Project 2 - MongoDB

Marco Rossini
Politecnico di Torino
Student ID: s291482
m.rossini@studenti.polito.it

***Abstract*—This report illustrates a solution proposal for the creation of a MongoDB database containing data from the IMDb and Rotten Tomatoes online movie databases. The necessary steps for performing the preliminary data ingestion and curation are presented and discussed. In addition, multiple standard and aggregation queries are solved and motivated, together with a custom query of interest which can be useful in a real use case.**

I. Project overview

This section provides an overview of the project at hand and specifies the associated goals, specifications and requirements.

A JSON file containing data from the IMDb online movie database was made available, and must serve as the data source for creating a MongoDB movie database for analysis purposes. The database must be created by importing the JSON data in MongoDB Compass, the official GUI for MongoDB. Unfortunately, the provided data contain some errors and is not ready to be used immediately, therefore some preliminary data ingestion and curation steps must be performed beforehand and motivated. After performing these necessary steps, the database must be exploited to solve some provided queries of interests. The provided queries can be of different types: standard and aggregated. The former ones must be solved using the standard MongoDB document filtering query bar, which allows to retrieve only the documents which match the specified criteria. The latter ones must instead be solved using both the MongoDB aggregation framework and the MapReduce paradigm. All the reported solutions for the queries must include a brief comment which analytically describes the performed steps. In addition, an explanation of the utility in a real world scenario for one of the requested MapReduce queries must be provided. Finally, an additional novel query must be proposed and solved, together with a description of when and how it could be useful in a real use case.

In this report, a solution proposal aimed at meeting these requirements is illustrated and motivated. In particular, the preliminary data ingestion and curation steps are presented and discussed. In addition, the standard and aggregated queries needed to meet the project's requirements are reported, together with an additional custom query of interest which can be useful in a real use case.

II. Data ingestion and curation

This section discusses the necessary steps for performing the preliminary data ingestion and curation. The data curation is an important step which aims at managing the available data in order to make it more useful for users engaging in data discovery and analysis, leaving out only the highest quality information.

After importing the available movie data from the JSON file in MongoDB Compass, documents were inspected. Some attributes were found as having the wrong data types, therefore some JavaScript functions were written in order to parse the correct data types, then and all documents were updated accordingly.[1]

The performed step are described as follows:

- `year`: previously stored as a string, this attribute was parsed and converted to an integer by iterating over all documents, exploiting the `NumberInt()` MongoDB constructor.
- `runtime`: previously stored as a string with pattern `"H h M min"`, the `H` (hours) and `M` (minutes) numerical values were parsed for this attribute and converted to integers by iterating over all documents, exploiting the `NumberInt()` MongoDB constructor. The value for hours (if present) was then multiplied by `60`, and added up to the minutes value, in order to compute the total movie runtime in minutes.
- `lastupdated`: previously stored as a string, this attribute was parsed and converted to a date by iterating over all documents, exploiting the `Date()` MongoDB constructor.
- `imdb.votes`: previously stored as a string, this attribute was parsed and converted to an integer by iterating over all documents, exploiting the `NumberInt()` MongoDB constructor.
- `tomatoes.boxOffice`: previously stored as a string with patterns `"$X.YM"` or `"$X.Yk"`, representing the box office gross in millions or thousands of dollars, respectively. For this attribute, the `X.Y` numerical value was parsed and converted to a decimal one by iterating over all documents, exploiting the `NumberDecimal()` MongoDB constructor. This value was then multiplied by `1000000` or `1000`, depending on the presence of the `M` or `k` as the last character respectively, in order to compute the total box office gross in dollars.

It is worth mentioning that the above data curation steps were crucial for the next phase of analysis: if the original

[1]The complete JavaScript code snippets of the defined data curation functions can be found at this link.

dirty data (characterized by many wrong data types) had been ingested without any fix in the following phase, in fact, it would not be possible to evaluate the information related to the many dirty attributes, making it impossible to solve many of the queries defined as mandatory by the project requirements, dramatically impacting the effectiveness of the analysis.

III. Standard queries resolution

This section reports and explains the proposed solutions for the required standard queries. The queries were solved using the standard MongoDB document filtering query bar, which allows to retrieve only the documents which match the specified criteria.[2]

1) Find all the movies which have been scored higher than 4.5 on Rotten Tomatoes. Sort results by ascending release date.

```
FILTER: { "tomatoes.viewer.rating": { $gt: 4.5 },
          released: { $exists: 1 } }
SORT:   { released: 1 }
```

This query was managed by filtering all documents by `tomatoes.viewer.rating`, imposing a value greater than `4.5` via the `$gt` operator. The documents filtered this way were then sorted by ascending `released` attribute via the value `1`. Although not explicitly required, existence of the `released` attribute was imposed via the `$exists` operator, in order to avoid sorting being non-meaningful for documents lacking such attribute (this can be omitted if a retrieval of all the documents matching the requested condition is a priority).

2) Find the movies which have been written by 3 writers and directed by 2 directors.

```
FILTER: { writers: { $size: 3 },
          directors: { $size: 2 } }
```

This query was managed by filtering all documents by `writers`, imposing a value equal to `3` for the length of the associated array via the `$size` operator, and by `directors`, imposing a value equal to `2` for the length of the associated array. The array sizes were then exploited as the measures for inferring the number of people for each category.

3) For the movies which belong to the "Drama" genre and belong to the USA country, show their plot, duration and title. Sort results by descending duration.

```
FILTER:  { genres: "Drama", countries: "USA" }
PROJECT: { plot: 1, runtime: 1, title: 1}
SORT:    { runtime: -1 }
```

This query was managed by filtering all documents by `genres`, imposing that the associated array contains `"Drama"` as an element, and by `countries`, imposing that the associated array contains `"USA"` as an element. In order to retrieve only the specified requested attributes for each document, a projection was performed on them by imposing the value `1` for `plot`, `runtime` and `title`. Finally, the documents filtered this way were sorted by descending `runtime` attribute via the value `-1`.

[2]The complete text of the solved standard queries can be found at this link.

4) Find the movies which were published between 1900 and 1910, with and IMDb rating higher than 9.0 and which contain the `fullplot` attribute. Show the publication year and the length of the full plot in terms of number of characters. Sort results by ascending IMDb rating.

```
FILTER:  { year: { $gte: 1900, $lte: 1910 },
           "imdb.rating": { $gt: 9.0 },
           fullplot: { $exists: 1 } }
PROJECT: { year: 1, "fullplot_length":
           { $strLenCP: "$fullplot" } }
SORT:    { "imdb.rating": 1 }
```

This query was managed by filtering all documents by `year`, imposing a value greater or equal than `1900` via the `$gte` operator, and less or equal than `1910` via the `$lte` operator (first requirement). In addition, documents were filtered by `imdb.rating`, imposing a value greater than `9.0` via the `$gt` operator (second requirement), and by imposing that the `fullplot` attribute exists via the `$exists` operator (third condition). In order to retrieve only the specified requested attributes for each document, a projection was performed by imposing the value `1` for `year`. The `fullplot` length in terms of number of character was instead computed via the `$strLenCP` operator and returned with name `fullplot_length`. Finally, the documents filtered this way were sorted by ascending `imdb.rating` attribute via the value `1`.

IV. Aggregation framework and MapReduce

This section reports and explains the proposed solutions for the required aggregation queries. The queries reported in section *A* were solved using the MongoDB aggregation framework, while the queries reported in section *B* were solved using the MapReduce paradigm.[3]

A. Aggregation queries

1) Find the average rating score on Rotten Tomatoes for each publication year.

This query was managed by executing a single `$group` stage. The average rating was computed by using the `$avg` operator as the accumulator for the group, specifying the variable `$tomatoes.viewer.rating` as the expression. Documents were grouped by `$year`, specifying it as the ID of the group.

```
$group: { _id: "$year", average_rating:
  { $avg: "$tomatoes.viewer.rating" } }
```

2) For movies which include Italy as a country, get the average number of directors (if existing).

This query was managed by executing a first `$match` stage, in order to impose that the `countries` array contains `"Italy"` as an element and that the `directors` attribute exists.

[3]The complete text of the solved aggregation queries can be found at this link (MongoDB framework) and this link (MapReduce framework).

```
$match: { countries: "Italy",
  directors: { $exists: 1 } }
```

The second stage to be executed was a `$group`. The average number of directors was computed by using the `$avg` operator as the accumulator for the group, specifying the size of the `directors` array (via the `$size` operator) as the expression.

```
$group: { _id: null, average_directors:
  { $avg: { $size: "$directors" } } }
```

3) For movies which contain an IMDb score rating (as a number) compute, separately for each movie genre, the average published year and the maximum score on IMDb.

This query was managed by executing a first `$match` stage, in order to impose that the `imdb.rating` attribute exists (via the `$exists` operator, specifying the value `1`) and that its type is a number (via the `$type` operator).

```
$match: { "imdb.rating":
  { $exists: 1, $type: "number" } }
```

The second stage to be executed was an `$unwind`, in order to deconstruct the `genres` array from the input documents and return a document for each element.

```
$unwind: { path: "$genres" }
```

The third stage to be executed was a `$group`. The average published year was computed by using the `$avg` operator as the first accumulator for the group, specifying the variable `$year` as the expression. The maximum score on IMDb was computed by using the `$max` operator as the second accumulator for the group, specifying the variable `$imdb.rating` as the expression. Documents were grouped by `$genres`, specifying it as the ID of the group.

```
$group: { _id: "$genres",
  average_year: { $avg: "$year" },
  maximum_score: { $max: "$imdb.rating" } }
```

4) Count the number of movies directed by each director. Sort results by descending number of directed movies.

This query was managed by executing a first `$unwind` stage, in order to deconstruct the `directors` array from the input documents and return a document for each element.

```
$unwind: { path: "$directors" }
```

The second stage to be executed was a `$group`. The number of movies was computed by using the `$sum` operator as the accumulator for the group, specifying the value `1` as the expression. Documents were grouped by `$directors`, specifying it as the ID of the group.

```
$group: { _id: "$directors",
  movies_directed: { $sum: 1 } }
```

The third stage to be executed was a `$sort`, in order to sort results by descending `movies_directed` (via the value `-1`).

```
$sort: { movies_directed: -1 }
```

B. MapReduce framework

1) Find the number of movies published for each year.

This query was managed by defining the `mapFunction` in order to `emit` an output document associating each `year` with the value `1`.

```
var mapFunction = function(){emit(this.year, 1);};
```

The `reduceFunction` was then defined in order to return, for each `year`, the sum of all previously associated `1` values, getting this way the number of movies.

```
var reduceFunction = function (id, values) {
  return Array.sum(values); };
```

2) Group movies according to their number of writers (if existing). For each group, find the average number of words in the title.

This query was managed by defining the `mapFunction` in order to `emit` an output document associating each `writers.length` (i.e. the number of writers) with the `title` splitted by a whitespace (i.e. the number of words).

```
var mapFunction = function () {
  if (this.writers) { emit(this.writers.length,
    String(this.title).split(" ").length); } };
```

The `reduceFunction` was then defined in order to return, for each `writers.length`, the division between the sum of all previously associated number of words values divided by their total count, getting this way the average number of words in the title.

```
var reduceFunction = function (id, values) {
  return Array.sum(values) / values.length; };
```

3) Count the number of movies available for each language (if existing).

This query was managed by defining the `mapFunction` in order to `emit` an output document associating each `languages[i]` with the value `1`, for each `i-th` language.

```
var mapFunction = function () {
  if (this.languages) {
    for (var i = 0; i < this.languages.length;
        i++) { emit(this.languages[i], 1); } } };
```

The `reduceFunction` was then defined in order to return, for each `languages[i]`, the sum of all previously associated `1` values, getting this way the number of movies.

```
var reduceFunction = function (id, values) {
  return Array.sum(values); };
```

This query could be usefully applied in a real world business scenario: a streaming platform (e.g. Netflix) might be interested in conducting market analysis to understand which countries could be promising for extending their service on the respective areas. By leveraging on the results retrieved by the performed query, the platform executives would then be helped in supporting their choice, having the possibility to understand if the language availability for that country is covered enough, or it will be necessary to perform additional movie translations.

V. Interest query

This section reports and explains an additional novel query of interest. The proposed query is aggregated, and was solved using the MongoDB aggregation framework. An explanation of its utility in a real use case is also presented and discussed.[4]

***Interest query**: for each genre, retrieve the 30 top rated movie titles in terms of "overall rating". The "overall rating" is calculated as the average of the normalized ratings by Metacritic, IMDb and Rotten Tomatoes. Only movies which contain all the three ratings must be considered.*

*Stage 1 - **`$match`**:*

The first stage to be executed was a `$match`, in order to meet the requirement of considering only the documents which contain all the three ratings. The stage was therefore composed by three `$exists` operators for the attributes `metacritic`, `imdb.rating` and `tomatoes.critic.rating`, which must exist at the same time (value `1`).

```
$match: { metacritic: { $exists: 1 },
  "imdb.rating": { $exists: 1 },
  "tomatoes.critic.rating": { $exists: 1 } }
```

*Stage 2 - **`$unwind`**:*

The second stage to be executed was an `$unwind`, in order to deconstruct the `genres` array from the input documents and return a document for each element. This preliminary step was needed by the next stages in order to meet the requirement of considering the top movies for each genre.

```
$unwind: { path: "$genres" }
```

*Stage 3 - **`$project`**:*

The third stage to be executed was a `$project`, in order to select and compute the specific fields needed by the next stages to solve the query. The existing `title` and `genres` attributes were easily passed by imposing the value `1`. The `overall_rating` was instead computed in the following way: the `metacritic` rating was normalized with a division by `10` (`$divide` operator) in order to belong to the same range of `[0,10]` as `imdb.rating` and `tomatoes.critic.rating` instead of `[0,100]`, thus allowing the computation of the average to be meaningful. The three ratings were then added together (`$sum` operator) and then divided by `3` (`$divide` operator) in order to compute the `overall_rating`.

```
$project: { title: 1,
  genres: 1,
  overall_rating: {
    $divide: [ {
      $sum: [ {
        $divide: [ "$metacritic", 10 ] },
        "$imdb.rating",
        "$tomatoes.critic.rating" ] }, 3 ] } }
```

[4]The complete text of the solved interest query can be found at this link.

*Stage 4 - **`$sort`**:*

The fourth stage to be executed was a `$sort`, in order to sort results by descending `overall_rating`. This step was needed by the next stages in order to meet the requirement of considering only the top movies by "overall rating".

```
$sort: { overall_rating: -1 }
```

*Stage 5 - **`$group`**:*

The fifth stage to be executed was a `$group`, in order to group results by genre as required. The `genres` attribute was therefore specified as the ID of the group. In order to retrieve for each genre the movie titles, the `$push` operator was used as the accumulator for the group, specifying the variable `$$ROOT.title` as the expression. This allowed the `$push` operator to create a new array of movie titles (named `movies`) for each document retrieved by the previous stage, preserving its order. The combined use of the `$push` operator inside the `$group` one allowed then to group such arrays of movie titles by genre (sorted by descending "overall rating").

```
$group: { _id: "$genres",
  movies: { $push: "$$ROOT.title" } }
```

*Stage 6 - **`$project`**:*

The fifth and last stage to be executed was a `$project`, in order to select, for each document, only the `30` top rated movie titles. In order to achieve this, the `$slice` operator was used, specifying `movies` as the array to be sliced by a value of `30`. The resulting array was then named `top_30`.

```
$project: { top_30: { $slice: [ "$movies", 30 ] }}
```

The result of this last stage allowed to finally retrieve the required documents.

The described interest query was proposed since it could be helpful in different real use cases. Two of them could be the following:

- *A business use case:* a streaming platform (e.g. Netflix) might be interested in recommending an undecided user who is browsing their website for a possible movie to watch. By leveraging on the results retrieved by the performed interest query, the platform may show on the homepage, for each genre, a carousel reporting the 30 top rated movies. The user would then be helped in performing his or her choice, having the ability to choose from a careful selection of the best movies (according to the most popular aggregated ratings), along with the flexibility to browse a specific genre based on his or her own preferences.
- *A private use case:* a person who is becoming passionate about cinema might be interested in drawing up a list of movies to watch in the near future based on trusted opinions. By leveraging on the results retrieved by the performed interest query, he or she could narrow his or her field of choice to a more targeted and reliable set of movies to focus on, along with the flexibility to browse a specific genre based on his or her own preferences.