# [DM] Project - Implementing a dataset on mongodb and then show relevant usage

Alessandro Pisent
Matricola : 2085678

**Abstract**

This project details the implementation of the IMDb dataset, originally in a relational format, into a document-based database (MongoDB). The data processing workflow using PySpark is explained, along with optimized queries to extract valuable insights from the dataset.

## Contents

# 1 Implementation and importing of the dataset

In this part of the project we will process and enrich IMDb datasets using PySpark, and then store the transformed data into MongoDB for further analysis or application use. The pipeline is modular and scalable, making use of Spark's distributed processing capabilities and MongoDB's flexible storage format.

The core objective of this part of the project is to:

- **Ingest and process multiple IMDb datasets** such as title metadata, people details, episodes, cast and crew, and ratings.
- **Perform data enrichment and aggregation** to generate structured collections for movies, TV series (with embedded episodes), shorts, and people.
- **Persist the transformed data** in a MongoDB database, which can be later used for querying, analytics, or integration with other services.

## 1.1 Data Sources and Schemas

The project uses several TSV datasets from IMDb, each with a clearly defined schema:

- **title.basics.tsv** - Contains core metadata about titles (e.g., movies, TV series, shorts).
  **Key fields:** tconst, titleType, primaryTitle, startYear, genres, etc.
- **name.basics.tsv** - Contains information about people involved in titles.

  **Key fields:** nconst, primaryName, birthYear, primaryProfession, knownForTitles, etc.
- **title.episode.tsv** - Contains metadata about TV series episodes.

  **Key fields:** tconst, parentTconst, seasonNumber, episodeNumber.
- **title.principals.tsv** - Contains details on cast and crew for each title.

  **Key fields:** tconst, nconst, category, job, characters, etc.
- **title.ratings.tsv** - Contains ratings information for each title.

  **Key fields:** tconst, averageRating, numVotes.

Each dataset is read into Spark DataFrames using an explicit schema to ensure data integrity and consistency during processing.

## 1.2 System Architecture

- **Spark Configuration:** a Spark session with custom configurations such as increased memory allocation and an elevated number of shuffle partitions. This ensures that the processing can scale efficiently.
- **MongoDB Connector:** The session is configured with the MongoDB Spark connector to facilitate seamless writing of the processed data into MongoDB.

## 1.3 Data Processing Workflow

### 1.3.1 List of people per each title

A join operation is performed to link people with the titles they are associated with, followed by aggregation:

- **Join Operation:** The `process_principals` function performs a broadcast join between the cast/crew data (`title_principals`) and people details (name_basics) based on the unique identifier `nconst`.
- **Aggregation:**After joining, the data is aggregated per title (`tconst`), with a sorted list of associated people (actors, directors, etc.) encapsulated in a nested structure.

```
def process_principals(title_principals, name_basics):
    """
    Process title principals data by joining it with name_basics to enrich
    with people's details and aggregating them per title.

    Args:
        title_principals (DataFrame): DataFrame containing title principals.
        name_basics (DataFrame): DataFrame containing people's information.

    Returns:
        DataFrame: A DataFrame with aggregated principal details per title.
    """

    logging.info("Processing principals and joining with people data ...")
    principals_people = title_principals.join(
        broadcast(name_basics), on="nconst", how="left"
    ).select(
        "tconst", "ordering", "nconst",
        col("primaryName").alias("name"),
        "category", "job", "characters",
        "birthYear", "deathYear", "primaryProfession"
    )

    # Debug: inspect the joined DataFrame
    logging.info("Schema of principals_people:")
    principals_people.printSchema()
    #principals_people.show(5, truncate=False)

    aggregated_principals = principals_people.groupBy("tconst") \
        .agg(sort_array(collect_list(
            struct("ordering", "name", "category", "job", "characters",
                   "nconst", "birthYear", "deathYear", "primaryProfession")
        ), asc=True).alias("people"))

    # Debug: inspect the aggregated DataFrame
    logging.info("Schema of aggregated_principals:")
    aggregated_principals.printSchema()
    #aggregated_principals.show(5, truncate=False)

    return aggregated_principals
```

### 1.3.2 Categorize the type of titles

Titles are categorized into separate collections, and relevant information (like rating) is added:

- **Movies** – Filtered by `titleType == "movie"`, with relevant fields extracted.
- **TV Series** – Processed similarly, later enriched with episodes.
- **Shorts** – Titles of type `short` undergo similar processing as movies.

```
def prepare_title_collections(title_basics, aggregated_principals, title_ratings):
    """
    Prepare collections for Movies, TV Series, and Shorts by:
      - Filtering by title type.
      - Renaming columns to match the target schema.
      - Joining aggregated principals and ratings.

    Args:
        title_basics (DataFrame): DataFrame containing basic title details.
        aggregated_principals (DataFrame): DataFrame with aggregated principal data.
        title_ratings (DataFrame): DataFrame containing title ratings.

    Returns:
        tuple: A tuple containing:
            - movies_total (DataFrame): Processed movies collection.
            - tv_series_total (DataFrame): Processed TV series collection.
            - shorts_total (DataFrame): Processed shorts collection.
```

```
    """
    logging.info("Preparing title collections ...")

    # Movies: they only have start year
    movies_basic = title_basics.filter(col("titleType") == "movie") \
        .withColumnRenamed("startYear", "year") \
        .withColumnRenamed("primaryTitle", "title") \
        .drop("titleType")\
        .drop("endYear")

    # TV Series
    tv_series_basic = title_basics.filter(col("titleType") == "tvSeries") \
        .withColumnRenamed("primaryTitle", "title") \
        .drop("titleType")

    # Shorts: they only have start year
    shorts_basic = title_basics.filter(col("titleType") == "short") \
        .withColumnRenamed("startYear", "year") \
        .withColumnRenamed("primaryTitle", "title") \
        .drop("titleType")\
        .drop("endYear")

    # Convert genres string to array for each title type
    movies_basic = movies_basic.withColumn("genres", split(col("genres"), ","))
    tv_series_basic = tv_series_basic.withColumn("genres", split(col("genres"), ","))
    shorts_basic = shorts_basic.withColumn("genres", split(col("genres"), ","))

    # Join in aggregated principals (people) for each title.
    movies_with_people = movies_basic.join(aggregated_principals, on="tconst", how="left")
    tv_series_with_people = tv_series_basic.join(aggregated_principals, on="tconst", how="left")
    shorts_with_people = shorts_basic.join(aggregated_principals, on="tconst", how="left")

    # Join ratings and embed them as a nested struct.
    movies_total = movies_with_people.join(title_ratings, on="tconst", how="left") \
        .withColumn("rating", struct(col("numVotes"), col("averageRating"))) \
        .drop("numVotes", "averageRating")

    tv_series_total = tv_series_with_people.join(title_ratings, on="tconst", how="left") \
        .withColumn("rating", struct(col("numVotes"), col("averageRating"))) \
        .drop("numVotes", "averageRating")

    shorts_total = shorts_with_people.join(title_ratings, on="tconst", how="left") \
        .withColumn("rating", struct(col("numVotes"), col("averageRating"))) \
        .drop("numVotes", "averageRating")

    return movies_total, tv_series_total, shorts_total
```

### 1.3.3 Adding to TV Series collection the Episodes

The dataset is further enriched by embedding episodes within TV series:

- **Join:** Episode metadata is linked with title information.
- **Ordering:** Episodes are sorted within their respective TV series.

```
def add_episodes_to_tv_series(tv_series_total, title_episode, title_basics, title_ratings):
    """
    Enrich the TV series collection by adding episode details.

    Steps:
      - Join title_episode with title_basics to fetch episode metadata.
      - Create an ordering structure for episodes (Season, Episode).
      - Group episodes by the parent TV series.

    Args:
        tv_series_total (DataFrame): DataFrame of processed TV series.
        title_episode (DataFrame): DataFrame of episode information.
        title_basics (DataFrame): DataFrame with title metadata.
        title_basics (DataFrame): DataFrame with rating metadata.

    Returns:
        DataFrame: Updated TV series DataFrame with embedded episodes.
    """
    logging.info("Adding episode details to TV Series ...")

    episodes_details = title_episode.join(
        title_basics.select(
            col("tconst").alias("ep_tconst"),
            col("primaryTitle").alias("ep_title"),
            col("isAdult").alias("ep_isAdult"),
            col("startYear").alias("ep_year"),
            col("runtimeMinutes").alias("ep_runtime"),
            "genres"  # include if needed
        ),
        title_episode.tconst == col("ep_tconst"),
        how="left"
    )

    # Build ordering for episodes
    episodes_details = episodes_details.withColumn("ordering",
        struct(col("seasonNumber").alias("Season"), col("episodeNumber").alias("Episode"))
    )

    episodes_with_rating = episodes_details.join(title_ratings, on="tconst", how="left") \
```

```
        .withColumn("rating", struct(col("numVotes"), col("averageRating"))) \
        .drop("numVotes", "averageRating")

    episodes_final = episodes_with_rating.select(
        "parentTconst",
        col("ep_title").alias("Title"),
        col("ep_isAdult").alias("isAdult"),
        col("ep_year").alias("year"),
        col("ep_runtime").alias("runtime"),
        "ordering",
        "rating"
    )

    # Group episodes by the parent TV series and optionally sort them
    episodes_agg = episodes_final.groupBy("parentTconst") \
        .agg(sort_array(collect_list(
            struct("Title", "isAdult", "year", "runtime", "ordering", "rating")
        )).alias("episodes"))

    tv_series_total = tv_series_total.join(
        episodes_agg,
        tv_series_total.tconst == episodes_agg.parentTconst,
        how="left"
    ).drop("parentTconst")

    return tv_series_total
```

### 1.3.4 Processing People Data

The separate `people` collection is added from the dataset, and then some relevant information are added:

- **Data Cleaning:** Splitting multi-valued fields such as `primaryProfession` and `knownForTitles`.
- **Enrichment:** Titles associated with people are joined to fetch relevant metadata.

```
def process_people(name_basics, title_basics):
    """
    Process the people collection by:
      - Splitting primaryProfession and knownForTitles into arrays.
      - Enriching knownForTitles with metadata from title_basics.
      - Aggregating knownForTitles into a structured list.

    Args:
        name_basics (DataFrame): DataFrame containing people metadata.
        title_basics (DataFrame): DataFrame containing title metadata.

    Returns:
        DataFrame: Processed people DataFrame with enriched mainTitles.
    """
    logging.info("Processing people and enriching with main titles ...")

    # Convert primaryProfession and knownForTitles to arrays
    people_df = name_basics \
        .withColumn("primaryProfession", split(col("primaryProfession"), ",")) \
        .withColumn("knownForTitlesArray", split(col("knownForTitles"), ","))

    # Explode knownForTitles to get one row per title
    exploded = people_df.withColumn("explodedTconst", explode("knownForTitlesArray"))
    exploded = exploded.withColumn("explodedTconst", trim(col("explodedTconst")))

    # Prepare a lookup DataFrame from title.basics with tconst, primaryTitle, and startYear
    titles_lookup = title_basics.select(
        col("tconst"),
        col("primaryTitle").alias("title"),
        col("startYear").alias("year")
    )

    # Join exploded people with title details
    exploded = exploded.join(
        titles_lookup,
        exploded.explodedTconst == titles_lookup.tconst,
        how="left"
    )

    # Aggregate main titles for each person as a list of structs
    main_titles = exploded.groupBy("nconst").agg(
        collect_list(
            struct(
                col("explodedTconst").alias("tconst"),
                col("title"),
                col("year")
            )
        ).alias("mainTitles")
    )

    # Join back to the original people_df and drop intermediate columns
    people_final = people_df.drop("knownForTitles", "knownForTitlesArray") \
        .join(main_titles, on="nconst", how="left")

    return people_final
```

## 1.4 Write in MongoDB

- **Generic Write Function:**

  The `write_to_mongodb` function is a utility that writes a given DataFrame to a specified MongoDB collection. It supports both default and repartitioned writes to handle larger datasets effectively.

- **Collections Stored:**

  The final DataFrames are written into separate collections:

  - **movies**
  - **tvSeries**
  - **shorts**
  - **people**

```python
def write_to_mongodb(df, collection_name,
                     repartition=False,
                     dim=10_000,
                     db_name="imdbSpark"):
    """
    Write the given DataFrame to MongoDB.

    Args:
        df (DataFrame): The DataFrame to be written.
        collection_name (str): The target MongoDB collection name.
        repartition (bool, optional): Whether to repartition the data before writing. Defaults to False.
        dim (int, optional): Number of partitions if repartitioning is enabled. Defaults to 10,000.
    """
    logging.info(f"Writing collection '{collection_name}' to MongoDB ...")
    if not repartition:
        df.write.format("mongodb") \
            .mode("overwrite") \
            .option("database", db_name) \
            .option("collection", collection_name) \
            .save()
    else:
        logging.info(f"Writing on {dim} repartitions")
        df.repartition(dim).write.format("mongodb") \
            .mode("overwrite") \
            .option("database", db_name) \
            .option("collection", collection_name) \
            .save()
```

# 2 MongoDB schema

Here is the main schema for the collections of documents in the mongoDB database.

## 2.1 Collection: tvSeries

```
- _id: ObjectId
- tconst: str
- title: str
- originalTitle: str
- isAdult: int (0 - 1)
- startYear: int
- endYear: int
- runtimeMinutes: int
- genres: list of str
- people: list of documents:
  - name
  - ordering
```

```
  - bithyear
  - deathYear
  - primaryProfession
  - category
  - job
  - characters
- rating: document
  - numVotes: int
  - averageRating: float
- episodes: list of documents
  - Title : str
  - isAdult : int (0 - 1)
  - ordering : document:
    - Episode : int
    - Season : int
  - rating: document
    - numVotes: int
    - averageRating: float
  - runtimeMinutes: int
  - year : int
```

## 2.2 Collection: movies

```
- _id: ObjectId
- tconst: str
- title: str
- originalTitle: str
- isAdult: int (0 - 1)
- year: int
- runtimeMinutes: int
- genres: list of str
- people: list of documents:
  - name : str
  - ordering : int
  - bithyear : int
  - deathYear : int
  - primaryProfession : str
  - category : str
  - job : str
  - characters : list of str
- rating: document
  - numVotes: int
  - averageRating: float
```

## 2.3 Collection: shorts

```
- _id: ObjectId
- tconst: str
- title: str
```

```
- originalTitle: str
- isAdult: int (0 - 1)
- year: int
- runtimeMinutes: int
- genres: list of str
- people: list of documents:
  - name
  - ordering
  - bithyear
  - deathYear
  - primaryProfession
  - category
  - job
  - characters
- rating: document
  - numVotes: int
  - averageRating: float
```

### 2.4 Collection: people

```
- _id: ObjectId
- nconst: str
- primaryName: str
- birthYear: int
- deathYear: int
- primaryProfession: (list) str
- mainTitles: (list) document
  - tconst: str
  - title: str
  - year: int
```

## 3 MongoDB usage

### 3.1 View : allTitles

We can create an all-title view that we can then utilize.

```
// Select the database to use.
use('imdb');

// First, create a view that unions movies, tvSeries, and shorts
db.createView("titles", "tvSeries", [
    {
      $addFields:{
          titleType: { $literal: "tvSeries" }
      }
    },
    // Union with shorts
    {
      $unionWith: {
        coll: "shorts",
        pipeline: [
          {
            $addFields: { startYear: "$year",// Rename "year" to "startYear"
              titleType: { $literal: "short" },
            }
          },
          {
            $project: {

              year: 0

            }
          }
        ]
      }
    },
    // union movies
    {
      $unionWith: {
        coll: "movies",
        pipeline: [
          {
            $addFields: {
              startYear: "$year",
              titleType: { $literal: "movies" },
             } // Rename "year" to "startYear"
          },
          {
            $project: {

              year: 0
             } // Remove old "year" field
          }
        ]
      }
    },
  ])
```

## 3.2 Query: avg Rating per Genre

We can query the movies collection (or any other collection, since they all have genres) to get the average raiting of each genres

```
// Select the database to use.
use('imdb');

db.movies.aggregate([
    // Unwind the genres array so that each genre is processed individually
    { $unwind: "$genres" },

    // Group by each genre and calculate the average rating and total count
    { $group: {
        _id: "$genres",
        avgRating: { $avg: "$rating.averageRating" },
        totalMovies: { $sum: 1 }
    }},

    // Sort genres by average rating in descending order
    { $sort: { avgRating: -1 } }
  ])
```

## 3.3 Query: top 10 movies

We can directly query for the top 10 movies by rating, and then get all the necessary information without a join.

I added the comment so that if we want we can choose to return just the minimal informations.

```
use('imdb');
db.movies.find(
    { "rating.averageRating": { $gte:9.0 }, "rating.numVotes": { $gte: 1000 } },
    //{ "title": 1, "rating.averageRating": 1, "rating.numVotes": 1 }
  )
  .sort({ "rating.averageRating": -1 })
  .limit(10)
```

## 3.4 Query: top 10 highest rated episode on Games of Thrones

We can query the embedded documents inside a single document, for example: Here we query the top 10 rated episodes of "Games of thrones"

```
use("imdb");

db.tvSeries.aggregate([
    // Match the Game of Thrones series
    {
      $match: { title: "Game of Thrones" }
    },
    // Unwind the episodes array so that we can sort and filter individual episodes
    {
      $unwind: "$episodes"
    },
    // Filter out episodes that do not have a rating
    {
      $match: { "episodes.rating.averageRating": { $exists: true } }
    },
    // Sort episodes by rating in descending order
    {
      $sort: { "episodes.rating.averageRating": -1 }
    },
    // Limit to the top 10 episodes
    {
      $limit: 10
    },
    // Reshape the output to display relevant episode details
    {
      $project: {
        _id: 0,
        Title: "$episodes.Title",
        Season: "$episodes.ordering.Season",
        Episode: "$episodes.ordering.Episode",
        Rating: "$episodes.rating.averageRating",
        Votes: "$episodes.rating.numVotes",
        Runtime: "$episodes.runtimeMinutes",
        Year: "$episodes.year"
      }
    }
  ])
```

## 3.5 Query: top 10 episodes of top 10 Shows

We can also query the top 10 rated episodes of the top 10 rated tv Shows

```
use("imdb");
db.tvSeries.aggregate([
    // Step 1: Filter TV series with at least 10,000 votes
    {
      $match: { "rating.numVotes": { $gte: 10000 } }
    },
    // Step 2: Sort TV series by average rating in descending order
    {
      $sort: { "rating.averageRating": -1 }
    },
    // Step 3: Limit to the top 10 TV series
    {
      $limit: 10
    },
    // Step 4: Unwind episodes to sort them individually
    {
      $unwind: "$episodes"
    },
    // Step 5: Sort episodes by rating (highest first)
    {
      $sort: { "episodes.rating.averageRating": -1 }
    },
    // Step 6: Group back into the TV series structure, keeping only the top 10 episodes
    {
      $group: {
        _id: "$_id", // Group by TV series ID
        tconst: { $first: "$tconst" },
        title: { $first: "$title" },
        originalTitle: { $first: "$originalTitle" },
        isAdult: { $first: "$isAdult" },
        startYear: { $first: "$startYear" },
        endYear: { $first: "$endYear" },
        runtimeMinutes: { $first: "$runtimeMinutes" },
        genres: { $first: "$genres" },
        //people: { $first: "$people" },
        rating: { $first: "$rating" },
        episodes: { $push: "$episodes" }
      }
    },
    // Step 7: Keep only the top 10 episodes per series
    {
      $project: {
        _id: 1,
        tconst: 1,
        title: 1,
        originalTitle: 1,
        isAdult: 1,
        startYear: 1,
        endYear: 1,
        runtimeMinutes: 1,
        genres: 1,
        people: 1,
        rating: 1,
        episodes: { $slice: ["$episodes", 10] } // Limit episodes to 10 per TV series
      }
    }
  ])
```

## 3.6 Query: Christoper Nolan director

Simple query of all the movie in which chrisopher nolan is the director:

```
use("imdb");
db.movies.find(
    { "people": { $elemMatch: { category: "director", name: "Christopher Nolan" } } },
    {"people":0}
  ).sort({ "rating.averageRating": -1 })
```

## 3.7 Query: Christoper Nolan director and no less important people

Since in the dataset per each title the array of people has an ordering, a query that we could do is to get all the movies in which Cristopher Nolan is the director, and then all the the people that are before him in the ordering.

```
use("imdb");
db.movies.aggregate([
    // Match movies that have Christopher Nolan as a director
    {
      $match: {
        "people": {
          $elemMatch: { category: "director", name: "Christopher Nolan" }
        }
      }
    },
    // Compute Nolan's ordering value
    {
      $addFields: {
        nolanOrdering: {
          $min: {
            $map: {
              input: {
                $filter: {
                  input: "$people",
                  as: "p",
                  cond: {
                    $and: [
                      { $eq: [ "$$p.category", "director" ] },
                      { $eq: [ "$$p.name", "Christopher Nolan" ] }
                    ]
                  }
                }
              },
              as: "p",
              in: "$$p.ordering"
            }
          }
        }
      }
    },
    // Filter the people array to only include entries up to Nolan's ordering
    {
      $project: {
        title: 1,
        rating: 1,
        people: {
          $filter: {
            input: "$people",
            as: "p",
            cond: { $lte: [ "$$p.ordering", "$nolanOrdering" ] }
          }
        }
      }
    }
  ]).sort({ "rating.averageRating": -1 })
```

## 3.8 Query: top 10 most "important" Actors

It could be intresting to see what are the most important actors, we decided to quantify importance as $importance = \text{numTitles} \cdot \text{avgVote}$

```
// Select the database to use.

use('imdb')
db.titles.aggregate([
  // 1. Filter out titles without ratings
  {
    $match: { "rating.averageRating": { $ne: null } }
  },
  // 2. Unwind the people array
  {
    $unwind: "$people"
  },
  // 3. Filter only actors and actresses
  {
    $match: {
      "people.category": { $in: ["actor", "actress"] }
    }
  },
  // 4. Group by actor/actress and titleType to compute per-type averages
  {
    $group: {
      _id: {
        actorName: "$people.name",
        titleType: "$titleType"
      },
      avgRating: { $avg: "$rating.averageRating" },
      numTitles: { $sum: 1 }
    }
  },
  // 5. Group again by actorName to compute overall average and weighted importance
  {
    $group: {
      _id: "$_id.actorName",
      overallAvgRating: { $avg: "$avgRating" },
      totalTitles: { $sum: "$numTitles" },
      perTitleType: {
        $push: {
          titleType: "$_id.titleType",
          avgRating: { $round: ["$avgRating", 2] },
          numTitles: "$numTitles"
        }
      }
    }
  },
  // 6. Add the weighted importance field
  {
    $addFields: {
      weightedImportance: { $multiply: ["$overallAvgRating", "$totalTitles"] }
    }
  },
  // 7. Sort by weighted importance in descending order
  {
    $sort: { weightedImportance: -1 }
  },
  // 8. Limit to top 10 people
  {
    $limit: 10
  },
  // 9. Format output
  {
    $project: {
      _id: 0,
      actorName: "$_id",
      overallAvgRating: { $round: ["$overallAvgRating", 2] },
      totalTitles: 1,
      weightedImportance: 1,
      perTitleType: 1
    }
  }
]);
```

# 4 Conclusions

MongoDB provided a flexible and scalable solution for storing the processed data. The document-based structure of MongoDB allowed us to embed related data (e.g., episodes within TV series), making it easier to query and analyze the data. The use of MongoDB also enabled high-performance read and write operations, suitable for querying and analysis.

The final MongoDB collections, such as movies, TV series, and people, can be easily queried for various use cases. We demonstrated this with example queries, including retrieving average ratings per genre, top-rated movies, and specific queries like finding all movies directed by Christopher Nolan. These queries can be directly applied to provide insights into the IMDb dataset.