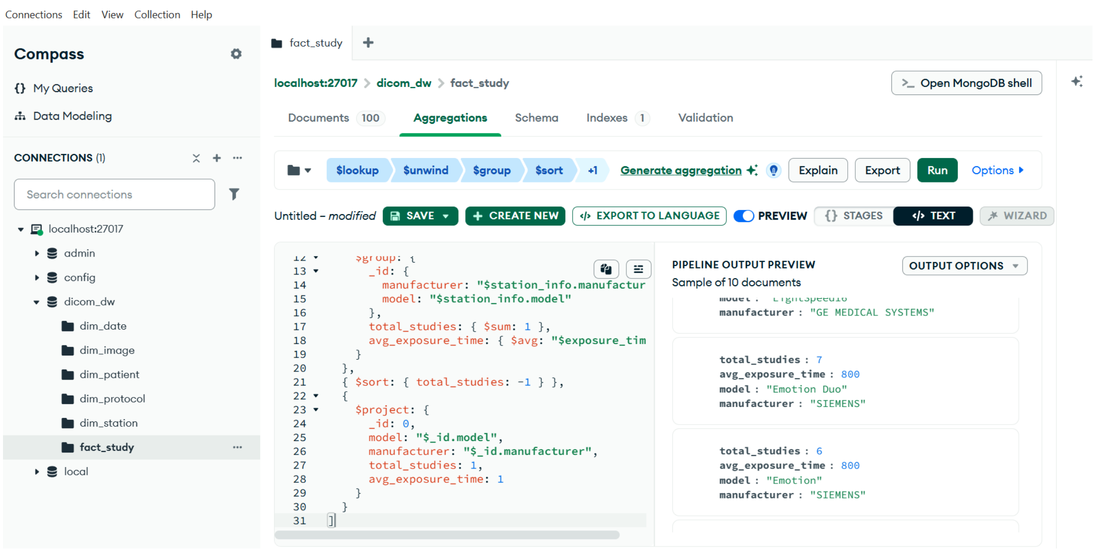
```
total_studies : 50
body_part : "LUNG"

total_studies : 50
body_part : "CHEST"
```

Figure 9: Result of Query 1 in MongoDB Compass (Count by body part).

Discussion: This query is informative as it confirms two key facts about the dataset: firstly, the `$group` stage proves that the only two distinct values present in the Body Part tag (0018, 0015) are "CHEST" and "LUNG". Second, the counts reveal that the dataset is perfectly balanced with 50 studies for LUNG and 50 studies for CHEST, summing to the expected total of 100 unique studies. This confirms a 100 percent success rate in processing the 100 available files and suggests the source `dicom_dir` is a small, intentionally curated dataset.

3.2.2 Analysis 2: Average Exposure Time by Manufacturer

Question: What is the average exposure time for each equipment model?
The query joins `fact_study` with `dim_station` (`$lookup`), it then groups (`$group`) by the unique combination of `station.manufacturer` and `station.model`. It calculates the average (`$avg`) of the `exposure_time` field and counts the total studies (`$sum: 1`) for each specific machine. This allows for a granular comparison of equipment performance.



```
    $group: {
      _id: {
        manufacturer: "$station_info.manufactur
        model: "$station_info.model"
      },
      total_studies: { $sum: 1 },
      avg_exposure_time: { $avg: "$exposure_tim
    }
  },
  { $sort: { total_studies: -1 } },
  {
    $project: {
      _id: 0,
      model: "$_id.model",
      manufacturer: "$_id.manufacturer",
      total_studies: 1,
      avg_exposure_time: 1
    }
  }
]
```

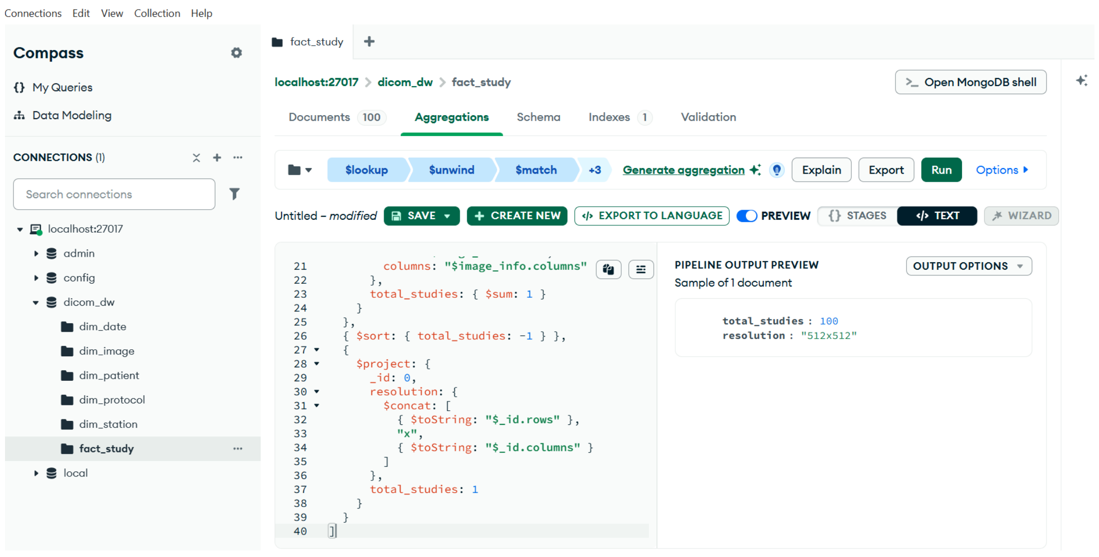
```
manufacturer : "GE MEDICAL SYSTEMS"

total_studies : 7
avg_exposure_time : 800
model : "Emotion Duo"
manufacturer : "SIEMENS"

total_studies : 6
avg_exposure_time : 800
model : "Emotion"
manufacturer : "SIEMENS"
```

Figure 10: Result of Query 2 in MongoDB Compass (Average exposure time by manufacturer).

Discussion: This analysis reveals specific machine models used in the dataset. For instance, the "SIEMENS" manufacturer is represented by models like "Emotion" (6 studies, avg time 800) and "Emotion Duo" (6 studies, avg time 800). The query allows for direct performance comparison: the "GE MEDICAL SYSTEMS" "LightSpeed" (7 studies) has an average exposure time of 800. This grouping by both manufacturer and model provides a high level of detail for performance metrics.

3.2.3 Analysis 3: Most Common Image Resolutions

Question: What are the most common image resolutions in the dataset and how many studies use each one?
This query analyzes the technical properties of the images by joining the `fact_study` collection with the `dim_image` dimension using `$lookup`. It then flattens the results with `$unwind` and uses `$match` to filter out any records with missing `rows` or `columns` data, ensuring data consistency. Next the pipeline groups the data by image dimensions with `$group` and counts the number of studies for each unique resolution using `$sum`. Finally, it sorts (`$sort`) the results and reformats the output with `$project` to clearly display the resolution as a single field (like "512x512").



```
          columns: "$image_info.columns"
        },
        total_studies: { $sum: 1 }
      }
    },
    { $sort: { total_studies: -1 } },
    {
      $project: {
        _id: 0,
        resolution: {
          $concat: [
            { $toString: "$_id.rows" },
            "x",
            { $toString: "$_id.columns" }
          ]
        },
        total_studies: 1
      }
    }
]
```

```
total_studies : 100
resolution : "512x512"
```

Figure 11: Result of Query 3 in MongoDB Compass (Grouping studies by image resolution).

Discussion: The query's output is definitive: 100 percent of the 100 studies in the database use a 512x512 resolution. This confirms that the entire dataset is technically consistent aligning perfectly with the lab guide's description of CT slices which noted that "Images in this format are typically 512x512 pixels".

4 Conclusion

The whole project demonstrated a successful end-to-end implementation of an ETL pipeline for the medical imaging data warehousing, starting from a collection of raw DICOM files. We were able to build a Python-based process that reliably extracts, transforms, and loads the data into a star schema hosted on MongoDB. The design proved robust and provided the functionality required. Key architectural decisions—such as putting in file-level exception handling and tag-level safe guards—were crucial in making sure the pipeline ran to completion over a dataset of inconsistent files. Also important was the use of deterministic surrogate keys and a "get or create" function in maintaining dimension table integrity and preventing data duplication.

Once populated, the dimensional model was shown to be highly effective for analysis. Using MongoDB's aggregation framework, we were able to join the fact table with multiple dimensions to answer complex questions. The successful implementation of idempotency in the Load step ensured data reliability by preventing the duplication of fact records, a bug discovered and corrected during initial testing. The final confirmation of the source dataset properties includes its composition (50 "CHEST" and 50 "LUNG" studies, totaling 100), its technical consistency in terms of resolution (100 percent 512x512), and the performance metrics of its various equipment, such as "SIEMENS" and "GE." This project serves as a practical validation of how dimensional modeling can be applied to NoSQL databases to support powerful business intelligence and data analysis.