Query Processing and Optimization for Analyzing and Visualizing Data from Selected IMDb Datasets

Lander Peter E. Cua[1], Jacob Bryan B. Gaba[2], Mark Edward M. Gonzales[3] and Hylene Jules G. Lee[4]

Software Technology Department, College of Computer Studies, De La Salle University, Manila, Philippines

[1]lander_peter_cua@dlsu.edu.ph, [2]jacob_bryan_gaba@dlsu.edu.ph, [3]mark_gonzales@dlsu.edu.ph, [4]hylene_jules_lee@dlsu.edu.ph

ABSTRACT

The project aims to provide data visualizations and analyses of selected IMDb datasets in order to generate insights that can inform the decisions of both consumers and producers of the film industry. Two IMDb datasets were selected and copied into a local machine using ETL scripts. A denormalized database was then constructed to serve as the data warehouse for OLAP queries; afterwards, further ETL scripts and data wrangling processes were used to populate the data warehouse. An OLAP application was then created in the form of an interactive dashboard that displays visualizations of the seven generated queries, which involve roll-up, drill-down, slice, dice, and pivot operations on the data. After the results were generated, experimental iterations of the original queries were generated, involving the use of normalized databases, the introduction of indexes and composite indexes, and the creation of revised queries, in order to optimize the queries especially in terms of execution time. Moreover, their execution plans were also examined and compared with one another to characterize the behavior of the queries and examine the bottlenecks and efficiencies of each iteration. In all, the authors were able to demonstrate the benefits of widely used optimization techniques, particularly denormalization and indexing, on real-world data. However, the authors also discovered use cases wherein these techniques detract from the performance of queries, such as the creation of indexes for queries involving roll-up and drill-down. In all, the project generated insights for both the film industry and database design and management alike.

Keywords

Online Analytical Processing (OLAP), Query Optimization, Data Wrangling, Movie Data Analysis

1. Introduction

Over the years, movies have been a reliable source of entertainment for people from all across the world with the movie industry producing hundreds of movies per year with genres varying from comedy, thriller, drama, history, etc. [1]. Starting from the 21st century, before the onset of the COVID-19 pandemic, the annual income of box office revenues alone ranges from $7 to 11 billion, making it an overwhelmingly successful multi-billion dollar industry [28].

With the continuous influx of movies, various online platforms such as the Internet Movie Database (IMDb) and RottenTomatoes keep track of existing and newly released movies by gathering information about each movie's actors, directors, production company, budget, and even ratings. With millions of tracked movies, IMDb is considered to be one of the top consumer sites for the movie industry as it provides an extensive searchable database for both casual use as well as academic viewing [17].

Given the reliability and extent of its database, IMDb was used throughout this paper to analyze and visualize relevant movie data. More specifically, the two datasets: *(i)* IMDb Dataset by the Jožef Stefan Institute and *(ii)* IMDb Extensive Dataset were imported into a local MySQL data warehouse using an extraction, transformation, and loading (ETL) script and were subjected to various query processes. The goal of this study is to highlight the relationships between the different variables in the datasets, namely the directors, actors, genres, gross, votes, and rankings, while applying the necessary transformations to the dataset and employing the use of optimization techniques to make querying the large datasets more efficient.

This paper also visualizes the findings obtained from the query processes through the development of an online analytical processing (OLAP) application entitled "Reel Data" using the data visualization software Tableau, where optimization strategies were applied to enhance the application's performance. The OLAP application was designed to be easily utilized by both casual moviegoers and movie critics alike, allowing them to quickly see trends and notable movie industry statistics that could perhaps influence their next choice of movie.

2. Data Warehouse

This section gives a discussion of the IMDb datasets that were selected for the use cases of the OLAP application, as well as an explanation of the dimensional model of the data warehouse.

2.1 Selected IMDb Datasets

In order to enrich the volume of data and the value of the data analysis, two IMDb datasets were used in this project:

- ***Primary Dataset:*** **IMDb Dataset by the Jožef Stefan Institute** (henceforth referred to as **IMDb IJS**). Collected by Janez Kranjc of the Jožef Stefan Institute [38] and hosted in the Czech Technological University Prague Relational Learning Repository [27], it is a relational database that contains information on movies and their respective genres, actors (alongside their roles), and directors. It is hosted in a MariaDB server, which is an open-source fork of MySQL.

  The entity-relationship diagram (ERD) of this dataset is shown in Figure 1. The tables are described in Table 1.

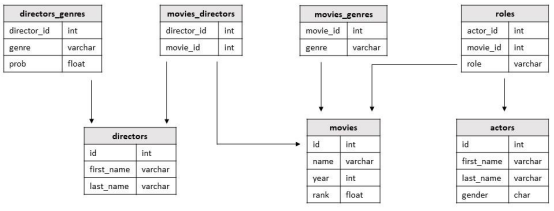

Figure 1. ERD of IMDb IJS

Table 1. Tables of IMDb IJS

Name	Description
`movies`	Movies
`actors`	Actors
`directors`	Directors
`roles`	Mapping between the movies and actors, alongside their respective roles
`movies_directors`	Mapping between the movies and directors
`directors_genres`	Mapping between the directors and genres
`movies_genres`	Mapping between the movies and genres

- ***Auxiliary Dataset:*** **IMDb Extensive Dataset** (henceforth referred to as **IMDb Supplementary**). Scraped by Stefano Leone and hosted in the data science platform Kaggle [21], it is a set of comma-separated values (CSV) files that contain information on movies and their respective years of release, countries of origin, ratings, alternative titles, running times, languages, descriptions, budgets, and cast and production crew members (alongside their roles and biographical notes).

  It also provides detailed information on the ratings of each movie, showing their breakdowns based on the number of stars, sex, age, and nationality (whether the rater is from the United States or not).

  The ERD is shown in Figure 2. Note that, since CSV files do not explicitly store relationships and data types, these were implied based on their structures and contents. The files (tables) are described in Table 2.

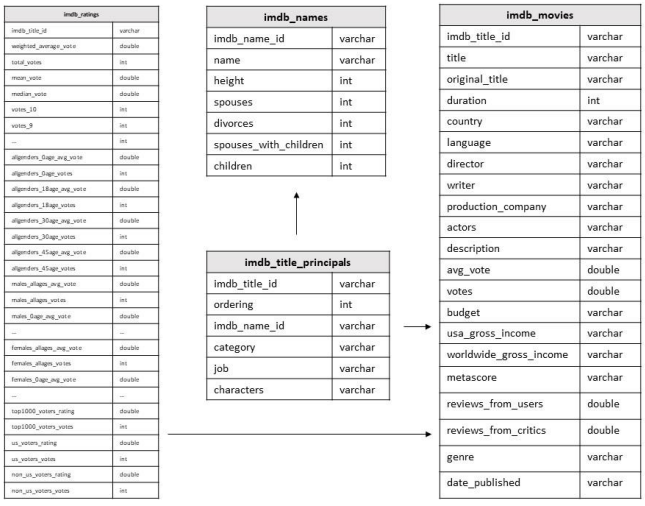

Figure 2. ERD of IMDb Supplementary

The motivation for supplementing the primary dataset with an auxiliary data source can be gleaned from these descriptions in Tables 1 and 2. Although IMDb IJS contains entries for 381,762 movies, the only numerical data it provides are the ratings of the movies (corresponding to the column `` `rank` ``). However, since these ratings are essentially averages, it would be incorrect to perform roll-up operations on them; for instance, aggregating them via average (i.e., `AVG(`rank`)`) is equivalent to computing the average of averages, which is an improper or inutile metric in most data analytics use cases [22].

Table 2. Tables of IMDb Supplementary

Name	Description
`imdb_movies`	Movies
`imdb_ratings`	Numerical breakdown of the ratings
`imdb_names`	Cast and production crew members
`imdb_title_principals`	Mapping between the movies and cast and production crew members, alongside their respective roles

On the other hand, IMDb Supplementary provides a detailed numerical breakdown of the ratings, thus allowing for proper aggregation and, consequently, for execution of roll-up operations. Moreover, it also contains information on the country/countries of origin of the movies, adding a geographical dimension to the data.

2.2 Dimensional Model

The constructed data warehouse follows a star schema, composed of one (1) fact table and seven (7) dimension tables. The selection of tables for inclusion in the schema and the use of a star schema is justified by the relevant use cases in the OLAP application. The foremost advantage of designing a star schema is that it is denormalized, which is in consonance with the recommendation of Johnson and Jones [18] to employ a denormalized database to accommodate the high volume of read operations (compared to insertions and updates) that characterize an OLAP environment.

In particular, denormalization results in fewer tables albeit with more columns. The decrease in the number of tables benefits join operations — which are fixtures of data analysis — since the number of nested loops also decreases. For illustration, suppose that there are m normalized tables, with the i^{th} table having r_i rows. Assuming a naïve nested-loop join algorithm that requires full table scans, the time complexity of joining all these tables is $\Theta(\prod_{i=1}^{m} r_i)$, which has an upper bound of $O((\max r_i)^m)$.

Suppose that denormalization results in p tables, with the i^{th} table having s_i rows. The time complexity of a naïve nested-loop join is $\Theta(\prod_{i=1}^{p} s_i)$, which has an upper bound of $O((\max s_i)^p)$. While the introduction of redundancies following denormalization yields an increase in the number of rows, i.e., $r_i < s_i$, the dominant factor in the asymptotic analysis is not the base (the number of rows) but the exponent (the number of tables). Therefore, the decrease in the number of tables after denormalization, i.e., $m > p$, is indicative of magnitudes of performance optimization with respect to execution time. From a low-level (hardware) perspective, having fewer tables also minimizes the number of secondary storage accesses needed during query processing [40], which incur particularly high costs for hard-disk drives due to disk rotation latency.

These advantages of employing a star schema are supported by Han, Kamber, and Pe [13]. They also pointed out that, in data warehouse implementations, star schemas are more widely used

compared to snowflake schemas since the latter type involves normalization into multiple related tables, thus slowing down join operations.

Nevertheless, it may be important to note some trade-offs with using a star schema to improve retrieval and join speeds. As a consequence of redundant attributes and data, space requirements increase, and insertion and updates become slower. However, as a data warehouse is ipso facto intended to serve as a nonvolatile repository for large volumes of data and insertions are not expected to happen at the same rapid pace as in online transaction processing (OLTP) applications, these trade-offs are anticipated and do not pose constitute the primary concerns in designing the dimensional model.

The star schema for this project's data warehouse is shown in Figure 3 and explained in Section 2.2.1.

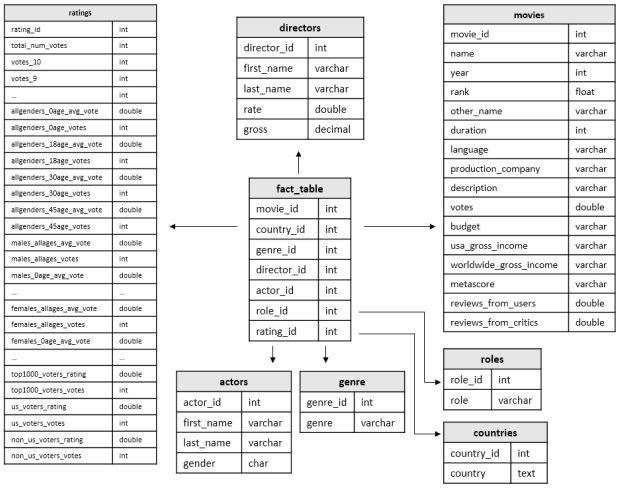

Figure 3. ERD of the Data Warehouse (Star Schema)

2.2.1 Fact and Dimension Tables

The fact table consists of seven (7) columns, namely `movie_id`, `country_id`, `genre_id`, `director_id`, `actor_id`, `role_id`, and `rating_id`, that relate it to the dimensional tables: `movies`, `countries`, `genres`, `directors`, `actors`, `roles`, and `ratings`. This section focuses on the denormalization strategies employed in the construction of these tables.

First, the lowest level of granularity was identified to be the movies, as it is the atomic level of data in a movie data analysis. Hence, `movie_id` was added to the fact table, and the pertinent attributes in `movies` and `imdb_movies` were merged into the `movies` dimension table. To standardize the names of the columns and make them more descriptive, these columns were renamed:

- `imdb_ijs.movies.id` to `movie_id`
- `imdb_ijs.directors.id` to `director_id`
- `imdb_ijs.actors.id` to `actor_id`
- `imdb_supplementary.imdb_movies.title` to `other_name` (since the movie name provided in `imdb_ijs`, the primary data source, is equivalent to `original_title` in `imdb_supplementary`, the auxiliary dataset)

The selection of tables and attributes from IMDb IJS and IMDb Supplementary to be integrated into the star schema was based on the particular OLAP use cases that the authors intended for this project; these use cases, which seek to provide some analytics and insights into the movie industry, are explained in detail in Section 4 (OLAP Application). A more in-depth discussion of the specific considerations during the selection process is given in Section 2.3.

Han, Kamber, and Pe [13] recommend collapsing many-to-many relationships with low cardinalities and volatility; examples of these are movies-to-directors, movies-to-actors, movies-to-genres, and movies-to-countries. For the directors, this strategy was utilized to add `directors_id` to the fact table and eliminate the need for the `movies_directors` mapping table (Listing 1). The key added to the fact table as part of denormalization is in bold.

Listing 1. Director Denormalization

Normalized:
```
movies(movie_id, name, ...)
directors(director_id, first_name, last_name)
movies_directors(director_id, movie_id)
```

Denormalized:
```
fact_table(movie_id, directors_id)
directors(director_id, first_name, last_name)
```

Similarly, a collapsing denormalization strategy was used for the actors, resulting to the addition of `actor_id` to the fact table (Listing 2).

Listing 2. Actor Denormalization

Normalized:
```
movies(movie_id, name, ...)
actors(actor_id, first_name, last_name,
  gender)
roles(actor_id, movie_id, role)
```

Denormalized:
```
fact_table(movie_id, directors_id, actors_id)
actors(actor_id, first_name, last_name,
  gender)
```

However, since the `roles` mapping table also contains the roles played by the actors, a vertical partitioning strategy was employed to separate `roles` into another table and drop the attributes that serve only to map movies and actors (Listing 3). Since the roles per se are lengthy text fields (up to 100 characters) and they are not among the foci of the OLAP use cases in this project, it would be more advantageous to relegate them to another table in order to optimize query performance by decreasing the number of pages and, consequently, reducing disk input/output (I/O) [6]. This columnar splitting also makes the said relation more explicit or visible to the database users [40].

Listing 3. Role Denormalization

Normalized:
```
movies(movie_id, name, ...)
actors(actor_id, first_name, last_name,
  gender)
roles(actor_id, movie_id, role)
```

Denormalized:
```
fact_table(movie_id, directors_id, actors_id,
  role_id)
roles(role_id, role)
* roles_mapping(actor_id, movie_id)   * Dropped
```

Note that an auto-incrementing surrogate key, i.e., `role_id`, was added as well as part of the denormalization. While the attribute `role` could have served as the primary key, four-byte integers are more suited to cluster indexing since they occupy fewer bytes; moreover, as opposed to strings, which have to be parsed to the equivalent ASCII code per character prior to applying collation rules [41], comparing integers is usually a native CPU instruction. The use of surrogate keys in OLAP dimensional modeling is also recognized in existing database literature [19].

The same motivations for constructing an auto-incrementing surrogate key extend to the genres (Listing 4).

Listing 4. Genre Denormalization

Normalized:

```
movies(movie_id, name, ...)
movies_genres(movie_id, genre)
```

Denormalized:

```
fact_table(movie_id, genre_id, directors_id,
  actors_id, role_id)
genres(genre_id, genre)
```

Meanwhile, vertical partitioning of the `movies` dimension table and addition of an auto-incrementing surrogate key were jointly applied to the countries of origin with the intention of highlighting the geographical dimension of the data (Listing 5).

Listing 5. Country Denormalization

Normalized:

```
movies(movie_id, name, ..., country, ...)
```

Denormalized:

```
fact_table(movie_id, country_id, genre_id,
  directors_id, actors_id, role_id)
movies(movie_id, name, ...)
country(country_id, country)
```

Finally, collapsing and addition of an auto-incrementing surrogate key were employed to the numerical breakdown of the ratings (Listing 6). Note that a temporary table (`ijs2supplementary`) was created to map the movies in IMDb IJS to the movies in IMDb Supplementary via their IDs.

Listing 6. Rating Denormalization (Intermediate)

Normalized:

```
movies(movie_id, name, ...)
ijs2supplementary(movie_id, imdb_title_id)
imdb_ratings(imdb_title_id, votes_10, votes_9,
   ...)
```

Denormalized (Intermediate):

```
fact_table(movie_id, country_id, genre_id,
  directors_id, actors_id, role_id, rating_id)
ratings(ratings_id, votes_10, votes_9, ...)
```

Another dimension modeling technique that is specific to OLAP is the inclusion of derived attributes that reflect frequently used aggregation functions [19]. Precomputing these values helps speed up and simplify queries since these values are stored as part of the data warehouse. Therefore, the overall number of votes, the total per gender and age, and their averages were also included in the data warehouse schema (Listing 7).

Listing 7. Rating Denormalization (Final)

Normalized:

```
movies(movie_id, name, ...)
actors(actor_id, first_name, last_name,
gender)
imdb_ratings(imdb_title_id, votes_10, votes_9,
   ...)
```

Denormalized (With attributes from aggregation):

```
fact_table(movie_id, country_id, genre_id,
  directors_id, actors_id, role_id, rating_id)
ratings(ratings_id, total_num_votes, votes_10,
  votes_9, ..., allgenders_0age_avg_vote,
  allgenders_0age_avg_vote,
  allgenders_18age_avg_vote,
  allgenders_30age_avg_vote,
  allgenders_45age_avg_vote,
  males_allages_avg_vote, males_allages_votes,
  females_allages_avg_vote,
  females_allages_votes, ...)
```

It may be important to note that the resulting tables do not exhibit an overt dimension hierarchy as a result of the nature of the data sources and of the domain as well. Oracle [43] stipulates two strict constraints in defining a hierarchy. First, the parent-to-children relationship should obey a one-to-many relationship, i.e., the parent may have many children but a child can have only a single parent. Second, the relationship between hierarchy levels and their dependent dimension attributes should remain one-to-one.

Ergo, while it is possible to group movies based on their genres, their many-to-many relationship violates the first condition of hierarchical integrity since a genre can have multiple movies and a movie can have multiple genres as well. This is in contrast to the GOSales dataset (previously analyzed in the course), where a product line can have multiple products but a product can belong to only a single product line.

Common examples of hierarchical relationships found in database literature include those in geographical (e.g., state → city → town) and temporal (e.g., year → quarter → month) dimension tables. However, IMDb IJS contains only the year of release, and IMDb Supplementary contains only the country of origin. Despite this scarcity of available attributes, dimension hierarchies can still be constructed to an extent by deriving the parent. For example, in the `movies` dimension table, the year of a movie's release can be taken as a child of the decade (which can be algorithmically computed from the year), as seen in Figure 4.

Figure 4 also illustrates how the decade → year relationship satisfies the two constraints for hierarchical integrity.

Figure 4. Constructed Dimension Hierarchy (Decade → Year)

Having many-to-many relationships is a recognized challenge in relational database design, especially in the context of OLAP; Malinowski and Zimányi [24] describe these relationships "as very common in real-life applications." At least two authors [24,

23] have even proposed the terminology *non-strict hierarchy* in an attempt to relax or extend the classical notion of hierarchical integrity; however, support for this conceptual notion has not yet seen widespread implementation in OLAP tools.

In summary, the star schema of the data warehouse features a fact table consisting of seven columns; the facts were selected in such a way that *(i)* the fact table serves as the central link for connecting the different dimension tables and *(ii)* the movies are the lowest level of granularity for the analytics. To this end, each of the facts is an integer ID that relates the fact table to every dimension table, and all collapsing strategies for denormalization consult the `movies` table and the pertinent mappings.

On the other hand, the dimension tables were derived from the two data sources and were selected on the basis of their relevance to the OLAP application (discussed in detail in Section 4). The `movies` table was constructed from merging IMDb IJS' `movies` and IMDb Supplementary's `imdb_movies`; the `actors`, `directors`, `genre`, and `roles` tables were derived from IMDb IJS whereas the `countries` and `ratings` tables were from IMDb Supplementary.

The eight tables comprising the data warehouse are described in Table 3.

Table 3. Tables of the Data Warehouse

Name	Description
`fact_table`	Fact table collating the IDs relating to the dimension tables
`- movie_id`	ID of the movie (with respect to the `movies` table)
`- country_id`	ID of the country (with respect to the `countries` table)
`- genre_id`	ID of the genre (with respect to the `genres` table)
`- director_id`	ID of the director (with respect to the `directors` table)
`- actor_id`	ID of the actor (with respect to the `actors` table)
`actors`	Dimension table containing information about all actors in the database
`- actor_id`	ID of the actor
`- first_name`	Actor's first name
`- last_name`	Actor's last name
`- gender`	Actor's gender (M - Male | F - Female)
`countries`	Dimension table containing all countries wherein films in the database have premiered.
`- country_id`	ID of the country
`- country`	Country's name
`directors`	Dimension table containing all film directors in the database.
`- director_id`	ID of the director
`- first_name`	Director's first name
`- last_name`	Director's last name
`- rate`	Total ratings of all the directors' movies
`- gross`	Director's total gross earnings[1]
`genres`	Dimension table containing all genres of films in the database
`- genre_id`	ID of the genre
`- genre`	Name of the genre
`movies`	Dimension table containing all films in the database.
`- movie_id`	ID of the movie
`- name`	Name of the movie
`- year`	Year the movie was released
`- rank`	Total score of the movie (out of 10) among all users
`- other_name`	Movie's alternate title -- if any
`- duration`	Movie's duration
`- language`	Language of the film
`- production_company`	Company in charge of the film's production
`- description`	Short description of the film
`- votes`	Total number of votes on the movie's rank
`- budget`	Total budget of the film
`- usa_gross_income`	Total gross of the film (restricted to the United States)
`- worldwide_gross_income`	Total gross of the film
`- metascore`	Average critic rating of a film (will be null if less than four of critic's reviews are collected)
`- reviews_from_users`	Number of user reviews of the film
`- reviews_from_critics`	Number of critic reviews of the film
`ratings`	Dimension table containing all ratings of films in the database.

[1] `rate` and `gross` were sourced from an external data source: IMDb Full [27]. However, since these were the only attributes taken from IMDb Full, this dataset was not included in the thorough discussion of data sources in Section 2.1 for brevity.

- rating_id	ID of the rating
- votes_10	Count of user votes with a value of '10'
- votes_9	Count of user votes with a value of '9'
- votes_8	Count of user votes with a value of '8'
- votes_7	Count of user votes with a value of '7'
- votes_6	Count of user votes with a value of '6'
- votes_5	Count of user votes with a value of '5'
- votes_4	Count of user votes with a value of '4'
- votes_3	Count of user votes with a value of '3'
- votes_2	Count of user votes with a value of '2'
- votes_1	Count of user votes with a value of '1'
- allgenders_0age_avg_vote	Average of votes from age range 0 to 17 across all genders
- allgenders_0age_votes	Count of votes from age range 0 to 17 across all genders
- allgenders_18age_avg_vote	Average of votes from age range 18 to 29 across all genders
- allgenders_18age_votes	Count of votes from age range 18 to 29 across all genders
- allgenders_30age_avg_vote	Average of votes from age range 30 to 44 across all genders
- allgenders_30age_votes	Count of votes from age range 30 to 44 across all genders
- allgenders_45age_avg_vote	Average of votes from age range 45+ across all genders
- allgenders_45age_votes	Count of votes from age range 45+ across all genders
- males_allages_avg_vote	Average of votes of males across all ages
- males_allages_votes	Count of votes of males across all ages
- males_0age_avg_vote	Average of votes of males from age range 0 to 17
- males_0age_votes	Count of votes of males from age range 0 to 17
- males_18age_avg_vote	Average of votes of males from age range 18 to 29
- males_18age_votes	Count of votes of males from age range 18 to 29
- males_30age_avg_vote	Average of votes of males from age range 30 to 44
- males_30age_votes	Count of votes of males from age range 30 to 44
- males_45age_avg_vote	Average of votes of males from age range 45+
- males_45age_votes	Count of votes of males from age range 45+
- females_allages_avg_vote	Average of votes of females across all ages
- females_allages_votes	Count of votes of females across all ages
- females_0age_avg_vote	Average of votes of females from age range 0 to 17
- females_0age_votes	Count of votes of females from age range 0 to 17
- females_18age_avg_vote	Average of votes of females from age range 18 to 29
- females_18age_votes	Count of votes of females from age range 18 to 29
- females_30age_avg_vote	Average of votes of females from age range 30 to 44
- females_30age_votes	Count of votes of females from age range 30 to 44
- females_45age_avg_vote	Average of votes of females from age range 45+
- females_45age_votes	Count of votes of females from age range 45+
- top1000_voters_rating	Average of votes from the top 1000 users who have voted the most number of times on IMDb
- top1000_voters_votes	Count of votes from the top 1000 users who have voted the most number of times on IMDb
- us_voters_rating	Average rating of voters in the United States
- us_voters_votes	Count of votes from voters in the United States
- non_us_voters_rating	Average rating of voters outside the United States
- non_us_voters_votes	Count of votes from voters outside the united states
roles	Dimension table containing all roles of all actors in each movie in the database.
- role_id	ID of the role
- role	Name of the role

2.3 Issues Encountered and Considerations

The primary issue that affected the database design process was the size of the datasets. The largest table in the primary data source (`actors`) contains 815,842 rows; hence, a trial-and-error approach where the data would first be loaded into a draft schema, which would undergo repeated refinement (in case it was found to be not well suited for the OLAP requirements) was not the most optimal approach in light of the project time frame and the limited computational resources of the authors.

In this regard, the authors followed the waterfall methodology in software engineering, where the OLAP applications were first formulated, analyzed, and finalized before designing the schema. In the schema design process, the authors encountered three major

decision points: *(i)* the selection of tables, *(ii)* the denormalization strategies, and *(iii)* the dimension hierarchies.

The main criterion for the selection of tables and columns was their relevance to the OLAP applications. At the same time, the researchers intended to give some provision in the data warehouse for relations and attributes not directly used in the queries but may be relevant for future analysis. Therefore, all the tables in IMDb IJS, the primary dataset, were included in the schema, with the exception of `directors_genres` since all the data in this table, such as the probability of a director directing a movie of a particular genre, are already analytical in nature, i.e., instead of storing raw data, they contain results of analyzing the data found in the other tables.

For IMDb Supplementary, its tables containing the movies (which adds several attributes to IMDb IJS' `movies`) and ratings (which is not present in IMDb IJS) were integrated into the star schema. The two remaining tables, namely those containing biographical notes on the cast and production crew and their respective roles, were excluded since the primary dataset already had information on the directors and the actors' roles, and these tables, which are only parts of the auxiliary dataset, were not contributory to the project's OLAP applications.

The abundance of denormalization strategies was also a challenge for the authors, especially since each has its own advantages and trade-offs. In these instances, the authors consulted existing database literature in order to inform the evaluation in the context of the current project. For instance, collapsing many-to-many relationships, as was done in this project, eliminates abstraction and conceptual distinction between data since the explicit "link table" is dropped and the entries are flattened into a single table.

This loss of conceptual distinction between data was made up for by the choice of dimension tables, which also led to the decision to employ vertical partitioning. Although initially counterintuitive as it resulted in the addition of tables (which seemed contrary to the purpose of denormalization), it helped improve the clarity and organization of the schema for both designers and end users.

Physically, the collapsing strategy also increases table sizes, as evidenced in the fact table having 6,617,850 rows (over eight times that of the largest table in the source datasets). Nevertheless, Shin and Sanders [40] note that this comes with a 15.8% decrease in join cost, and time optimization is the overriding concern in this project's OLAP applications.

Another design decision was made in relation to the fact table. Typically, the composite of the keys is assigned as the primary key of the fact table although this was not possible due to the presence of null values in some attributes. Nonetheless, this was not entirely unexpected, as real-world datasets, such as the IMDb database, tend to exhibit bias towards films produced in Western countries when it comes to completeness of data [7].

For these cases, one alternative suggested by Kimball and Ross [19] is to introduce a surrogate fact key, i.e., an auto-incrementing integer key for the fact table. However, since keys are not essential in data warehouses (the necessary integrity constraints are already assumed to have been enforced in the transactional databases), the authors decided not to enforce a primary key constraint for the fact table. The addition of a surrogate fact key would only add baggage to the data warehouse and contribute to overhead during insertion. It does not provide any performance gain as well since the fact table is rarely referred to using its surrogate key; instead, optimization was done via setting up indexes on its attributes (discussed in Section 5 [Query Processing and Optimization]). Hobbs, Hillson, Lawande, and Smith [14] also acknowledge fact tables without primary keys in their database warehouse literature.

Finally, as explained in the preceding subsection (Section 2.2), constructing dimension hierarchies was also a challenge. The decade → year shown in Figure 4 prompted the authors to consider whether the decade should be added as an attribute in the `movies` table or whether it would help to create a new table for the temporal dimension (containing the decade and year). In the end, the authors decided to push through with neither approach, as this would introduce potentially contrived complexities to the schema. Furthermore, the decade can be computed by performing straightforward, inexpensive arithmetic on the year; thus, the contribution of having it as a distinct attribute is negligible.

3. ETL Script

This section discusses the script for extracting, transforming, and loading (ETL) the data into the data warehouse. The ETL pipeline was done using Apache NiFi, an open-source, distributed dataflow software from the Apache Software Foundation [3].

3.1 Creating Local Copies and Preliminary Data Wrangling

Since the server of the Czech Technological University Prague Relational Learning Repository is not suited for high volumes of network traffic and the databases it hosts are read-only, a local copy of the IMDb IJS dataset was created. Moreover, since the IMDb Supplementary is provided as CSV files, the dataset was first transformed before also loading into the MySQL server.

It was also necessary to perform some preliminary data wrangling (integrated into the transformation phase) prior to transferring data into the warehouse, as leaving them without any preprocessing would interfere with the join operations in the extraction phase or have fundamental repercussions on the data representation that would otherwise be complicated to address, especially once the entries have already been loaded into the warehouse.

The ETL pipelines for IMDb IJS and IMDb Supplementary are illustrated in Figures 5 and 6, respectively. (Only the `directors` table in the external data source IMDb Full was considered since only two columns from this dataset, namely the director's total rating and gross earnings, are of interest to the authors. Since it is a relational database, it follows the same extraction ETL pipeline as IMDb IJS.)

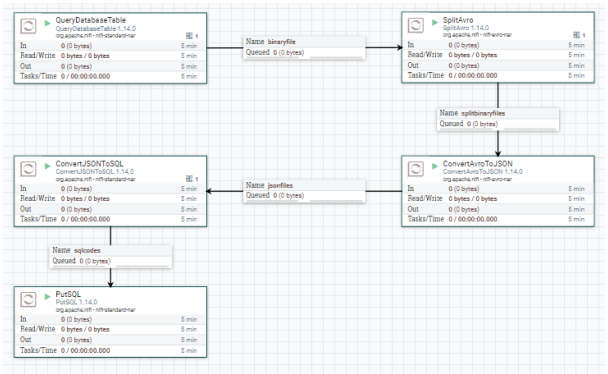

Figure 5. ETL Pipeline for Creating Local Copies (IMDb IJS)

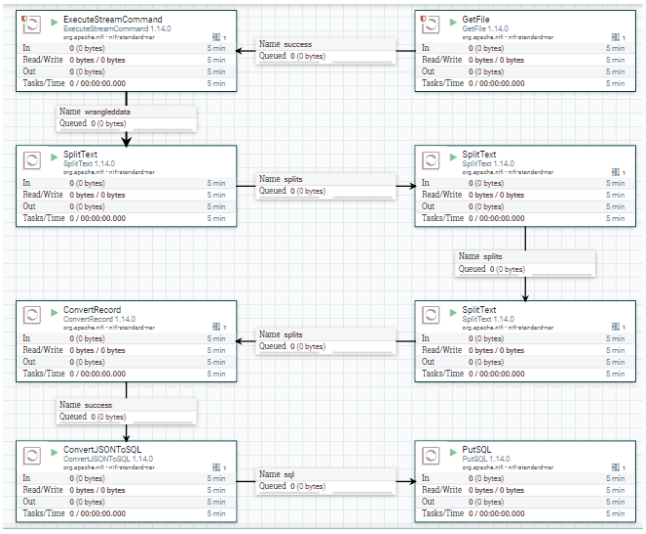

Figure 6. ETL Pipeline for Creating Local Copies and Preliminary Data Wrangling (IMDb Supplementary)

3.1.1 Extract

The contents of the tables in the IMDb IJS schema were extracted to the data warehouse, making a total of seven tables. For each table, the `QueryDatabaseTable` processor was used to extract its contents. First, a connection was established to the source of the data, which is the `imdb_ijs` schema on the MariaDB server, using a Database Connection Pooling Service. In order to lessen the bottleneck on the succeeding steps of the process, the results of the query were split into multiple binary files of 10,000 rows each, whereupon they can be further split and transformed by the succeeding processors in the pipeline.

On the other hand, the four tables of the IMDb Supplementary schema were stored as text files (particularly CSV files); hence, the `GetFile` processor was used for data extraction. Rather than establishing a connection to the source server, the source files were stored in a local machine directory, the path to which was provided as the input directory of the processor. The source files were deleted after the extraction process in order to prevent duplicates caused by the continuous data retrieval of the processor. The process yielded text files that are further split and transformed along the pipeline.

3.1.2 Transform – IMDb IJS

After extracting the binary file, the `SplitAvro` processor was used to split the table into multiple binary files, one for each row. As Apache Nifi does not support direct Avro-to-SQL conversion, the `ConvertAvroToJSON` processor was used to first convert the AVRO files to the JSON format. Finally, the individual JSON files were then processed through the `ConvertJSONToSQL` processor, which was connected to the data warehouse using a JDBC connection pool in order to inform the data types of the data values in the JSON files.

3.1.3 Transform – IMDb Supplementary

Before splitting the resultant file, it is necessary to perform some data wrangling. This is due to the `date_published` column containing entries that were inconsistent in format while a significant number of rows of the `country` and `genre` columns included compound entries separated by commas.

The dates originally listed in the dataset had three varying schemes: (1) the `DD-MM-YYYY` format, (2) the `DD/MM/YYYY` format, and (3) only the year in `YYYY` format. Each `date_published` entry was split into three columns representing the day, month, and year, and these columns were concatenated into the format `YYYY-MM-DD` which MySQL recognizes as a `DATE` data type. As for the entries which comprised only of a year, these were all given fixed `01` values for both the `MM` and `DD` fields for consistency.

In addition, the countries of origin and the genres were exploded; "explode" refers to both the process of transforming each element of a list or array value into a separate row and the function in the data manipulation library Pandas for performing this operation [37]. This is similar to the concept of denesting in JavaScript Object Notation (JSON) files.

In IMDb Extensive, the countries of origin are encoded as single strings under the column `country`, with commas used to separate multiple countries. Hence, exploding the country value `Italy, France` will yield two rows with the same attributes except for the country (the first row will have `Italy` as its country while the second one will have `France` as its country).

Likewise, the `genre` column containing strings that consisted of multiple genres were separated in the same manner. After exploding the genre entry `Crime, Drama, Horror`, three rows containing the same attributes except for the split genres were added in place of the original row.

Both of these data wrangling tasks were performed in Python to maximize the use of Pandas; the scripts are given in Listings 8. In order to integrate these Python scripts as part of the Apache NiFi pipeline, the `ExecuteStreamCommand` processor was used. Its command arguments were set to `date_published country`, and `genre` to allow access to these columns via the `sys.argv` list.

Listing 8. Script for Preprocessing IMDb Supplementary

```
import pandas as pd
import sys

movies_df = pd.read_csv(sys.stdin)

split_dates_df = movies_df
temp =
   split_dates_df[sys.argv[1]].str.split
   ('[-/]', expand = True)
split_dates_df["year"] = temp[2]
split_dates_df["month"] = temp[1]
split_dates_df["day"] = temp[0]

split_genres_df = movies_df.drop(['year',
   'month', 'day'], axis = 1)

split_dates_df.loc[split_dates_df.day.astype
   (int) >= 1894, 'year'] = split_dates_df.day
split_dates_df.loc[split_dates_df.day.astype
   (int) >= 1894, 'day'] = '01'

split_dates_df['year'].replace(to_replace=
   [None], value='01', inplace=True)
split_dates_df['month'].replace(to_replace=
   [None], value='01', inplace=True)
split_dates_df['day'].replace(to_replace=
   [None], value='01', inplace=True)
```

```
split_dates_df =
split_dates_df.drop(sys.argv[1], axis = 1)
split_dates_df[sys.argv[1]] =
split_dates_df['year'] + '-' +
split_dates_df['month'] + '-' +
split_dates_df['day']
split_dates_df = split_dates_df.drop(['year',
   'month', 'day'], axis = 1)

split_countries_df = split_dates_df
split_countries_df =
split_countries_df.assign(country=
   split_countries_df[sys.argv[2]].
   str.split(',')).explode(sys.argv[2])

split_genres_df = split_countries_df
split_genres_df =
split_genres_df.assign(genre=
   split_genres_df[sys.argv[3]].
   str.split(',')).explode(sys.argv[3])

split_genres_df.to_csv(sys.stdout,
   index=False)
```

After loading the dataset into a `pandas` dataframe object, the entries of the `date_published` column accessible through `sys.argv[1]` were split on the characters `-` and `/` to account for the `DD-MM-YYYY` and `DD/MM/YYYY` formats, and were stored in three separate columns `year`, `month`, and `day` in the `split_dates_df` dataframe. As previously mentioned, some entries in the `date_published` column contained only the year of publication. This resulted in the newly-generated `day` and `month` columns to yield `None` values. For consistency, these values were all replaced with `01` before the three new columns were finally concatenated - following the `YYYY-MM-DD` format - and inserted into the `sys.argv[1]` column.

Following this, the entries containing countries of origin and genres with comma-separated strings were split into multiple rows using the `pandas` explode function in conjunction with `split` and `assign`. `split` generates a list from the entries separated using `,` as the delimiter, `explode` transforms the lists into rows, and `assign` generates a new entry in the table for each separated country or genre.

In order to process the data one entry at a time, the `SplitText` processor was used to split the file into individual rows. However, splitting the file immediately into single rows proved to be a bottleneck in the pipeline, taking up excessive time and resulting in out-of-memory errors. Thus, in order to optimize the pipeline, multiple `SplitText` processors were used; the first processor split the file into flow files of 10000 rows, the second processor split these into smaller flow files of 100 rows, and the final processor split these into single rows. Ultimately, the use of successive `SplitText` processors lowered both the time and space costs of the pipeline.

Since the Apache Nifi processors do not offer direct conversions between binary and SQL files, the `ConvertRecord` processor was first used to convert the CSV files to their JSON equivalents. In particular, the processor was set up with a `CSVReader` and a `JSONRecordSetWriter`.

Finally, the resulting JSON files were converted to SQL using the `ConvertJSONtoSQL` processor. A JDBC connection pool to the destination (i.e., the IMDb Supplementary tables in the data warehouse) was established in order to ensure that the values of the data match the data types of the destination.

3.1.4 Load

After undergoing the data wrangling and transformation processes, the data is loaded onto the data warehouse using the `PutSQL` processor. The processor is connected to the data warehouse using a JDBC connection pool, and is used to run an `INSERT` statement for each table. As the `PutSQL` processor is used solely for loading in the pipeline, no further processing is done through its SQL statement; all data in the table is inserted into the data warehouse.

3.2 Additional Data Wrangling

After the creation of the local copies, additional data wrangling was performed in preparation for the ETL pipeline to transfer the entries into the data warehouse. Note that, for all the processes described in this section, the source and destination schema is the one that houses the local copies (not the data warehouse).

3.2.1 Standardizing Movie Titles

The purpose of this step is the standardization of the encoding of movie titles (names) between the two source datasets since IMDb IJS transposes the initial grammatical article to the end of the title and appends the year of release in parentheses whereas IMDb Supplementary does not.

To illustrate, consider the film *The Godfather*. It is encoded in IMDb IJS as `Godfather, The (1972)` , whereas it is encoded in IMDb Supplementary as `The Godfather`. In the context of this project, the standardized format is the one followed by IMDb Supplementary. Listing 9 presents the SQL script for this standardization.

Listing 9. Movie Title Standardization

```
UPDATE movies SET `name` =
   RTRIM(REVERSE(SUBSTRING(REVERSE(`name`),
   LOCATE(" ",REVERSE(`name`)))))
WHERE `name` IN (
   SELECT `name`
   FROM (
      SELECT `name`
      FROM movies
      WHERE name LIKE "%(%)"
   ) AS tmp
);

UPDATE movies SET `name` =
   RTRIM(REVERSE(SUBSTRING(REVERSE(`name`),
   LOCATE(" ",REVERSE(`name`)))))
WHERE `name` IN (
   SELECT `name`
   FROM (
      SELECT `name`
      FROM movies
      WHERE name LIKE "%, The"
   ) AS tmp
);

UPDATE movies SET `name` = CONCAT("The ",
   `name`)
WHERE `name` IN (
   SELECT `name`
   FROM (
      SELECT `name`
      FROM movies
```

```
        WHERE name LIKE "%,"
    ) AS tmp
);

UPDATE movies SET `name` = SUBSTRING(`name`,
    1, CHAR_LENGTH(`name`) - 1)
WHERE `name` IN (
    SELECT `name`
    FROM (
        SELECT `name`
        FROM movies
        WHERE name LIKE "%,"
    ) AS tmp
);
```

The script in Listing 9 consists of four queries executed sequentially: *(i)* removing the last word in strings where the last word begins and ends with parentheses (effectively removing the year), *(ii)* removing the last word in strings where the last word is an article preceded by a comma (effectively removing the article), *(iii)* prepending this removed article to strings where the last character is a comma (since the second query only removed the article but not the comma), and *(iv)* removing the last character (comma) of strings where the last character is a comma.

The pipeline for this preliminary data wrangling consists of two types of blocks: a `GenerateFlowFile` processor and three consecutive `PutSQL` processors. The three SQL `UPDATE` queries were placed inside the `PutSQL` processors. However, since this processor required incoming flow files (these flow files should contain the query to be executed although this can be overridden via the `SQL Statement` parameter [4], as is the case here), it was necessary for the first `PutSQL` block to be preceded by a `GenerateFlowFile` block, which was used to generate flow files with random content [5]. The pipeline is illustrated in Figure 7.

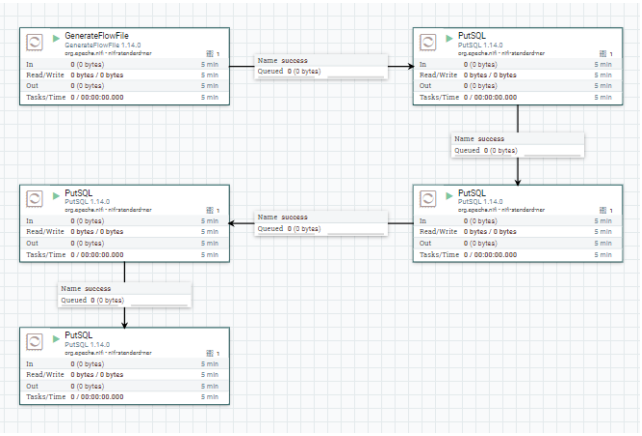

Figure 7. ETL Pipeline for Standardizing Movie Titles

3.2.2 Mapping Movie IDs

Similar to how the source dataset IMDb IJS featured mapping tables, a temporary mapping table was also created to store the mapping between the movie IDs in the primary dataset (IMDb IJS) and in the auxiliary dataset (IMDb Supplementary). Left join was used in order to prevent the loss of entries from the primary dataset, alongside case-insensitive matching to accommodate for varying data encoding schemes.

The utility of this mapping table is derived from Listing 6 and will be evident in the Join operations involving IMDb Supplementary's `imdb_ratings` table. The script for the actual table creation is shown in Listing 10.

Listing 10. Temporary Mapping Table for Movie IDs

```
CREATE TABLE ijs2supplementary AS (
    SELECT DISTINCT movie_id, imdb_title_id
    FROM movies
    LEFT JOIN imdb_movies
        ON LOWER(movies.name) =
            LOWER(imdb_movies.original_title)
        AND movies.year =
            YEAR(imdb_movies.date_published)
);
```

3.3 Transferring to Data Warehouse

This section discusses the ETL script for transferring the primary and auxiliary data sources into the data warehouse following the star schema. Before the ETL per se, an empty schema was already created in MySQL following the dimensional model described in Section 2. The ETL pipeline is given in Figure 8.

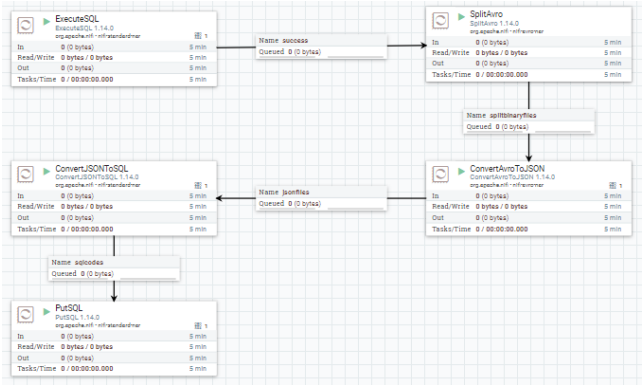

Figure 8. ETL Pipeline for Transferring to Data Warehouse

3.3.1 Extract

Unlike in the creation of local copies (Section 3.1.1), where the `QueryDatabaseTable` processor was used to extract from the source dataset, the more powerful `ExecuteSQL` processor was employed since the `SELECT` statement involves joining tables.

In order to merge IMDb IJS' `movies` and IMDb Supplementary's `imdb_movies`, an SQL script was written to perform a left join based on their titles (names) and year of release. Left join was used in order to prevent loss of entries from the primary dataset (i.e., IMDb IJS).

Listing 11 presents the SQL query placed inside the `ExecuteSQL` processor that is part of the ETL pipeline for the `movies` table.

Listing 11. Extraction Query for `movies` Table

```
SELECT DISTINCT movie_id, `name`, `year`,
    `rank`, title as other_name, duration,
    `language`, production_company,
    `description`, votes, budget,
    usa_gross_income,
    worlwide_gross_income AS
    worldwide_gross_income, metascore,
    reviews_from_users, reviews_from_critics
FROM movies
LEFT JOIN imdb_movies
```

```
    ON LOWER(movies.name) =
       LOWER(imdb_movies.original_title)
    AND movies.year =
       YEAR(imdb_movies.date_published);
```

The total ratings and gross earnings of the directors are sourced from an external dataset (IMDb Full); note that only these two attributes were taken from this dataset. The pertinent SQL command is provided in Listing 12.

Listing 12. Extraction Query for `directors` Table

```
SELECT director_id, first_name, last_name,
   rate, gross
FROM directors LEFT JOIN imdb_full_directors d
ON directors.first_name = d.first_name
AND directors.last_name = d.last_name;
```

The queries for the `actors`, `roles`, and `ratings` tables are straightforward `SELECT` statements (Listing 13 to 15).

Listing 13. Extraction Query for `actors` Table

```
SELECT actor_id, first_name, last_name, gender
FROM actors;
```

Listing 14. Extraction Query for `roles` Table

```
SELECT actor_id, movie_id
FROM roles;
```

Listing 15. Extraction Query for `ratings` Table

```
SELECT total_votes AS
   total_num_vote, votes_10, votes_9, votes_8,
   votes_7, votes_6, votes_5, votes_4,
   votes_3, votes_2, votes_1,
   allgenders_0age_avg_vote,
   allgenders_0age_votes,
   allgenders_18age_avg_vote,
   allgenders_18age_votes,
   allgenders_30age_avg_vote,
   allgenders_30age_votes,
   allgenders_45age_avg_vote,
   allgenders_45age_votes,
   males_allages_avg_vote,
   males_allages_votes, males_0age_avg_vote,
   males_0age_votes, males_18age_avg_vote,
   males_18age_votes, males_30age_avg_vote,
   males_30age_votes, males_45age_avg_vote,
   males_45age_votes,
   females_allages_avg_vote,
   females_allages_votes,
   females_0age_avg_vote, females_0age_votes,
   females_18age_avg_vote,
   females_18age_votes,
   females_30age_avg_vote,
   females_30age_votes,
   females_45age_avg_vote,
   females_45age_votes, top1000_voters_rating,
   top1000_voters_votes, us_voters_rating,
   us_voters_votes, non_us_voters_rating,
   non_us_voters_votes
FROM imdb_ratings;
```

The queries for the `genres` and `countries` table are analogous, but the result set should contain distinct entries since the intention of the schema design is to establish a bijection (i.e., a one-to-one relationship) between the surrogate key and a genre (for the `genres` table) and between the surrogate key and a country (for the `countries` table), as seen in Listings 16 and 17.

Listing 16. Extraction Query for `genres` Table

```
SELECT DISTINCT genre
FROM movies_genres;
```

Listing 17. Extraction Query for `countries` Table

```
SELECT DISTINCT country
FROM imdb_movies;
```

Finally, the extraction query for the fact table is characterized by multiple join operations (Listing 18) since this table serves as the consolidation of all the keys relating to the dimension tables. The number of join operations is also compounded by the need to refer to several mapping tables in the normalized source database.

Listing 18. Extraction Query for the Fact Table

```
SELECT movie_id, country_id, genre_id,
   director_id, actor_id, role_id, rating_id
FROM movies
LEFT JOIN (movies_directors
JOIN directors
   ON directors.director_id =
   movies_directors.director_id)
ON movies.movie_id =
      movies_directors.movie_id
LEFT JOIN (movies_actors
JOIN actors
   ON actors.actor_id =
      movies_actors.actor_id)
ON movies.movie_id =
   movies_actors.movie_id
LEFT JOIN genres
   ON movies.movie_id = movies_genres.movie_id
LEFT JOIN roles
   ON movies.movie_id = roles.movie_id
   AND actors.actor_id = roles.actor_id
LEFT JOIN (ijs2supplementary
JOIN imdb_movies
   ON ijs2supplementary.imdb_title_id =
      imdb_movies.imdb_title_id
JOIN imdb_ratings
   ON ijs2supplementary.imdb_title_id =
      imdb_ratings.imdb_title_id)
ON movies.movie_id =
   ijs2supplementary.movie_id;
```

3.3.2 Transform

After extracting the binary file, the `SplitAvro` processor was used to split the table into multiple binary files, one for each row. As Apache Nifi does not support direct Avro-to-SQL conversion, the `ConvertAvroToJSON` processor was used to first convert the AVRO files to the JSON format. Finally, the individual JSON files were then processed through the `ConvertJSONToSQL` processor, which was connected to the data warehouse using a JDBC connection pool in order to inform the data types of the data values in the JSON files.

3.3.3 Load

After undergoing the data wrangling and transformation processes, the data is loaded onto the data warehouse using the `PutSQL` processor. The processor is connected to the data

warehouse using a JDBC connection pool, and is used to run an `INSERT` statement for each table. As the `PutSQL` processor is used solely for loading in the pipeline, no further processing is done through its SQL statement; all data in the table are inserted into the data warehouse.

3.4 Data Quality and Freshness

To ensure the data quality and the correctness of the ETL script, both manual and automated tests were set in place, based on the suite of tests proposed by Yaddow [48]. These are discussed in detail in the Results and Analysis, specifically in Section 6.1.1. (Functional Testing of the ETL Script).

In order to ensure the freshness of the data being analyzed, the data must be updated as needed. In order to do so, data must also be regularly read and extracted from their respective sources. The scheduling option of the Apache NiFi processors was used to automate the operations of the processors.

In order to ensure that data is read and loaded into the data warehouse as soon as there are relevant updates in the data sources, the event-driven scheduling strategy can be used; however, this strategy has yet to be fully supported by Apache NiFi [9], and not all processors (including the `GetFile` and `QueryDatabaseTable` processors used for extracting data for the data warehouse) provide event-driven scheduling.

Thus, the timer-driven strategy was employed in a manner that strongly mimicked the event-driven strategy; specifically, the run schedule of the processors was set to 0 seconds in order to have them run as long as they have data to process. Despite the additional cost this requires as compared to scheduled or periodic data retrieval, this ensures that data is updated and stored in the data warehouse in real time. While this behavior can also be replicated using CRON-driven scheduling, the authors chose timer-driven scheduling in order to minimize the complexity of the scheduling configurations [9].

Moreover, since all the processes involved in the ETL are part of the pipeline (including the execution of Python scripts used for the preliminary data cleaning), they allow for progressive updates and loading of movies.

3.5 Data Warehouse Volume

After performing all the extraction, transformation, and loading steps discussed in this section, an increase in data volume can be observed as a result of collapsing many-to-many relationships. As explained in Section 2.3, utilizing this denormalization strategy and increasing table sizes are trade-offs to decrease the cost of join operations. Table 4 summarizes the number of rows for each of the tables comprising the data warehouse.

3.6 Issues Encountered

Some issues were encountered in view of the volume of the data that was extracted, transformed, and loaded into the warehouse.

3.6.1 NiFi and Server Configuration

Aside from tweaking the configuration settings in Apache NiFi, some MySQL server variables were also modified in order to speed up the join operations, especially since most of the tables in the source datasets had hundreds of thousands of rows. Not performing these optimizations resulted in out-of-memory errors or in the MySQL server losing connection before the completion of the requested query.

Table 4. Number of Rows of the Tables in the Data Warehouse

Name	Number of Rows
`fact_table`	6,617,850
`actors`	815,842
`countries`	184
`directors`	85,731
`genres`	21
`movies`	381,762
`ratings`	35,428
`roles`	2,513,767

For the SQL-to-SQL ETL process performed for the data in IMDb IJS, the primary issue was the bottleneck created by the pipeline when extracting large volumes of data. As the `QueryDatabaseTable` processor originally only created one binary file containing all the extracted data, the succeeding processor, `SplitAvro`, would take much longer to finish and provide flow files, leaving the succeeding processors idle.

Moreover, after the SplitAvro processor completes the data splitting, the succeeding processors are inundated with data; in particular, the processor often creates a large number of files to the point that the queue of files for the ConvertAvroToJSON processor would be completely full.

Although adjusting the object thresholds remedies the problem regarding queue capacity, this does not address the bottleneck at the `SplitAvro` processor. In order to better distribute the work among the pipeline processors, the results of the QueryDatabaseTable processor were split into multiple files of 10000 rows each; with this setup, after splitting the first binary file, `SplitAvro` outputs a smaller batch of 10000 objects to the `ConvertAvroToJSON` processor and allows both processors to work in parallel.

For the CSV-to-SQL ETL process, however, the bottleneck created by the SplitText processor became a major issue, causing the team to get out-of-memory errors. Splitting the CSV files directly into single-row entries proved to require immense storage space.

In order to minimize the computational requirements of the pipeline and be able to extract the data, a series of SplitText processors was used in place of the single processor. As mentioned in Chapter 3.1.3 of this report, the file was split into batches of 10000, then 100, then single rows, which lessened the workload on the remaining faculty members and allowed multiple processors to work in parallel.

In order to optimize the performance of join operations, it was also necessary to tweak the values of certain MySQL and InnoDB variables related to buffering and caching (InnoDB is the storage engine of MySQL). Table 5 summarizes these changes.

Without these optimizations to the global variables of MySQL, the join operations for constructing the fact table took over three hours to complete, and there were repeated instances where the server itself lost connection in the middle of query execution.

Table 5. Tables of IMDb Supplementary

Variable	Description	Old Value	New Value
`innodb_log_buffer_size`	Size (in bytes) of the buffer that InnoDB uses to write to the log files on disk. Increasing this saves disk I/O [33].	1M	256M
`innodb_buffer_pool_size`	Size (in bytes) of the buffer pool. Larger values allow for less disk I/O when constantly accessing the same table data [33].	8M	5G
`innodb_log_file_size`	Size (in bytes) of each log file in a log group. Larger values save disk I/O by requiring less checkpoint flush activity in the buffer pool [33].	48M	5G
`join_buffer_size`	Minimum size of buffer used for plain index scans, range index scans, and joins without indexes (full table scans). Larger values improve performance by allowing more sequential access to the right-hand table of a join operation [35].	256K	4M
`sort_buffer_size`	Size of buffer used for any sort in a session. Increasing value speeds up `ORDER BY` and `GROUP BY` operations [35].	256K	4M
`read_buffer_size`	Size of buffer allocated for sequential scans for a `MyISAM` table. Increased value also increases performance for more sequential scans [35].	64K	4M
`key_buffer_size`	Also known as key cache, is the size of the buffer for index blocks for `MyISAM` tables. Increasing the value allows for better index handling on reads and writes, but cannot be extremely large in order to leave room for file system cache [35].	8M	64M

3.6.2 Data Wrangling for Foreign Titles

Section 3.2.1 discussed the additional data wrangling aimed at the standardization of the movie titles. Albeit the transposition of the position of the grammatical article seemed straightforward, one of the major complications was extending it to titles that are not in English; an example would be standardizing the French title `Vie en Rose, La` to `La Vie en rose`.

Table 6 shows the non-English grammatical articles that were also considered in the preliminary data wrangling.

Table 6. Non-English Grammatical Articles

Language	Grammatical Articles
Spanish	*el, la, los, las, un, una, unos, unas*
German	*der, den, dem, des, die, das, ein, eines, einem, eine, einer*
French	*le, les, la, l'* (special case: no space after the apostrophe when it is prepended)
Italian	*il, i, lo, gli, l'* (special case: no space after the apostrophe when it is prepended), *gli, la, le, un, uno, una, un'* (special case: no space after the apostrophe when it is prepended)
Portuguese	*o, os, a, as*
Turkish	*bir, bazi, az*
Filipino	*ang*
Polish	*jeden, kilka, malo, niewiele*
Danish	*en, et*
Norwegian	*en, ei, et*
Finnish	*yksi, joitakin, muutamat*
Romanian	*l, ul, e, le, a, un, o, niste*
Dutch	*de, het*

4. OLAP Application

This section focuses on the decision-making and analytical tasks for which this project is intended, alongside the specific reports that were generated as part of the OLAP application. The final data wrangling steps for the use cases in this project are also discussed, and screenshots of the interactive visualizations are presented to enrich the discussion.

4.1 Purpose of the Application

The application is an interactive dashboard that aims to provide insight into the movie industry through reports on the IMDb dataset. In particular, the application offers information regarding viewer feedback on movie genres, popular movie genres, and the most prolific actors, companies, and directors, among others. Moreover, the application is active in that users can toggle the amount of information they see and edit the filters used for the data visualization.

The application is intended for users to obtain a more informed view of the movie industry, which, depending on their roles and

responsibilities, may affect their succeeding decisions and even the direction of the movie industry as a whole. For example, a casual moviegoer who learns about the number of horror movies produced over the years may grow curious about periods of time wherein a plethora of horror movies were produced and seek out new films they have not seen before. Meanwhile, a casting director who is presented with the top actors for a genre may decide to contact these actors for an upcoming project. On the other hand, an actor or other crew member who has been recruited for a job with a production company may decide to accept the offer after seeing that the company has produced many films that have ranked highly and are well-received by audiences.

The application aims for such decisions to be informed by its generated data visualizations; samples of these visualizations and their interpretations can be seen in Section 5 of this report.

4.2 ETL Pipeline

The ETL pipeline is explained in detail in Section 3.

4.3 Final Data Wrangling

Integrated in the ETL phase are some preliminary data wrangling steps, which were necessary to ensure the rectitude of the Join operations and which had fundamental repercussions on the data representation during the transfer to the warehouse. In particular, it may be recalled that:

- The dates were standardized during the transformation phase that was part of the ETL for creating local copies of the datasets.
- The countries of origin and genres were exploded into separate rows during the transformation phase that was part of the ETL for creating local copies of the datasets.
- The titles of the movies were standardized prior to the ETL for transferring data into the data warehouse.

Completing the data wrangling phase is the standardization of the names of the production companies. Since it had no impact on the ETL pipeline and was only relevant to the OLAP application (particularly to the use case involving analytics on the production companies), it was deferred to this phase of the project.

Performing data cleaning on the production companies was deemed necessary since it included values that were essentially similar but were encoded with different descriptors. More specifically, most of the listed production companies had a country modifier enclosed in square brackets after the company name, for instance, `Canal+ [be]`. This format prevented the use of the aggregate function `GROUP BY` to cluster the entries since they are recognized as different strings.

In light of the aforementioned concern, substrings containing strictly production company names were generated for each row with `[` as the delimiter for separation. In performing this operation, conflicting company names such as `Canal+ [be]` and `Canal+ [fr]` were standardized into simply `Canal+`. Following this, some of the data for production company names were observed to contain further additional descriptions indicated by a `-` followed by the details contained within parentheses. To address this, another substring was generated from the newly encoded production company names with the delimiter `- (` as the indicator for extraction. As a result, entries such as `Allied Filmmakers - (present)` and `Allied Filmmakers - (in association with)` were standardized into `Allied Filmmakers`.

Note that, since the entries in data warehouses are nonvolatile, the names of the production companies that were already stored in the warehouse should not be modified. Therefore, a separate column containing the standardized names of the production companies was added solely for the analytics.

Listing 19. Script for Standardizing `prodcompanies` Names

```
ALTER TABLE movies
ADD production_company_clean VARCHAR(45);

UPDATE movies SET production_company_clean =
   (SELECT SUBSTRING_INDEX(production_company,
        '- (', 1)
    FROM (
       SELECT
       SUBSTRING_INDEX(production_company,
          ' [', 1)
       FROM movies));
```

The script listed above shows a subquery to extract a substring from the `production_company` column using the `SUBSTRING_INDEX` function. This function returns a substring of a string before a specified number of delimiters occurs. Since the data was observed to only contain at most one instance of an open square bracket for each row, the `SUBSTRING_INDEX` function was set to truncate the string before the first occurrence of the delimiter `[`. Following the subquery, an outer query was configured to perform an almost identical operation. The only difference was that the outer query's delimiter was set to truncate on the first instance of `- (` instead.

After which, the cleaned values were inserted into the newly-added `production_company_clean` column of the `movies` table, concluding the data wrangling for the names of the production companies in the dataset.

4.4 Generated Reports

This section presents the OLAP reports that were generated. In total, the authors identified seven (7) use cases that pertain to the movie industry and utilize or combine different OLAP operations, such as roll-up, drill-down, slice, dice, and pivot. The significance of each of these OLAP applications is discussed in Section 5 (Query Processing and Optimization). To enrich the discussion, screenshots of their interactive visualizations (created using the software Tableau Public) are also given in Figures 9 to 14.

4.4.1 For each genre, what is the total number of movie votes per year and decade?

This query shows the number of votes for each category (i.e., 10, 9, 8, and so on) as well as the total number of votes (i.e., the sum of votes for each category) for every decade, year, and genre available in the data. Apart from the itemized data for each genre, the generated report also contains aggregate values showing the number of votes for all genres in a given year, for all years in a given decade, and for all decades available in the data.

The roll-up operation was used for this query in order to generate the aggregate values for the broader classifications (i.e., decade and year) of the data. Alternatively, when viewed from the top down, the query also makes use of a drill-down operation in that for each decade, the number of votes for each year and genre can also be determined.

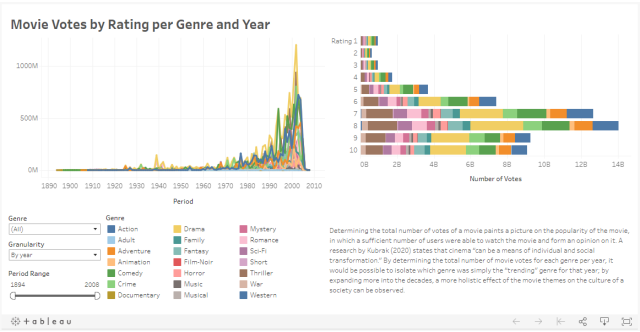

Figure 9. Tableau Dashboard for Query 4.4.1

4.4.2 For each year of each decade, what are the total number of male and female votes per genre?

This query shows the total number of male and female votes, as well as the total number of votes for both genders, for a given decade. Moreover, the query also generates more detailed breakdowns of the data, such as the number of votes per year in a given decade or the number of votes per genre in a given year.

This query made use of the drill-down operation in order to provide the breakdown of results after the decade level. However, when viewed from the bottom up, the query can also be said to make use of a roll-up operation, as it aggregates the number of votes of all genres in a year and all years in a decade.

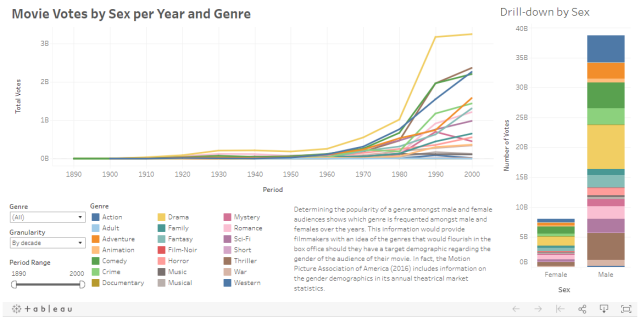

Figure 10. Tableau Dashboard for Query 4.4.2

4.4.3 Who are the top 15 actors with the most number of movie appearances for the action genre?

This query shows the top actors with the highest numbers of credited action movies in the data. Apart from their names, the number of action movies they have participated in and their ranks among the top 15 have also been displayed.

This query uses the slice operation in two ways. First, the limiting of movies to a certain genre required the type of filtering indicative of a slice operation. Moreover, the limiting of entries to the top 15 also requires a slice in that all actors ranked below 15 are excluded from the query results.

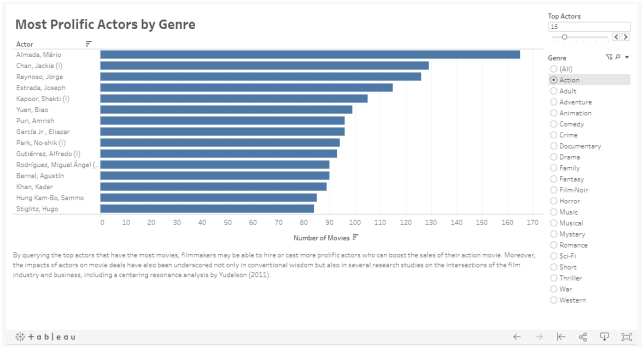

Figure 11. Tableau Dashboard for Query 4.4.3

4.4.4 For each country, what are the top 10 movie genres?

This query shows the top movie genres for each country based on the total number of movies released in the country for each genre. Additionally, the ranks of each genre among the top 10 are also displayed.

The query makes use of the slice operation, albeit on a derived attribute, by first assigning ranks to each genre within a country based on the number of movies in that genre that have been released in the country, before removing all genres that are not part of the top 10 from the query results.

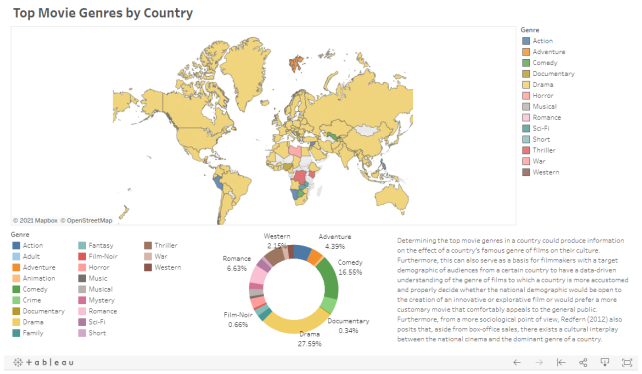

Figure 12. Tableau Dashboard for Query 4.4.4

4.4.5 How many horror movies are released in the USA for each year and decade?

This query shows the number of horror movies that have been released in the USA for each year and decade available in the data. Additionally, the query also shows the total number of movies released for all years in a given decade, as well as the total number of movies for all decades in the data.

The query makes use of a dice operation to filter the movie data according to their genre and country (displaying data for horror and the USA, respectively). Moreover, the query also uses the roll-up operation to obtain the aggregate number of horror movies released in the USA per decade, while from a top-down perspective, the query uses a drill-down operation to display the number of horror movies released in the USA for all years in a given decade.

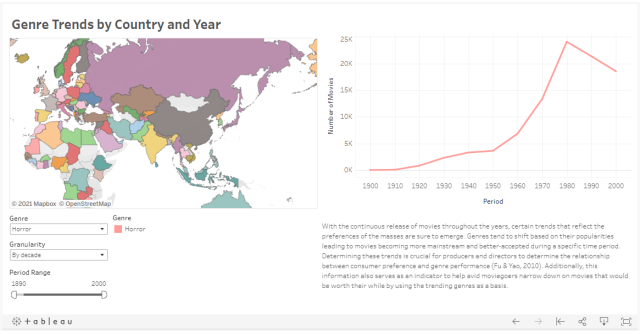

Figure 13. Tableau Dashboard for Query 4.4.5

4.4.6 What are the production companies that have released over 10 movies with a rank above the average rank and a number of votes above the average total number of votes?

This query first computes the average rank and average total number of votes across all movies, before showing the list of production companies that have released over 10 movies that meet two conditions: their rank is greater than the average rank, and they have a total number of votes greater than the average number.

The query uses a dice operation in order to filter the results according to their ranks and total numbers of votes. It is worth noting, however, that the dice operation has been applied to produce the result of a subquery; the query itself also makes use of a slice operation in order to filter the production companies according to the number of movies they have released.

Figure 14. Tableau Dashboard for Query 4.4.6

4.4.7 For each of the top 6 directors with the highest gross earnings, what are their 15 highest-rated movies?

This query first determines the top six directors according to their gross earnings before displaying the top 15 highest-rated movies of each of the directors.

The query makes use of the pivot operation by using the results of the initial query (i.e., the portion of the query producing the list of highest-grossing directors) as qualifiers for the succeeding query (i.e., the portion of the query producing the list of highest-rated movies per director). Moreover, the query also applies the slice operation on derived attributes by filtering the query results according to the ranks of the directors, then according to the ranks of their movies.

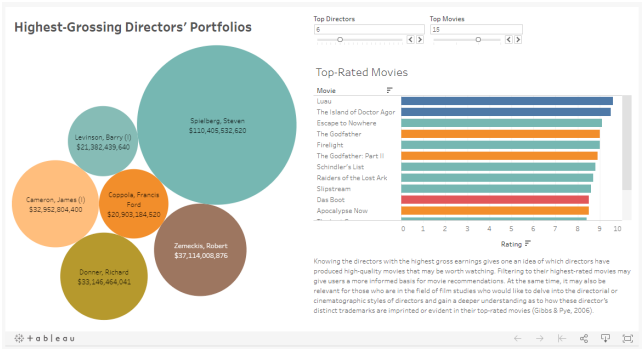

Figure 15. Tableau Dashboard for Query 4.4.7

5. Query Processing and Optimization

The previous sections have already described some optimizations that were implemented to improve the performance of the data warehouse operations. In particular, during the ETL phase, the size of the buffer and cache tables, as well as other system variables pertinent to the MySQL server, were increased primarily to speed up the execution of join operations (Section 3.5.1).

In examining the performance of the queries and optimizing them, the following order of experiments was followed:

- Running the query on the normalized database (i.e., the database resulting from creating the local copies of the datasets, as explained in Section 3.1.1). This normalized database uses the same indexes as in the original dataset.
- ***Schema optimization.*** Running the query on the actual data warehouse, which is denormalized following a star schema. However, there are no indexes in place (except for those that are mandated by MySQL, namely, the indexes for auto-incrementing surrogate keys).
- ***Index optimization.*** Running the query on the actual data warehouse, with indexes added on the attributes involved in the join operations, namely, the keys in the fact table and the primary keys in the dimension tables.
- ***Another round of index optimization.*** Running the query on the actual data warehouse.
- ***Query optimization*** **(if possible).** Running a revised, more optimized, and equivalent version of the query.

Each experiment was run for a total of 10 times, and the execution time reported refers to the average of the results (in order to statistically account for the effect of caching and other related under-the-hood optimizations automatically employed by MySQL to expedite the processing of queries after their initial run [29]).

5.1 For each genre, what is the total number of movie votes per year and decade?

Determining the number of votes of a movie paints a picture of its popularity, in which a sufficient number of users were able to watch the movie and form an opinion on it. A study done by Kubrak [2020] states that cinema "can be a means of individual and social transformation." By determining the total number of movie votes for each genre per year, it would be possible to isolate which genre was simply the "trending" genre for that year; by expanding more into the decades, a more holistic effect of the movie themes on the culture of a society can be observed.

5.1.1 Output

The resulting output has 1992 rows and 14 columns. An excerpt is given in Figure 16.

decade_released	year	genre	total_votes	votes_10	votes_9	votes_8	votes_7	votes_6	votes_5
NULL	NULL	NULL	60390538630	9063770405	9252537587	14024663182	12659453493	7400999751	3683119834
1890s	NULL	NULL	296433	31978	36868	80723	71746	36149	15923
1890s	1894	NULL	3056	184	138	474	1088	722	280
1890s	1894	Documentary	1528	92	69	237	544	361	140
1890s	1894	Short	1528	92	69	237	544	361	140
1890s	1895	NULL	16566	1128	822	2010	4572	4344	1950
1890s	1895	Drama	8283	564	411	1005	2286	2172	975
1890s	1895	Short	8283	564	411	1005	2286	2172	975
1890s	1896	NULL	25368	2280	1906	4457	6584	5074	2544
1890s	1896	Documentary	10334	784	523	1295	2250	2512	1466
1890s	1896	Drama	1036	42	39	74	222	316	200
1890s	1896	Short	13998	1454	1344	3088	4112	2246	878
1890s	1897	NULL	83962	9582	11298	21875	20238	10846	4782
1890s	1897	Comedy	9182	551	405	1347	2609	2115	953
1890s	1897	Crime	156	10	14	16	36	52	18
1890s	1897	Documentary	8382	571	601	1410	2109	1637	833
1890s	1897	Drama	32642	4196	5168	9481	7548	3380	1448
1890s	1897	Short	33600	4254	5110	9621	7936	3662	1530
1890s	1898	NULL	153283	17509	22476	51391	38376	13876	4820

Figure 16. Output of Query 5.1

5.1.2 Optimization and Statistics

Generating the OLAP report using a normalized database (Listing 20) took 23.400 seconds.

Listing 20. SQL Statement for Query 5.1 (Normalized)

```
SELECT CONCAT(FLOOR(m.year/10) * 10, 's') AS
   decade_released, m.year, g.genre,
   (SUM(r.votes_10) + SUM(r.votes_9) +
   SUM(r.votes_8) + SUM(r.votes_7) +
   SUM(r.votes_6) + SUM(r.votes_5) +
   SUM(r.votes_4) + SUM(r.votes_3) +
   SUM(r.votes_2) + SUM(r.votes_1)) AS
   total_num_votes,
   SUM(r.votes_10) AS votes_10, SUM(r.votes_9)
   AS votes_9, SUM(r.votes_8) AS votes_8,
   SUM(r.votes_7) AS votes_7, SUM(r.votes_6) AS
   votes_6, SUM(r.votes_5) AS votes_5,
   SUM(r.votes_4) AS votes_4, SUM(r.votes_3) AS
   votes_3, SUM(r.votes_2) AS votes_2,
   SUM(r.votes_1) AS votes_1
FROM movies m
JOIN genres g ON g.movie_id = m.movie_id
JOIN ijs2supplementary i ON i.movie_id =
   m.movie_id
JOIN imdb_ratings r ON r.imdb_title_id =
   i.imdb_title_id
GROUP BY decade_released, m.year, g.genre
   WITH ROLLUP
ORDER BY decade_released, m.year, g.genre;
```

As seen in Figure 17, the execution plan involves performing a full table scan on the `ijs2supplementary` mapping table, unique key lookup (via the primary key indexes) on both `movies` and `ratings`, and non-unique key lookup on `genres`. The tables were subsequently joined via the nested-loop join algorithm. The optimizer's decision to perform a full table scan on the mapping table can be attributed to the fact that it has the lowest cardinality among the tables.

The resulting joined table was then buffered into a temporary table before being grouped and sorted. The lack of a suitable index for grouping and sorting prompted the optimizer to use the `filesort` algorithm. However, `filesort` incurs additional space and time requirements, and may even result in the usage of temporary disk files should the main memory be insufficient [34].

It may be interesting to note that buffering the resulting joined table is necessary since roll-up involves calculating the subtotals per dimension. Without the roll-up operation, the temporary table is also eliminated from the query execution plan.

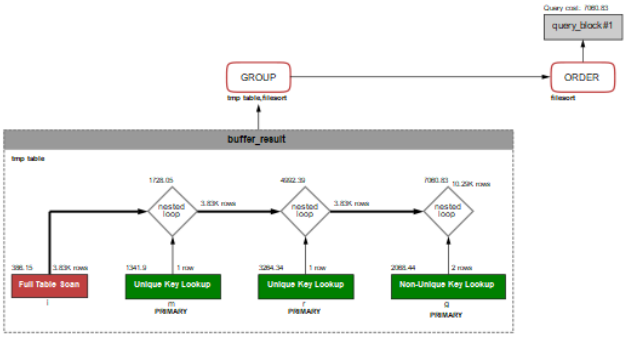

Figure 17. Query Execution Plan of Query 5.1 (Normalized)

Meanwhile, generating the report using the denormalized data warehouse (Listing 21) albeit without indexes registered 9.344 seconds, clocking an improvement.

Listing 21. SQL Statement for Query 5.1 (Denormalized)

```
SELECT CONCAT(FLOOR(m.year/10) * 10, 's') AS
   decade_released, m.year, g.genre,
   SUM(r.total_num_votes) AS total_votes,
   SUM(r.votes_10) AS votes_10, SUM(r.votes_9)
   AS votes_9, SUM(r.votes_8) AS votes_8,
   SUM(r.votes_7) AS votes_7, SUM(r.votes_6) AS
   votes_6, SUM(r.votes_5) AS votes_5,
   SUM(r.votes_4) AS votes_4, SUM(r.votes_3) AS
   votes_3, SUM(r.votes_2) AS votes_2,
   SUM(r.votes_1) AS votes_1
FROM fact_table f
JOIN movies m ON m.movie_id = f.movie_id
JOIN ratings r ON r.rating_id = f.rating_id
JOIN genres g ON g.genre_id = f.genre_id
GROUP BY decade_released, m.year, g.genre
   WITH ROLLUP
ORDER BY decade_released, m.year, g.genre;
```

The absence of indexes prompted the optimizer to use full table scans on all four tables: the fact table, `movies`, `ratings`, and `genre` (Figure 18). Consequently, the hash join algorithm was also employed to avoid the significant slowdown of using a block nested-loop algorithm. Similar to the previous case, the resulting joined table was first buffered before applying `filesort` for grouping and sorting due to the absence of suitable indexes.

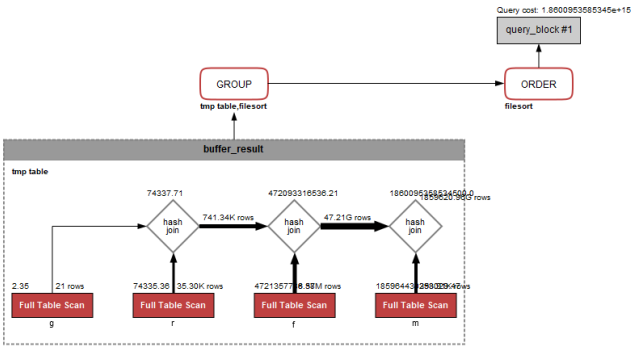

Figure 18. Query Execution Plan of Query 5.1 (Denormalized, Without Indexes)

Placing indexes on the attributes used in joining the tables led to an execution time of 15.453 seconds. As seen in Figure 19, the presence of indexes prompted the optimizer to perform only the single requisite full table scan on `ratings` and employ non-unique key lookup on the fact table and unique key lookup (via the primary key indexes) on both `genres` and `movies`. The joined table was buffered anew before applying `filesort`, indicating that the selected indexes were not suited for the grouping and sorting operations.

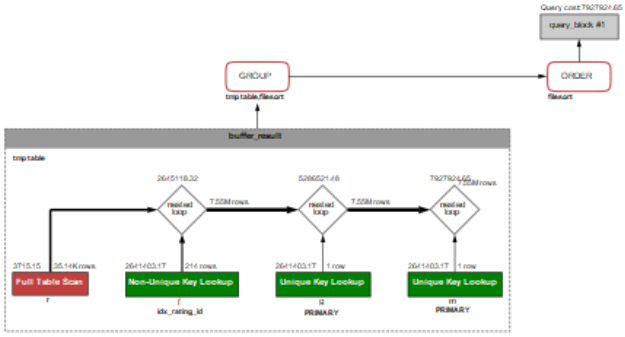

Figure 19. Query Execution Plan of Query 5.1 (Denormalized, With Indexes)

Interestingly, indexing led to an increase in the execution time. A possible explanation can be given by analyzing the nature of the query: even if the indexes were well selected, a query that involves processing almost all the rows of the tables — such as this roll-up operation — may benefit from the sequential access strategy of a full table scan, as noted by IBM [16].

Related to this, full table scans also maximize the capacity of the system to perform multiblock reads, which take advantage of concurrent I/O [15]. On the other hand, seeks are not designed for this type of concurrent optimization, and there are cases wherein the distribution of data may actually result in several inefficient B^+-tree traversals. B^+-tree is the data structure used by MySQL to store its indexes [36]. This inefficiency resulting from the number of tree traversals may be exacerbated by the optimizer's decision to use nested-loop join [47].

Creating a composite index for the attributes involved in the join, group, and order operations registered an increase in the execution time: 16.516 seconds.

It follows the same query execution plan as that of the indexed version and still uses `filesort`; this may be taken as an indicator that the optimizer finds it to be more efficient compared to using the created composite index.

However, a key difference is that the creation of the composite index changed the leftmost block to a full index scan (as opposed to a full table scan), as seen in Figure 20.

Full index scans are supposed to be faster compared to full table scans. However, this highlights the fact that the MySQL optimizer is not fully cognizant of the type of secondary disk storage [10]. In hard disk drives — such as the device used by the authors in conducting the experiments — full index scans may become slower since they feature random access, which is bogged down by disk rotation latency. On the other hand, full table scans are sequential, thus only requiring the initial disk seek.

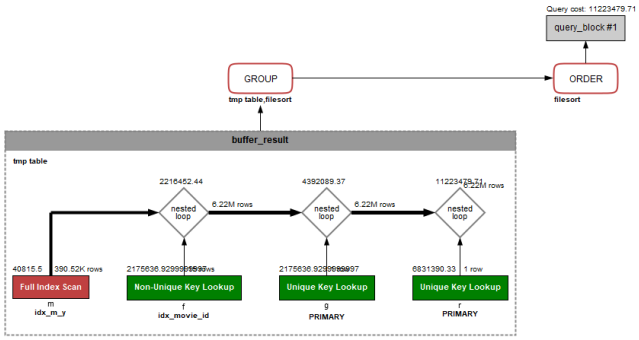

Figure 20. Query Execution Plan of Query 5.1 (Denormalized, With Composite Indexes)

5.2 For each year of each decade, what are the total number of male and female votes per genre?

Determining the popularity of a genre among male and female audiences shows which genre is frequented among male and females over the years. This information would provide filmmakers with an idea of the genres that would flourish in the box office should they have a target demographic regarding the gender of the audience of their movie. In fact, the Motion Picture Association of America [26] includes information on the gender demographics in its annual theatrical market statistics.

5.2.1 Output

The resulting output has 1992 rows and 6 columns. An excerpt is given in Figure 21.

decade_released	year	genre	total_votes	votes_male	votes_female
NULL	NULL	NULL	60390538630	38694163614	8093541206
1890s	NULL	NULL	296433	183862	45811
1890s	1894	NULL	3056	1862	526
1890s	1894	Documentary	1528	931	263
1890s	1894	Short	1528	931	263
1890s	1895	NULL	16566	10836	2256
1890s	1895	Drama	8283	5418	1128
1890s	1895	Short	8283	5418	1128
1890s	1896	NULL	25368	14872	4524
1890s	1896	Documentary	10334	6131	1899

Figure 21. Output of Query 5.2

5.2.2 Optimization and Statistics

MySQL does not have a built-in drill-down operation. However, it can be simulated with the use of `WITH ROLLUP`. After all, drill-down and roll-up differ only in the direction of the granularity of the desired report. The former aims for an increased or more granular level of detail, whereas the latter results in a less granular analysis that focuses on subtotals and grand totals.

Generating an OLAP report using a normalized database (Listing 22) took 12.5 seconds.

Listing 22. SQL Statement for Query 5.2 (Normalized)

```
SELECT CONCAT(FLOOR(m.year/10) * 10, 's') AS
   decade_released, m.year, g.genre,
   (SUM(r.votes_10) + SUM(r.votes_9) +
   SUM(r.votes_8) +
   SUM(r.votes_7) + SUM(r.votes_6) +
   SUM(r.votes_5) +
   SUM(r.votes_4) + SUM(r.votes_3) +
   SUM(r.votes_2) +
   SUM(r.votes_1)) AS total_num_votes,
```

```
   SUM(r.males_allages_votes) AS votes_male,
   SUM(r.females_allages_votes) AS votes_female
FROM movies m
JOIN genres g ON g.movie_id = m.movie_id
JOIN ijs2supplementary i ON i.movie_id =
   m.movie_id
JOIN imdb_ratings r ON r.imdb_title_id =
   i.imdb_title_id
GROUP BY decade_released, m.year, g.genre
   WITH ROLLUP
ORDER BY decade_released, m.year, g.genre;
```

The query execution plan in Figure 22 shows that a full table scan is performed on the `ijs2supplementary` mapping table. Similar to the reasoning presented in Section 5.1.2, this joining strategy employed by MySQL usually pushes the table with the smallest cardinality as the leftmost operand. This is followed by unique key lookup (via the primary key indexes) on both `movies` and `ratings`, and non-unique key lookup on `genres`.

Since roll-up involves aggregating data into subtotals, the joined table was buffered prior to grouping and sorting. With the optimizer not finding any suitable index that can help expedite grouping and sorting, it resorted to using `filesort`.

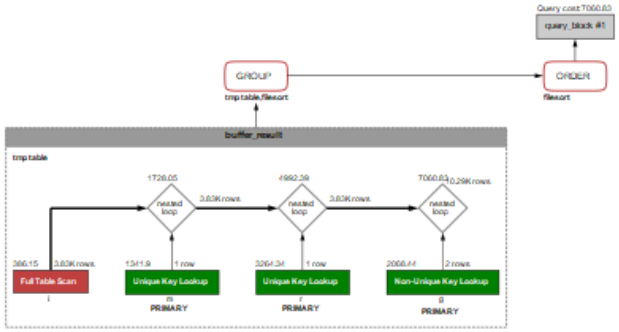

Figure 22. Query Execution Plan of Query 5.2 (Normalized)

On the other hand, using the denormalized data warehouse (Listing 23) albeit without indexes registered an execution time of 9.359 seconds, signifying an improvement. Note that the index on `genres` cannot be removed since MySQL mandates indexes for auto-incrementing surrogate keys.

Listing 23. SQL Statement for Query 5.2 (Denormalized)

```
SELECT CONCAT(FLOOR(m.year/10) * 10, 's') AS
   decade_released, m.year, g.genre,
   SUM(r.total_num_votes) AS total_votes,
   SUM(r.males_allages_votes) AS votes_male,
   SUM(r.females_allages_votes) AS votes_female
FROM fact_table f
JOIN movies m ON m.movie_id = f.movie_id
JOIN genres g ON g.genre_id = f.genre_id
JOIN ratings r ON r.rating_id = f.rating_id
GROUP BY decade_released, m.year, g.genre
   WITH ROLLUP
ORDER BY decade_released, m.year, g.genre;
```

Figure 23 presents an interesting case of combining multiple join algorithms in one query. The fact table, `movies`, and `ratings` were subjected to full table scans; however, the optimizer took advantage of the surrogate key index on `genres` to perform a unique key lookup instead. Consequently, only the scanned tables had to be joined via the hash join algorithm. The reduction in the number of rows returned by a unique key lookup made it possible to employ a nested-loop join algorithm, which is noted to be faster than a hash join if a join operand has a small cardinality since it discards the need to build a hash table [2].

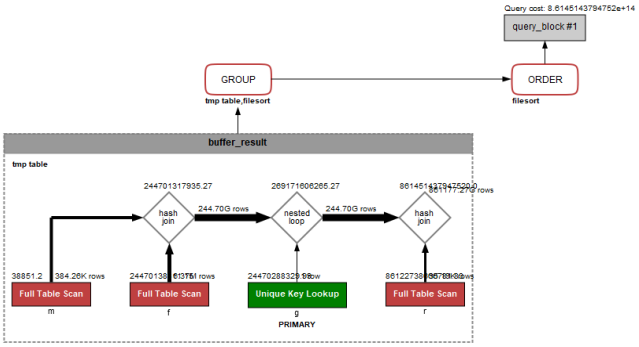

Figure 23. Query Execution Plan of Query 5.2 (Denormalized, Without Indexes Except for Surrogate Key)

Setting up indexes on the attributes in the join operations resulted in a slower execution time: 12.656 seconds. The query execution plan in Figure 24 shows that the optimizer used the index on the fact table to perform a non-unique key lookup and the primary key indexes on genres and movies to perform unique key lookups. Therefore, only the `ratings` table was fully scanned. Since the result sets of seeks have smaller cardinalities, nested-loop join can be applied. Note that nested-loop join consumes less memory resources; however, this saved space may have already been offset by the resources needed for the B$^+$-trees storing the indexes.

The joined table was buffered before applying `filesort`, indicating that the selected indexes were not suited for the grouping and sorting operations (similar to the previous roll-up query).

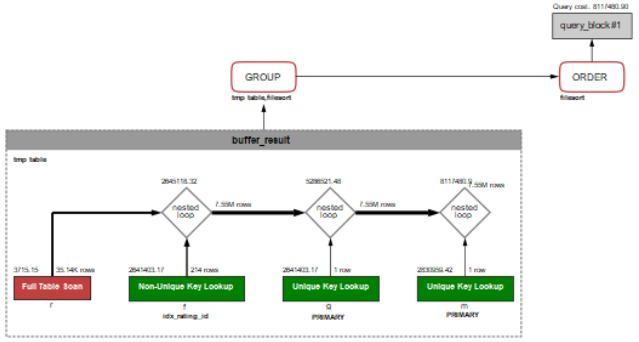

Figure 24. Query Execution Plan of Query 5.2 (Denormalized, With Indexes)

As explained in Section 5.1.2, the slowing down of query processing with the addition of indexes may be ascribed to the distribution of data and the fact that drill-down involves reading through almost all the rows of the tables. In this regard, the multiblock reads and sequential access scheme of full table scans may be more efficient compared to several non-concurrent and repeated B$^+$-tree traversals when indexes are employed.

Using a composite index for the attributes involved in the join, group, and order operations brought the execution time to

13.375 seconds. The resulting query execution plan is the same as that of the indexed version, with the exception of applying a full index scan as opposed to a full table scan (Figure 25).

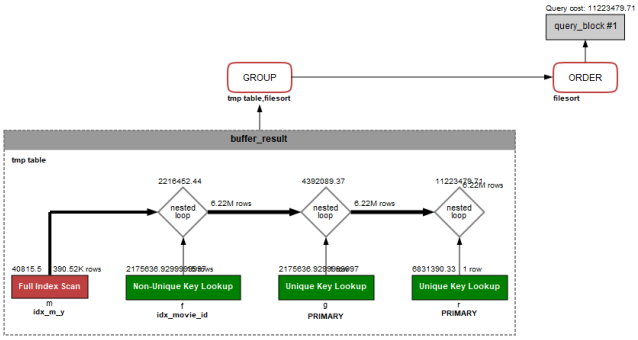

Figure 25. Query Execution Plan of Query 5.2 (Denormalized, With Composite Indexes)

Despite the theoretical advantage of using a full index scan, this does not translate into practice in the specific case of this query due to hardware bottlenecks. The random accesses that form part of a full index scan are expensive operations for hard disk drives, in contrast to the single disk seek and sequential access strategy of a full table scan. This explains the longer execution time when run in the machine of the authors.

5.3 Who are the top 15 actors with the most number of movie appearances for the action genre?

According to Statista [44], as of 2021, the action genre is the second most profitable genre with a total box office revenue of 47.93 million USD since 1995. Being an immensely popular movie genre, it is imperative that action movie filmmakers cast famous action movie actors that have high exposure to the public as part of the film's publicity and promotion. By querying the top 15 actors that have the most movies, filmmakers may be able to hire or cast more prolific actors who can boost the sales of their action movie. Moreover, the impacts of actors on movie deals have also been underscored not only in conventional wisdom but also in several research studies on the intersections of the film industry and business, including a centering resonance analysis by Yudelson [49].

5.3.1 Output

The resulting output has 15 rows and 2 columns. An excerpt is given in Figure 26.

name	Num_roles
Almada, Mário	165
Chan, Jackie (I)	126
Reynoso, Jorge	126
Estrada, Joseph	115
Kapoor, Shakti (I)	105
Yuen, Biao	99
Puri, Amrish	96
García Jr., Eliazar	96
Park, No-shik (I)	94
Bernal, Agustín	90

Figure 26. Output of Query 5.3

5.3.2 Optimization and Statistics

Generating the OLAP report using a normalized database (Listing 24) took 18.752 seconds.

Listing 24. SQL Statement for Query 5.3 (Normalized)

```
SELECT name, Num_roles
FROM (
   SELECT a.name, COUNT(m.movie_id)
      AS Num_roles,
      RANK() OVER
         (PARTITION BY g.genre
          ORDER BY COUNT(m.movie_id) DESC) AS
          Num_roles_rank
   FROM movies m
   JOIN genres g ON g.movie_id = m.movie_id
   JOIN roles i ON i.movie_id = m.movie_id
   JOIN actors a ON a.actor_id = i.actor_id
   WHERE g.genre = "Action"
   GROUP BY g.genre, a.name
) AS Role_rank
WHERE Num_roles_rank <= 15;
```

As seen in Figure 27, the execution plan for the inner subquery (excluding the `RANK` operation, which causes the Visual Explain Tool of MySQL Workbench to crash) consists of a full table scan on actors, followed by non-unique key lookup on roles and unique key lookups on both `movies` and `genres`. The outer query then employs a full table scan once the inner subquery is resolved.

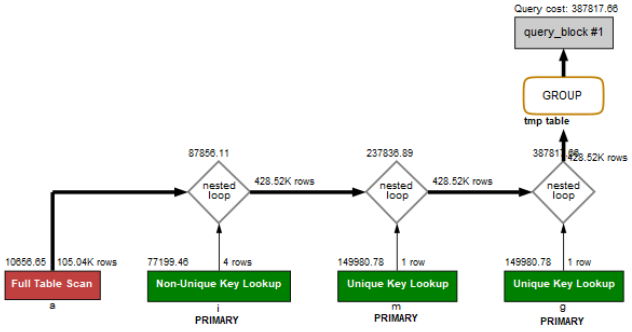

Figure 27. Query Execution Plan of Query 5.3 (Normalized)

A key difference between this execution plan and the previous ones is that the resulting table is not anymore buffered, as there is no need to perform aggregate subtotals. Instead, a temporary table is created for the purpose of facilitating the grouping of the rows of the joined table; this can be reused by the optimizer to speed up query processing (at the cost of added space complexity), but it is automatically dropped once the session ends [31].

Generating the OLAP report using the denormalized data warehouse (Listing 25) without indexes brought down the execution time to 11.860 seconds. Note that the index on `genres` cannot be removed since MySQL mandates indexes for auto-incrementing surrogate keys.

Listing 25. SQL Statement for Query 5.3 (Denormalized)

```
SELECT name, Num_roles
FROM (
   SELECT CONCAT(a.last_name, ", ",
      a.first_name) AS name,  COUNT(m.movie_id)
      AS Num_roles,
```

```
        RANK() OVER
            (PARTITION BY g.genre
             ORDER BY COUNT(m.movie_id) DESC) AS
             Num_roles_rank
    FROM fact_table f
    JOIN genres g ON f.genre_id = g.genre_id
    JOIN actors a ON f.actor_id = a.actor_id
    JOIN movies m ON f.movie_id = m.movie_id
    WHERE g.genre = "Action"
    GROUP BY g.genre, a.last_name, a.first_name
) AS Role_rank
WHERE Num_roles_rank <= 15;
```

Similar to the query execution plan in Section 5.2.2, this presents another case of employing different join algorithms in a single query. Full table scans were performed on the fact table, `movies`, and `actors`; on the other hand, unique key lookup via the auto-incrementing surrogate key was used on `genres`. Since lookups reduce the cardinalities of the intermediate result set, the space-saving nested-loop join algorithm was applied when joining with `genres`. The usual hash join algorithm was used to join all the other tables.

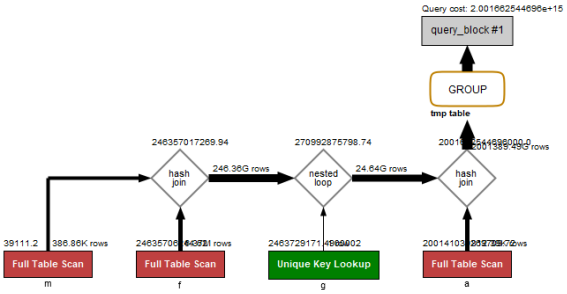

Figure 28. Query Execution Plan of Query 5.3 (Denormalized, Without Indexes Except for Surrogate Key)

It may also be interesting to note that, unlike the roll-up and drill-down queries, `filesort` was not used in grouping the rows. This indicates that the surrogate key index on genres can be used to optimize the `GROUP BY`; this also provides empirical support for the benefits of the design decision to introduce surrogate keys during the dimension modeling phase of the project.

Contrary to the previous OLAP queries, using indexes significantly improved the performance, lowering the run time down to 2.547 seconds. As seen in Figure 29, introducing indexes reduced the full table scan to only one, i.e., on the `genres` table. Since it is also the smallest table in the group (containing only 21 rows versus the hundreds of thousands of rows of the other tables), the scan is inexpensive and does not impact performance.

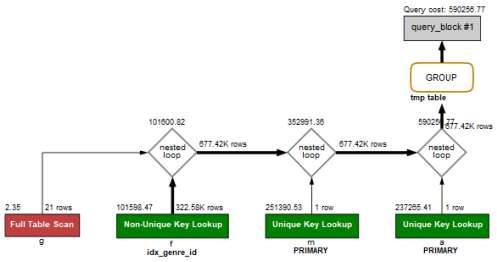

Figure 29. Query Execution Plan of Query 5.3 (Denormalized, With Indexes)

The creation of a composite index for the attributes included in the join and group operations also tapered down the time, clocking 1.391 seconds. Most notably, as seen in Figure 30, it has fully eliminated the need for any full table scan by employing non-unique key lookup on the `genres` table as well.

Therefore, the inner subquery is completed using only seeks, specifically two non-unique key lookups (for `genres` and the fact table) and two unique key lookups (for `movies` and `actors`).

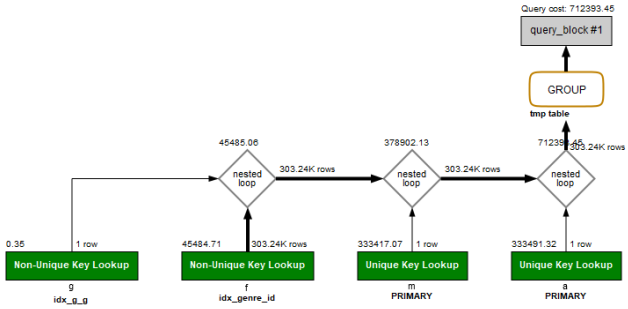

Figure 30. Query Execution Plan of Query 5.3 (Denormalized, With Composite Indexes)

However, the best performance was achieved by revising the query itself, as shown in Listing 26; this reduced the execution time to under a second: 0.728 seconds.

Listing 26. SQL Statement for Query 5.3 (Optimized)

```
SELECT CONCAT(a.last_name, ", ", a.first_name)
   AS name, COUNT(m.movie_id) AS Num_roles
FROM
    (SELECT *
     FROM fact_table
     WHERE genre_id = (
        SELECT genre_id
        FROM genres
        WHERE genre = "Action"))
     AS f
JOIN actors a ON f.actor_id = a.actor_id
JOIN movies m ON f.movie_id = m.movie_id
GROUP BY a.last_name, a.first_name
ORDER BY Num_roles DESC
LIMIT 15;
```

Since the query of interest is only a slice (in the context of OLAP operations), the revised query removes the unnecessary `RANK` and `PARTITION` operations and discards the unneeded grouping based on genre (as there is only a single genre considered in the query). It replaces the former with a combination of `ORDER BY` and `LIMIT`.

Moreover, it also uses a subquery so that only the rows in the fact table with `genre_id` equal to the `genre_id` of `Action` are considered in the joining — whereas the previous approaches applied the filter only after all the tables have been joined.

As seen in Figure 31 (which covers the entire query), the net effect is a decrease in the number of join operations, i.e., from three nested joins to only two. Consequently, it is an entire magnitude lower compared to the previous queries in terms of time complexity.

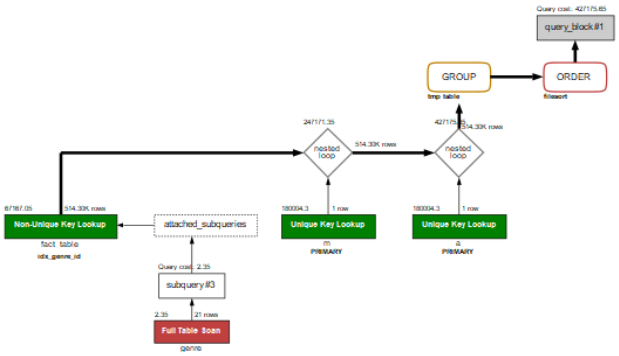

Figure 31. Query Execution Plan of Query 5.3 (Revised, Optimized Query)

5.4 For each country, what are the top 10 movie genres?

Similar to the significance of Query 5.1, determining the top 10 movie genres in a country could produce information on the effect of a country's famous genre of films on their culture. Furthermore, this can also serve as a basis for filmmakers with a target demographic of audiences from a certain country to have a data-driven understanding of the genre of films to which a country is more accustomed and properly decide whether the national demographic would be open to the creation of an innovative or explorative film or would prefer a more customary movie that comfortably appeals to the general public. Furthermore, from a more sociological point of view, Redfern [n.d.] also posits that, aside from box-office sales, there exists a cultural interplay between the national cinema and the dominant genre of a country.

5.4.1 Output

The resulting output has 1057 rows and 4 columns. An excerpt is given in Figure 32.

country	genre	Num_movies	Num_movies_rank
Afghanistan	Drama	14	1
Albania	Drama	39	1
Albania	Romance	31	2
Albania	Sci-Fi	31	2
Albania	Comedy	13	4
Albania	Family	6	5
Albania	Action	3	6
Albania	Documentary	1	7
Albania	Short	1	7
Algeria	Drama	187	1
Algeria	War	66	2
Algeria	Comedy	35	3

Figure 32. Output of Query 5.4

5.4.2 Optimization and Statistics

Generating the OLAP report using a normalized database (Listing 27) took 25.682 seconds.

The query execution plan for the inner subquery (excluding the `RANK` operation, which causes the Visual Explain Tool of MySQL Workbench to crash) is provided in Figure 33. It consists of one full index scan on `movies` followed by non-unique key lookups on the `countries` and `genres` tables. The presence of indexes in the normalized database allows for the usage of the nested-loop join algorithm. Similar to the execution plans in Section 5.3.2, the resulting table is not anymore buffered; instead, it proceeds to the `GROUP BY` operation and maintains a temporary table to improve query processing.

Listing 27. SQL Statement for Query 5.4 (Normalized)

```
SELECT *
FROM (
   SELECT c.country, g.genre, COUNT(m.movie_id)
      AS Num_movies,
      RANK() OVER
         (PARTITION BY c.country
          ORDER BY COUNT(m.movie_id) DESC) AS
            Num_movies_rank
   FROM movies m
   JOIN countries c ON c.movie_id = m.movie_id
   JOIN genres g ON g.movie_id = m.movie_id
   WHERE c.country  != ""
   GROUP BY c.country, g.genre
) AS Movies_rank
WHERE Num_movies_rank <= 10;
```

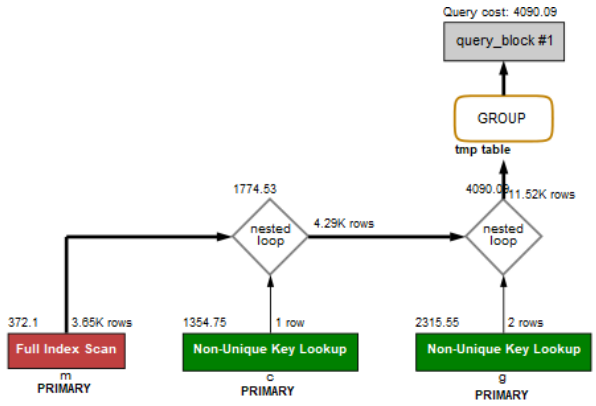

Figure 33. Query Execution Plan of Query 5.4 (Normalized)

Generating the OLAP report using the denormalized data warehouse (Listing 28) without indexes resulted in a decrease in the execution time, clocking 21.129 seconds. Note that the indexes on `countries` and `genres` cannot be removed since MySQL mandates indexes for auto-incrementing surrogate keys.

Listing 28. SQL Statement for Query 5.4 (Denormalized)

```
SELECT *
FROM (
   SELECT c.country, g.genre, COUNT(m.movie_id)
      AS Num_movies,
   RANK() OVER
      (PARTITION BY c.country
       ORDER BY COUNT(m.movie_id) DESC) AS
       Num_movies_rank
   FROM fact_table f
   JOIN countries c ON c.country_id =
      f.country_id
   JOIN genres g ON g.genre_id = f.genre_id
   JOIN movies m ON m.movie_id = f.movie_id
   WHERE c.country  != ""
   GROUP BY c.country, g.genre
) AS Movies_rank
WHERE Num_movies_rank <= 10;
```

Since this optimizer used the two auto-incrementing surrogate key indexes, the execution plan is unique compared to those of the preceding queries by starting and ending with full table scans on the fact table and `movies`, respectively (Figure 34). Between these are unique key lookups (via their surrogate primary key indexes) on `genres` and `countries`. Similar to the trend observed in the previous queries, the optimizer of MySQL uses the nested-loop join algorithm for tables with no indexes and hash joins for those that have indexes.

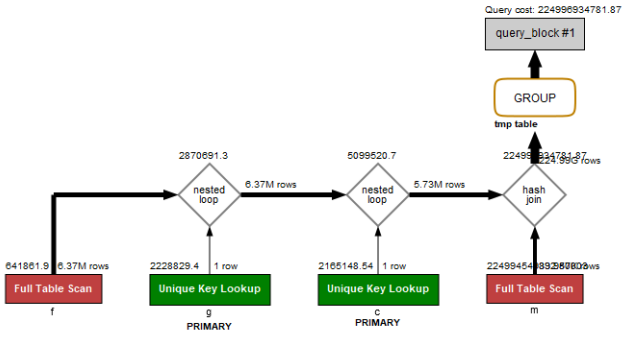

Figure 34. Query Execution Plan of Query 5.4 (Denormalized, Without Indexes Except for Surrogate Key)

Moreover, `filesort` was not used to group the rows, evincing that the surrogate key indexes are suited for the optimization of the `GROUP BY` clause. However, the outer subquery (i.e., the subquery that controls the number of top entries to be returned) still requires a full table scan.

Taking advantage of indexing had a positive effect on the performance of the query, which registered an execution time of 14.884 seconds. Figure 35 shows how it reduces the number of full table scans to the requisite minimum. Since the `genres` table only has 21 rows, the impact of going through each tuple during nested-loop join is insignificant. This scan is followed by three seeks: a non-unique key lookup on the fact table and unique key lookups (via primary key indexes) on `countries` and `movies`.

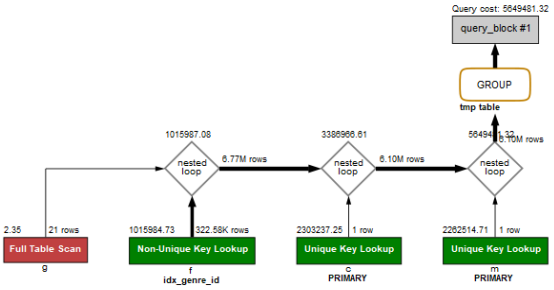

Figure 35. Query Execution Plan of Query 5.4 (Normalized)

Adding composite keys further decreased the execution time to 9.468 seconds. The query execution plan is similar to that of the previous experiment; however, instead of performing a full table scan on genres, it starts with a full index scan on movies, then proceeds to a non-unique key lookup on the fact table and unique key lookups on genres and `countries`. This change in the accessing mode and the order of joining the tables as a result of the introducing composite keys was contributory to the more optimal evaluation of the MySQL optimizer.

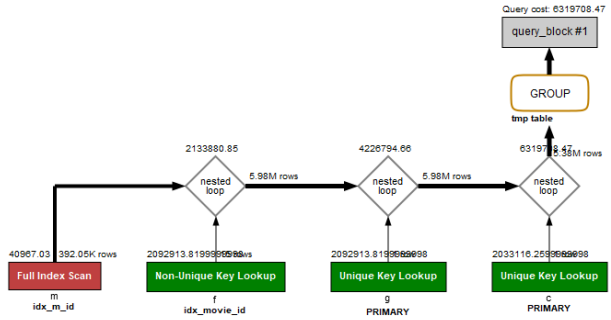

Figure 36. Query Execution Plan of Query 5.4 (Denormalized, With Composite Indexes)

5.5 How many horror movies are released in the USA for each year and decade?

According to BoxOfficeMojo datasets, box office revenue from movies in the horror genre has increased significantly in the late 2010s [11]. This can be attributed to the rise of true crime documentaries, which have fueled the market once more for horror and thriller films. By using queries to determine how many horror movies are released in the USA, it can be seen whether this increase in box office revenue for horror movies has been the result of a culture shift, a boost in the quality of horror movies, or simply just an upsurge in the number of produced horror films in recent years.

5.5.1 Output

The resulting output has 99 rows and 3 columns. An excerpt is given in Figure 37.

decade_released	year	num_movies
1990s	1998	1383
1990s	1999	1396
1990s	NULL	12076
2000s	2000	1604
2000s	2001	1300
2000s	2002	1929
2000s	2003	1762
2000s	2004	1787
2000s	2005	434
2000s	2006	5
2000s	NULL	8821
NULL	NULL	50221

Figure 37. Output of Query 5.5

5.5.2 Optimization and Statistics

Generating the OLAP report using a normalized database (Listing 29) took 31.154 seconds.

Listing 29. SQL Statement for Query 5.5 (Normalized)

```
SELECT CONCAT(FLOOR(year/10) * 10, 's') AS
   decade_released, m.year, COUNT(m.movie_id) AS
   num_movies
FROM fact_table f
JOIN genres g ON g.genre_id = f.genre_id
JOIN movies m ON m.movie_id = f.movie_id
JOIN countries c ON c.country_id = f.country_id
WHERE c.country = "USA"
   AND g.genre = "Horror"
GROUP BY decade_released, m.year WITH ROLLUP;
```

In Figure 38, it can be seen that the query execution plan involves a full index scan on the `countries` table, followed by unique key lookups on `movies` and `genres`. The said tables were joined using the nested-loop join algorithm. Since the query involves a roll-up, the resultant joined table is first buffered into a temporary table in order for the subtotals to be calculated. Subsequently, `filesort` was used to resolve the `GROUP BY` clause.

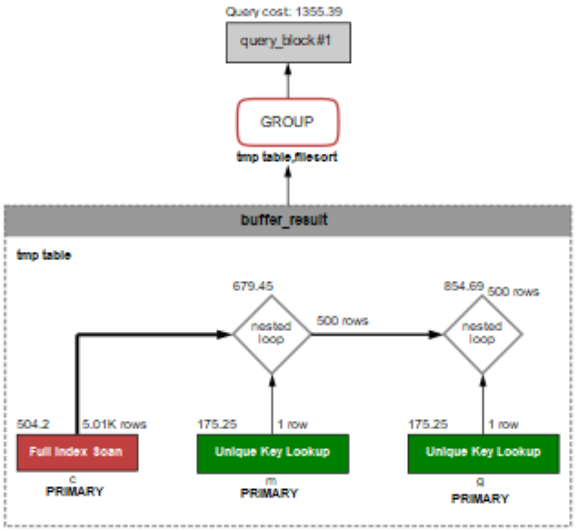

Figure 38. Query Execution Plan of Query 5.5 (Normalized)

Generating the same report using the denormalized data warehouse (Listing 30) registered a significantly shorter execution time of 2.860 seconds. Note that the index on `genres` cannot be removed since MySQL mandates indexes for auto-incrementing surrogate keys.

Listing 30. SQL Statement for Query 5.5 (Denormalized)

```
SELECT CONCAT(FLOOR(year/10) * 10, 's') AS
   decade_released, m.year, COUNT(m.movie_id) AS
   num_movies
FROM fact_table f
JOIN genres g ON g.genre_id = f.genre_id
JOIN movies m ON m.movie_id = f.movie_id
JOIN countries c ON c.country_id = f.country_id
WHERE c.country = "USA"
   AND g.genre = "Horror"
GROUP BY decade_released, m.year WITH ROLLUP;
```

With the absence of indexes aside from that for the surrogate key of `countries`, the optimizer resorted to the use of full table scans on `genres`, the fact table, and `movies`, and unique key lookup on `countries`. It also employed the hash-join algorithm for joining `genres` and the fact table, nested-loop join for joining with `genres`, and hash join anew for joining with `movies` (Figure 39). This choice of algorithm is heavily dependent on the presence of indexes. Similar to the previous experiment, the resulting table was first buffered before performing `filesort` for the grouping.

The addition of indexes brought the execution down to over half a second, i.e., 0.562 seconds. The order of the join statements in the query execution plan shown in Figure 40 is the same as that in the previous experiment. However, the full table scans (with the exception of the initial table scan on `genres`) were replaced with a non-unique key lookup on the fact table and unique key lookups on `countries` and `movies`. The nested-loop join algorithm was also employed throughout the entire query execution.

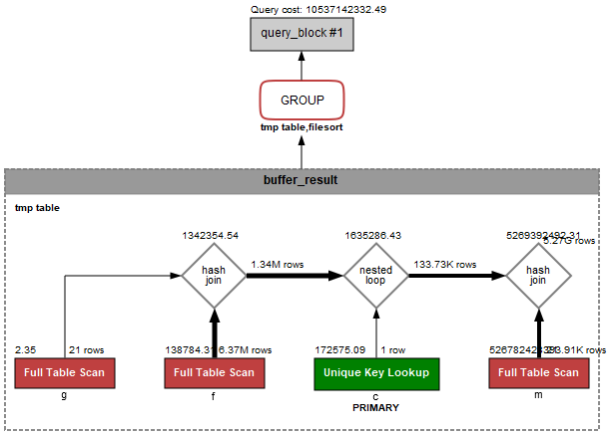

Figure 39. Query Execution Plan of Query 5.5 (Denormalized, Without Indexes Except for Surrogate Key)

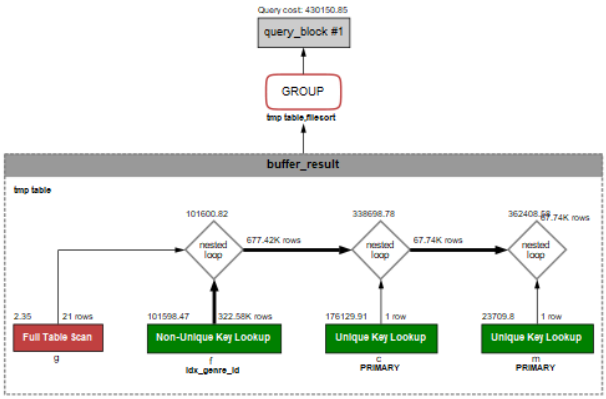

Figure 40. Query Execution Plan of Query 5.5 (Denormalized, With Indexes)

Interestingly, the creation of composite indexes for the attributes involved in the join and group operations yielded the same execution plan as in Figure 40. **However, there is a slight edge in its execution time, clocking 0.328 seconds,** although the significance of this improvement cannot be ascertained since the same query execution plan is followed. It is also possible that the difference was caused by under-the-hood optimization techniques not captured in the execution plan and by extraneous factors, such as memory contention of processes running in the system other than the MySQL service (especially since the author's machines are not dedicated servers).

Finally, the fastest execution time was registered after revising the query itself, which was recorded at 0.152 seconds. This optimized SQL statement is presented in Listing 31.

Listing 31. SQL Statement for Query 5.5 (Optimized)

```
SELECT CONCAT(FLOOR(year/10) * 10, 's') AS
  decade_released, m.year, COUNT(m.movie_id) AS
  num_movies
FROM (
   SELECT *
   FROM fact_table
```

```
    WHERE country_id = (
        SELECT country_id
        FROM countries
        WHERE country = "USA"
    )
    AND genre_id = (
              SELECT genre_id
        FROM genres
        WHERE genre = "Horror"
    )) AS f
 JOIN movies m ON m.movie_id = f.movie_id
 GROUP BY decade_released, m.year WITH ROLLUP;
```

Since the query fundamentally employs dicing operations, the rationale for the revision is to reduce the number of rows involved in the join operations by filtering the `country_id` to only that of `USA` and the `genre_id` to only that of `Horror`, even prior to performing the join; this was done using subqueries. This differs from the previous queries, which applied the filtering only after the tables have been joined.

Moreover, the main performance gain from this revised query is a result of the reduction in the number of joined tables, i.e., from the original three join operations to only a single one — this alone already presents a significant decrease in the time complexity by two orders of magnitude.

The presence of indexes suited to this join operation also made it possible for the optimizer to use the index merge method. MySQL employs this to implement an optimized procedure referred to as the *index merge intersection access algorithm*. This allows conditions joined by the logical operator `AND` (i.e., an intersection in set theory) to benefit from simultaneous scans on all the pertinent indexes [32]; the results of these scans are combined to yield the result set.

The visual diagram of this optimization can be gleaned from the query execution plan in Figure 41.

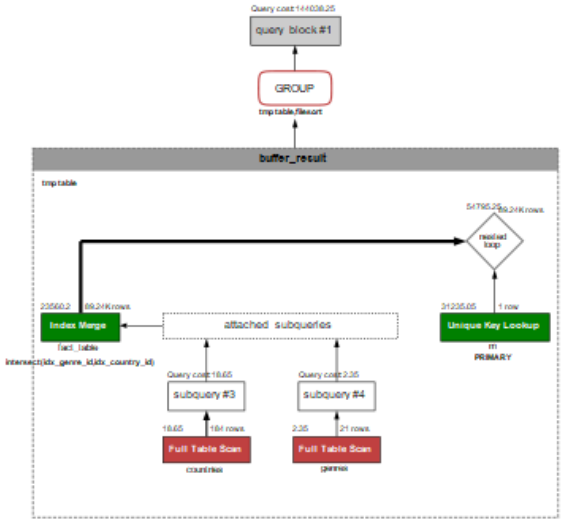

Figure 41. Query Execution Plan of Query 5.5 (Revised, Optimized Query)

5.6 What are the production companies that have released over 10 movies with a rank above the average rank and a number of votes above the average total number of votes?

Good writing and captivating stories are not all that make a movie successful. Many film writers can create stories that are not backed by the right production companies, resulting in failed film releases due to a lack of development planning, budget allocation, and distribution strategies [8]. By seeing which production companies have been successful in releasing high-quality movies that are above the average ratings for all movies, film writers can see the companies that have been able to propel stories from text on paper — to the big cinema screen.

5.6.1 Output

The resulting output has 10 rows and 2 columns. An excerpt is given in Figure 42.

production_company	num_movies
Warner Bros.	47
Universal Pictures	37
Twentieth Century Fox	31
Touchstone Pictures	18
Paramount Pictures	34
New Line Cinema	17
Metro-Goldwyn-Mayer (MGM)	13
Eon Productions	11
DreamWorks	11
Columbia Pictures	24

Figure 42. Output of Query 5.6

5.6.2 Optimization and Statistics

Generating the OLAP report using a normalized database (Listing 32) took 10.311 seconds.

Listing 32. SQL Statement for Query 5.6 (Normalized)

```
SELECT production_company, COUNT(imdb_title_id)
   AS num_movies
FROM (
   SELECT i.production_company, i.imdb_title_id
   FROM imdb_movies i
   WHERE i.imdb_title_id IN (
      SELECT r.imdb_title_id
      FROM imdb_ratings r
      WHERE r.mean_vote > (
         SELECT AVG(r.mean_vote)
         FROM imdb_ratings r
      )
   )
   AND imdb_title_id IN (
      SELECT i.imdb_title_id
      FROM imdb_movies i
      WHERE i.votes > (
          SELECT AVG(i.votes)
          FROM imdb_movies i
      )
   )
) AS selected_movies
GROUP BY production_company
HAVING num_movies > 10
ORDER BY num_movies DESC;
```

The execution plan in Figure 43 shows that, after the resolution of the subqueries, a full table scan is done on the table containing the movies, which is followed by unique key lookups (via primary key indexes) on the table containing the ratings and the movies anew. Afterwards, a temporary table is created for grouping, and, with the absence of suitable indexes for sorting, the `filesort` algorithm is employed for ordering.

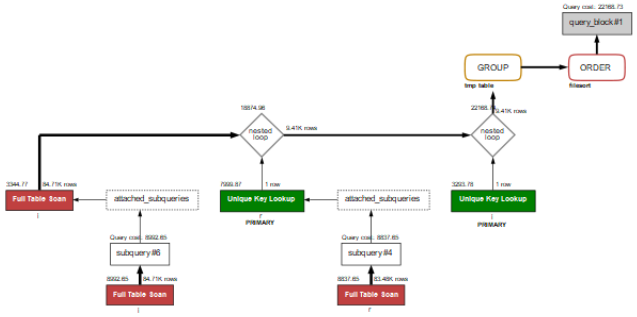

Figure 43. Query Execution Plan of Query 5.6 (Normalized)

On the other hand, using the denormalized data warehouse to produce the same report actually caused the execution time to increase to 12.468 seconds. The relevant SQL command for this is given in Listing 33.

Listing 33. SQL Statement for Query 5.6 (Denormalized)

```
SELECT production_company, COUNT(movie_id) AS
   num_movies
FROM (
   SELECT m.movie_id, m.production_company,
       total_num_votes AS total
   FROM movies m
   JOIN fact_table ft
      ON ft.movie_id = m.movie_id
   JOIN ratings ra
      ON ra.rating_id = ft.rating_id
   WHERE m.production_company IS NOT NULL
   AND m.rank > (
      SELECT AVG(m.rank)
      FROM movies m
   )
   GROUP BY m.movie_id
   HAVING total > (
      SELECT AVG(total_num_votes)
      FROM (
         SELECT  name, total_num_votes
         FROM fact_table f
         JOIN movies m
            ON m.movie_id = f.movie_id
         JOIN ratings r
            ON r.rating_id = f.rating_id
         JOIN genres g
            ON g.genre_id = f.genre_id
       ) as total_votes)
   ) AS selected_movies
GROUP BY production_company
HAVING num_movies > 10
ORDER BY num_movies DESC;
```

The query execution plan (Figure 44) provides some insights into the reason for the increase in the execution time. Aside from the necessary full table scan to resolve one of the subqueries, all tables involved in the SQL statement, namely `ratings`, `movies`, and the fact table, were subjected to full table scans and were subsequently joined via the hash join algorithm.

To execute the outermost query, the results of this join table was placed inside a materialized table; as a temporary table stored in the memory (ensuring fast access), it provides a performance gain when using subqueries since the materialized table can be referred to repeatedly every time it is needed during the query execution.

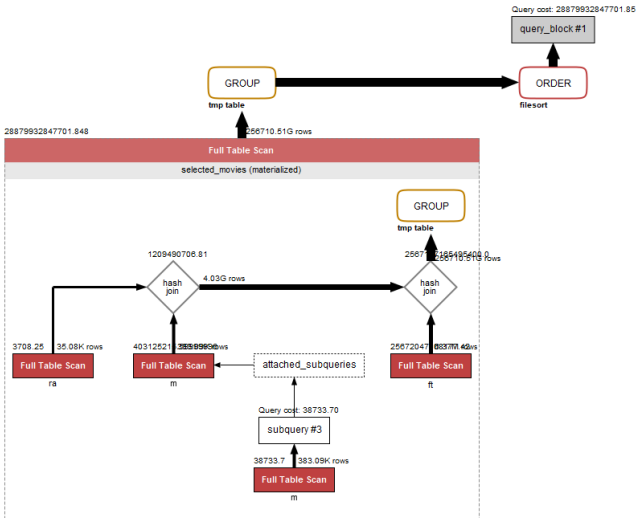

Figure 44. Query Execution Plan of Query 5.6 (Denormalized)

The addition of indexes on the attributes included in the join operation trimmed the execution time to 10.531 seconds. As seen in Figure 45, the optimizer performed a single full table scan only to resolve one of the inner subqueries. However, it opted to take advantage of the indexes and proceed with a full index scan on movies, non-unique key lookup on the fact table, and unique key lookup on ratings. The result of joining these tables (via nested-loop join) and grouping its rows was stored in a temporary materialized table. Although the created indexes were applicable for the grouping operation, it was not well suited for sorting; thus, `filesort` was used for the latter.

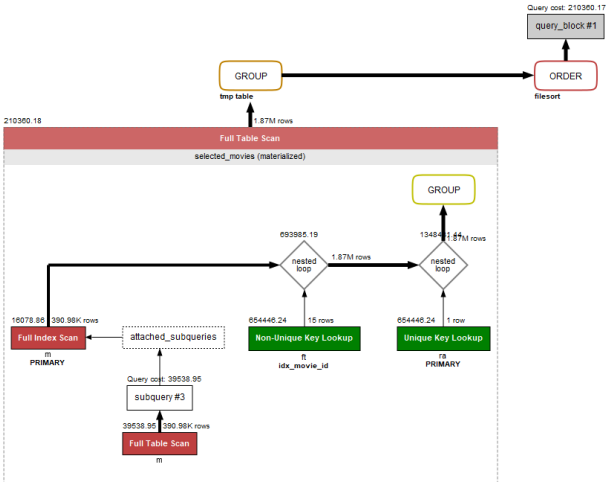

Figure 45. Query Execution Plan of Query 5.5 (Denormalized, With Indexes)

Finally, similar to the case seen in Section 5.5.2, the query execution plan after the creation of composite indexes for the attributes involved in joining, grouping, and ordering is identical to the execution plan presented in Figure 45. **However, using these composite indexes registered the fastest execution time, i.e., 8.762 seconds.** Directly attributing this performance gain to a particular evaluation of the optimizer is difficult since it shares the same execution plan as the previous experiment. Nevertheless, it is possible for this improvement to be the result of under-the-hood optimization techniques that are not specified in the execution plan, as well as a reduction in memory contention in the server or in the system itself.

5.7 For each of the top 6 directors with the highest gross earnings, what are their 15 highest-rated movies?

Knowing the directors with the highest gross earnings gives one an idea of which directors have produced high-quality movies that may be worth watching. Filtering their highest towards their 15 highest-rated movies may give users a more informed basis for movie recommendations. At the same time, it may also be relevant for those who are in the field of film studies who would like to delve into the directorial or cinematographic styles of directors and gain a deeper understanding as to how these director's distinct trademarks are imprinted or evident in their top-rated movies [12].

5.7.1 Output

The resulting output has 76 rows and 4 columns. An excerpt is given in Figure 46.

full_name	gross	name	rank
Spielberg, Steven	5520276631	Escape to Nowhere	9.1
Spielberg, Steven	5520276631	Firelight	9
Spielberg, Steven	5520276631	Schindler's List	8.8
Spielberg, Steven	5520276631	Raiders of the Lost Ark	8.7
Spielberg, Steven	5520276631	Slipstream	8.6
Spielberg, Steven	5520276631	The Last Gun	8.4
Spielberg, Steven	5520276631	Amblin'	8.4
Spielberg, Steven	5520276631	Saving Private Ryan	8.3
Spielberg, Steven	5520276631	Jaws	8.2
Spielberg, Steven	5520276631	Indiana Jones and the Last Crusade	8
Spielberg, Steven	5520276631	Close Encounters of the Third Kind	7.8
Spielberg, Steven	5520276631	E.T. the Extra-Terrestrial	7.8
Spielberg, Steven	5520276631	Minority Report	7.8
Spielberg, Steven	5520276631	Catch Me If You Can	7.7
Spielberg, Steven	5520276631	The Color Purple	7.6
Cameron, James...	3295280440	Aliens	8.2
Cameron, James...	3295280440	Terminator 2: Judgment Day	8.1

Figure 46. Output of Query 5.7

5.7.2 Optimization and Statistics

MySQL does not have a built-in pivot operation. However, this OLAP operation can be simulated with the use of subqueries, as demonstrated in the queries listed in this section.

Generating the OLAP report using a normalized database (Listing 34) took 2.906 seconds.

Listing 34. SQL Statement for Query 5.7 (Normalized)

```
SELECT name, title, `rank`
FROM (
   SELECT m.title, topdir.name, topdir.gross,
      r.rank, ROW_NUMBER() OVER(PARTITION BY
      topdir.name ORDER BY `rank` DESC) AS
      director_movie_rank
   FROM (
      SELECT d.directorid, d.name, d.gross
      FROM directors d
      ORDER BY d.gross DESC
      LIMIT 6
   ) as topdir
   JOIN movies m
   JOIN ratings r ON r.movieid = m.movieid
   JOIN movies2directors md ON md.movieid =
      m.movieid AND md.directorid =
      topdir.directorid
   ORDER BY topdir.gross DESC, r.rank DESC
) AS topdirmovies
WHERE director_movie_rank <= 15;
```

The presence of a materialized table confirms the nature of the query as a pivot operation. After performing a full table scan on the materialized table and a full index scan on the mapping table for the movies and directors, unique key lookups (via primary key indexes) are employed on both `movies` and `ratings`. The first two tables are joined via the hash join algorithm, whereas the remaining ones are joined via nested-loop join. Finally, the rows of the joined table are transferred to a temporary table for ordering via the `filesort` algorithm.

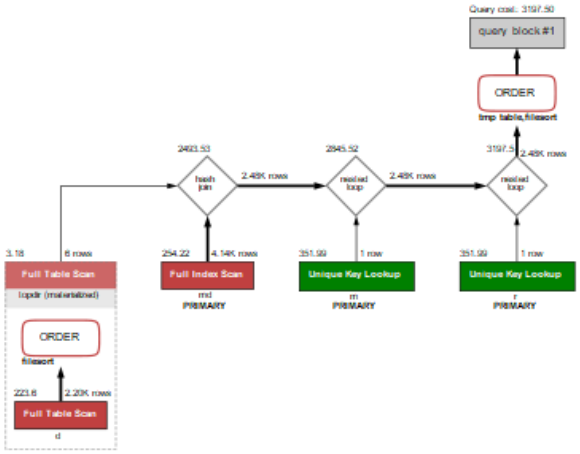

Figure 47. Query Execution Plan of Query 5.7 (Normalized)

Creating this report using the denormalized data warehouse (Listing 35), albeit without indexes, decreased the execution time to under 1 second, i.e., 0.157 seconds.

Listing 35. SQL Statement for Query 5.7 (Denormalized)

```
SELECT full_name, gross, name, `rank`
FROM (
   SELECT full_name, m.name, gross, m.rank,
      ROW_NUMBER() OVER
      (PARTITION BY td.director_id ORDER BY
          `rank` DESC) AS Director_movie_rank
   FROM (
      SELECT d.director_id,
         CONCAT(CONCAT(d.last_name, ", "),
         d.first_name) AS full_name, d.gross
      FROM directors d
      ORDER BY d.gross DESC
      LIMIT 6
```

```
    ) as td
    JOIN fact_table f ON td.director_id =
       f.director_id
    JOIN movies m ON m.movie_id = f.movie_id
    WHERE `rank` IS NOT NULL
    GROUP BY m.movie_id
    ORDER BY gross DESC, m.rank DESC
) AS top_director_movies
WHERE Director_movie_rank <= 15;
```

The absence of any indexes triggered the execution to consist solely of full table scans and hash joins, as seen in Figure 48. Interestingly, the execution plan also showed the creation of a nested materialized table to speed up the execution of the query. Materialized tables serve as temporary tables to which the results of a subquery can be stored for reuse by outer queries. They are also stored in the memory for faster retrieval.

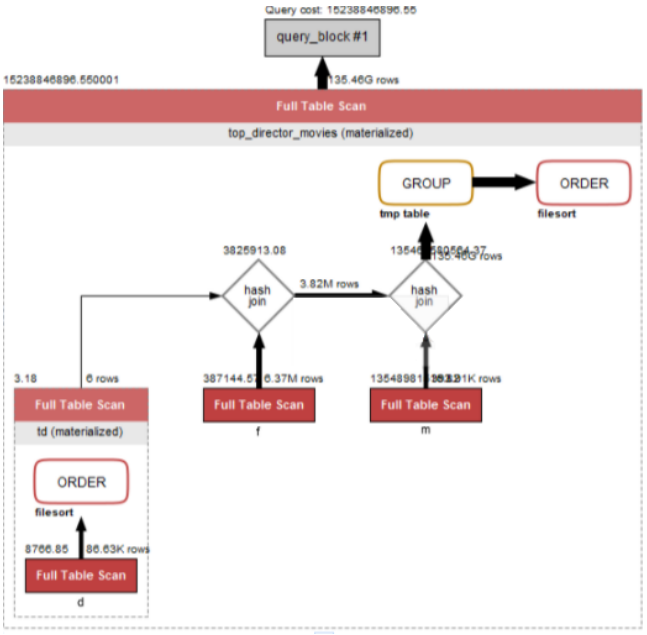

Figure 48. Query Execution Plan of Query 5.7 (Denormalized, Without Indexes)

Using indexes further brought down the execution time to less than a tenth of a second, i.e. 0.062 seconds. These indexes were set up on the columns used in joining the tables. Most notably, the index on the ID of the director transforms the full table scan on the fact table into a non-unique key lookup. The primary key index is then employed to perform a unique key lookup on `movies`. Similar to the previous experiment, a nested materialized table was also constructed to optimize subquery execution.

Finally, using composite indexes on the columns involved in the joining, grouping, and ordering registered the highest performance gain, clocking 0.032 seconds. The query execution plan is fundamentally similar to that of the previous indexed version, with the exception of using a full index scan (rather than a full table scan) on the innermost subquery. Figure 50 also shows that the index used in this full index scan is the created composite index. Although the performance of hard-disk drives decreases with random accesses, this particular query benefited from the faster retrieval times of using a B^+-tree versus a sequential read.

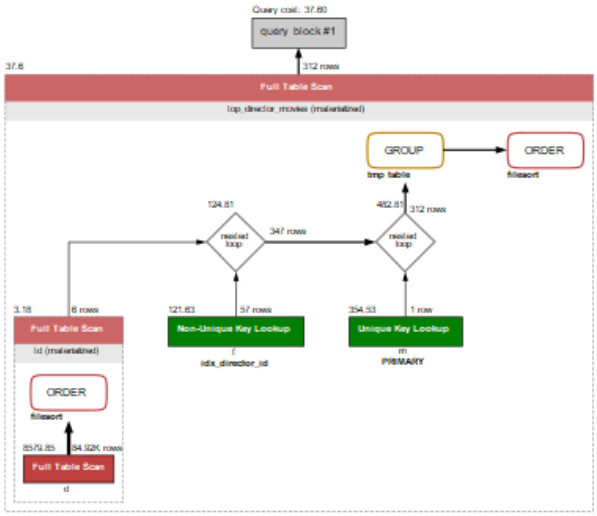

Figure 49. Query Execution Plan of Query 5.7 (Denormalized, Without Indexes)

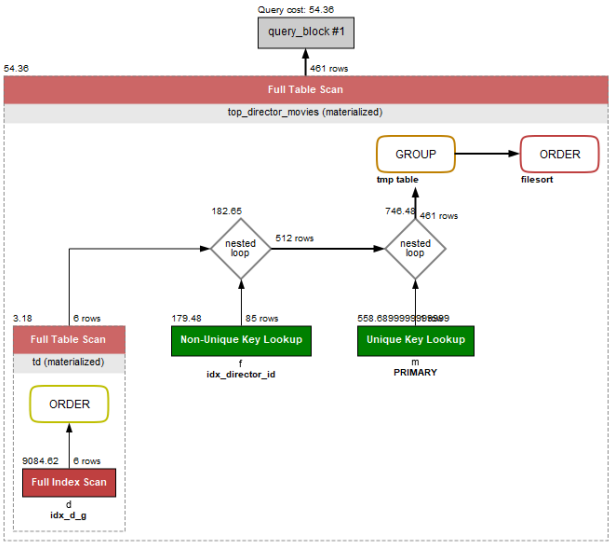

Figure 50. Query Execution Plan of Query 5.7 (Denormalized, Without Composite Indexes)

6. Results and Analysis

This section presents the results and analysis related to the various functional and performance tests conducted in order to verify the correctness of the ETL script and OLAP queries, alongside their performance in terms of their execution time.

6.1 Functional Testing

This section discusses the functional testing of the ETL script and the OLAP. In particular, functional testing ensures that the data to be used in all MySQL queries are complete, and will not cause any possible ambiguities with the analysis of the returned values.

6.1.1 ETL Script

To ensure the data quality and the correctness of the ETL script, both manual and automated tests were set in place. These were based on the suite of tests proposed by Yaddow [2019].

In checking the local copy of the IMDb IJS, its metadata was first matched against the metadata of the remote copy using the following script. In particular, the table names, column names, ordinal positions, nullity constraints, column types (data types and maximum character lengths for strings/character data types or numeric precision for integer/floating-point data types), and collation names (which also include the character set names) were compared using the script in Listing 36. The metadata validation was considered passed if all these pieces of information matched.

Listing 36. Metadata Validation

```
SELECT TABLE_NAME, COLUMN_NAME,
   ORDINAL_POSITION, IS_NULLABLE, COLUMN_TYPE,
   COLLATION_NAME
FROM INFORMATION_SCHEMA.COLUMNS
WHERE table_schema = 'imdb_ijs';
```

Then, the contents of the local copy were validated against the remote copy. To prevent expensive row-by-row comparison, the `CHECKSUM` function of MySQL was used, which computes the checksum for the contents of a table (an example for the `actors` table is given in Listing 37). Although it is not guaranteed that the hashing function is collision-free [30], it is already adequate for the purposes of this automated test. The content validation was considered passed if the checksums matched.

Listing 37. Content Validation

```
CHECKSUM TABLE imdb_ijs.actors;
```

However, this checksum-based approach does not apply to the datasets provided as CSV files. Instead, boundary value analysis was performed by comparing the maximum and minimum values of each column. The table cardinalities were compared as well. In this regard, the content validation for the CSV data source was deemed passed if their boundary values and cardinalities matched.

6.1.2 OLAP Queries

As evidenced in Section 5, all the OLAP queries have at least two versions: a version for the normalized dataset and another for the denormalized data warehouse. The formulations of these queries are fundamentally different due to the varying schema design, but their results should be the same. Therefore, the correctness of the OLAP queries in this project was verified using the `CHECKSUM` function discussed in the previous section. The content validation was considered passed if the checksums matched, i.e., the result sets from both queries were identical.

Aside from this automated comparison of checksums, the authors also had a form of sanity check by comparing the boundary values and cardinalities against those that were obtained via Tableau's graphical user interface (GUI) for conducting OLAP. This adds another layer of testing the logic of the queries since performing OLAP in MySQL is text-based coding whereas performing OLAP in Tableau is predominantly drag-and-drop.

Additionally, complementing the content validation, a battery of scripts was also run to ensure that the data wrangling steps were successfully and correctly reflected in the contents of the data warehouse. These scripts cover the following cases:

- There should be no movie title that ends with a comma (Listing 38).
- There should be no movie title that ends with a grammatical article (Listing 39). Aside from *a, an,* and *the*, the foreign grammatical articles are listed in Table 6.
- There should be no genre that contains a comma since the genres have been exploded (Listing 40).
- There should be no production company where its last word's first and last characters are brackets (Listing 41).
- There should be no production company where its last word's first and last characters are parentheses, and its penultimate word is terminated with a hyphen (Listing 42).

Listing 38. Movie Title Wrangling Validation (Comma)

```
SELECT COUNT(*) FROM imdb_star.movies
WHERE `name` LIKE "%,";
```

Listing 39. Movie Title Wrangling Validation (Article)

```
SELECT COUNT(*) FROM imdb_star.movies
WHERE `name` LIKE "%, The";
```

Listing 40. Genre Wrangling Validation

```
SELECT COUNT(*) FROM imdb_star.genres
WHERE genre LIKE "%,%";
```

Listing 41. Production Company Wrangling Validation (Brackets)

```
SELECT COUNT(*) FROM imdb_star.genres
WHERE genre LIKE "%[%]";
```

Listing 42. Production Company Wrangling Validation (Parentheses)

```
SELECT COUNT(*) FROM imdb_star.movies
WHERE production_company LIKE "%- (%)";
```

6.2 Performance Testing

In performing all the MySQL queries, a desktop computer with the following machine specifications was used.

- **Processor:** AMD Ryzen 5 5600x 6-Core Processor
- **Processor Base Frequency:** 3.70 GHz
- **Memory:** 16GB DDR4 2667MHz
- **Disk:** 1TB Hard Disk Drive

The performance of the queries was characterized by the duration (in seconds) needed to run them. A summary of the results, which were discussed in the preceding section, is presented in the table below, detailing the duration of the queries for the normalized schema, the denormalized schema (using no indexes, indexes on the join attributes, and composite indexes on the join, grouping, and ordering attributes), and revised queries on the denormalized schema, if applicable. The iterations yielding the shortest duration for each query are also underlined and given in boldface.

Several generalizations and query behaviors can be confirmed from the performance results. First, the use of a denormalized database results in faster query durations, as evinced in the performance results of all seven queries. In particular, the use of a snowflake schema, which was the chosen design for the denormalized database, ensured that naïve nested-loop joins for the database queries will be minimized, significantly optimizing a common bottleneck of queries in general.

Table 7. Performance Testing Results (in Seconds)

Query	Normalized	Denormalized			
		No Indexes	With Indexes	With Composite Indexes	Revised Query
1	23.400	**9.344**	15.453	16.516	N/A
2	12.500	**9.359**	12.656	13.375	N/A
3	18.752	11.860	2.547	1.391	**0.728**
4	25.682	21.129	14.884	**9.468**	N/A
5	31.154	2.860	0.562	0.328	**0.152**
6	10.311	12.468	10.531	**8.762**	N/A
7	2.906	0.157	0.062	**0.032**	N/A

Second, the creation of indexes can improve query durations, as highlighted in the performance results of queries 4 (Section 5.4), 6 (Section 5.5), and 7 (Section 5.7). For the aforementioned SQL queries, the iterations that logged the fastest durations were those wherein composite indexes were created. As indexes allow for directed searches on selected columns of the data, making use of indexes in database queries decreases the number of reads the system performs on a table, especially when there are joining or filtering conditions in the query. The use of a composite index, which spans multiple columns, further optimizes the number of reads of the system and results in faster durations.

From a theoretical lens, whereas naïve nested-loop joins result in quadratic time complexity for joining two tables, indexed nested-loop joins achieve this in linearithmic time since a lookup on a B^+-tree is sublinear, i.e., $O(\log n)$. However, this comes with an added space complexity $O(n)$ for maintaining the data structure.

Analyzing the cost of these join algorithms from a low-level perspective also supports the advantages of indexed nested-loop joins. Suppose b_r and b_s pertain to the number of blocks that contain relations r and s, respectively. Silberschatz, Korth, and Sudarshan [42] compute that, assuming a worst-case scenario, the number of block transfers needed for a naïve nested-loop join is $|r| \times b_s + b_r$ and the number of disk seeks needed is $|r| + b_r$. Assuming best-case, this reduces to $b_s + b_r$ transfers and 2 seeks.

On the other hand, employing an indexed nested-loop join costs only $b_r \times (T_{transfer} + T_{seek}) + |r| \times c$, where $T_{transfer}$ and T_{seek} refer to the block transfer time and disk seek time, respectively. Meanwhile, c is the cost of selection. If h is the height of the index with respect to the B^+-tree, then the value of c is equal to $(h + 1)(T_{transfer} + T_{seek})$ if the equality is tested on keys, regardless of whether the indexes are clustered or unclustered.

Third, there exist use cases wherein the creation of indexes lengthens the query duration rather than decreasing it. In particular, this behavior was observed for the first two queries, which involve the roll-up and drill-down operations, respectively. As detailed in the preceding section, this may be a common behavior for queries requiring the majority of the rows of the tables to be processed, such as those involving the aggregate functions used in the first two queries. A possible explanation for this behavior is that the system has optimizations in place for full table scans, such as the capacity to perform multiblock reads, that are not used for seek statements.

Finally, the performances of some queries may also improve drastically through revisions of the queries themselves, as seen in the results of queries 3 and 5, which involve the slice and dice operations, respectively. In contrast to the structures of the other queries, the revised queries similarly take advantage of the slice and dice conditions by filtering the rows of a table before joining it with the other tables. As detailed in the preceding section, this optimization lessens the time complexity of the revised queries by entire orders of magnitude, leading to the drastic decrease in query durations shown in Table 7. While this technique cannot be applied for the other queries, as their filtering operations are based on derived attributes that can only be determined after all necessary tables have been joined, further queries can be optimized when they fit this form, or if they can be restructured in such a way that filtering can be performed before joining.

6.3 Performance of the Dashboard

It is important to emphasize that the performance of the actual OLAP application, i.e., the interactive dashboard containing the data visualizations, is different from that of the queries presented in the previous section (Section 6.2).

Specifically, to meet the requirement in the project rubrics of having the OLAP application load in under five (5) seconds, it was imperative to perform further optimizations in Tableau itself, especially since allowances also have to be made for the rendering of the visualizations and interactive functionalities.

The most significant optimization was to use Tableau's relations instead of performing joins on the source tables. Unlike physical joins in MySQL, Tableau's relations perform the joins only when they are needed, i.e., they are deferred "to the time and context of analysis" [46]. This is analogous to the technique of lazy evaluation used in functional programming languages. Moreover, relationships are advantageous in that they do not require up-front join types; relationships are only defined by the fields that are matched with one another, making their implementations more flexible. The join types are automatically selected by Tableau based on the fields used, and these join types can also be adjusted during the analysis without sacrificing the granularity of the original data.

Moreover, parameters and calculated fields were also used for interactively and programmatically changing the cardinality of the result set to be displayed or switching between granularities of the report (in light of the decade → year dimension hierarchy). For instance, a calculated field was used to obtain the decade from the year, which is markedly faster since it utilizes Tableau's native functionalities and programming language. The code for this is shown in Listing 43.

Listing 43. Calculated Field for Decade → Year

```
CASE [Granularity]
   WHEN "By year" THEN [Year]
   WHEN "By decade" THEN [Year] - [Year] % 10
END
```

Tableau sets were also created out of the attribute data in order to dynamically filter the number of entries for inclusion in the visualization; the term "sets" may be a misnomer since they are technically subsets of the data [45]. Their cardinalities were set to be dependent on the parameter set by the user.

For instance, the user may drag the slider to 10 in order to limit the cardinality to only the top 10 directors with the highest gross earnings. Since Tableau dynamically links these subsets to the data instead of being static extracts [45], using this feature eliminates the need to reprocess the entire query from scratch should the user drag the slider to another value.

Finally, the parameters, which can be configured to be a range of integers, serve the pragmatic purpose of limiting the result size that can be requested by the end user to cardinalities that can be fetched within reasonable time, without compromising the insight that can be derived from the analytics.

The performance of the dashboard was timed by running them on the desktop version of Tableau Public in order to discard the effect of Internet connectivity of bandwidth, with the exception of those that include maps since they have to be fetched by Tableau from the servers of Mapbox and OpenStreetMap [25].

Nevertheless, the run times of all the interactive actions performed on the OLAP dashboard are under 3 seconds, thus meeting the requirements in the project specifications.

7. Conclusion

In all, completing the project required several different tools and applications; apart from using MySQL to store and retrieve data from the data warehouse, the team also made use of online databases, namely IMDb and Kaggle, in order to obtain the raw data, Apache Nifi to automate the ETL processes of the data warehouse, Python scripts to execute supplementary data transformations, and Tableau Public for creating interactive visualizations of the queries.

Through creating the data warehouse and performing the ETL processes, the team was able to obtain a deeper appreciation for the behavior and characteristics of data; though data is often regarded as a monolithic source of information when developing software or applications, this data-centric project evinced the immense variety of forms data can take, as well as the complications that could arise when attempting to synthesize data from different sources or data generated with different objectives.

To resolve these issues, the team similarly had to take a more diversified approach to data processing; rather than relying solely on Apache Nifi, the team searched for compatible tools such as Python scripts that could be integrated into the existing pipeline.

Through the development of the OLAP application, the team was able to further understand the OLAP operations and the magnitude of their applications in generating insights from the raw data. For instance, the team realized that although the roll-up and drill-down operations entailed different interpretations, their results can be generated using the same query details. Moreover, through a series of simple table joins and filters, the rather disjointed data stored in the data warehouse can be used to generate involved information, such as the relative standings of people or groups in the movie industry.

The final segment of the project was query processing, wherein the performance of the group's initial queries were measured and iterated upon to produce faster and more efficient results. While the results of the analyses supported some of the points discussed in the lectures, such as the benefits provided by using denormalized databases and creating indexes, the team was also able to discover practical examples of exceptions to some of these rules of thumb.

For instance, the first two queries performed better without the use of indexes because the roll-up and drill-down operations involved reading through almost all the rows in the table. Especially in hard-disk drives, which are subject to disk rotation latency, this favors multiblock sequential scans instead of random accesses resulting from repeated B^+-tree traversals.

Ultimately, the team realized that data is best managed on a case-to-case basis; although there are many tools and techniques available for optimizing query performance, a good database designer must be able to discern when to introduce these tools and when they would serve as bottlenecks rather than benefits depending on the structure and predicted use cases of the data.

Finally, apart from technological experience and realizations, the team was also able to use the data to generate several profound insights regarding the film industry, which may benefit film producers and consumers alike. For instance, moviegoers curious about the flavor of a particular movie can learn more about its context by examining the history of its genre in the country it was released in over the years. Moreover, before deciding to purchase tickets for a screening, they can look up a director's gross worth and most successful movies to predict whether they would enjoy the director's current offering.

On the production side, producers and casting directors could use the same database to find the most prolific actors for a film genre and possibly recruit them for an upcoming project, or examine the track record of a certain production company to assess whether working with them would prove successful. Through these queries and more, users can gather insights and generalizations from the data that were not apparent on a granular level, which can ultimately serve to characterize the past, describe the present, and predict the future of the industry and its effects on stakeholders.

8. References

[1] Afzal, H. and Latif, M.H. 2016. *Prediction of Movies popularity Using Machine Learning Techniques.* International Journal of Computer Science and Network Security (IJCSNS) 16, 8 (Aug. 2016). Retrieved from https://www.researchgate.net/profile/Hammad-Afzal/publication/311913687_Prediction_of_Movies_popularity_Using_Machine_Learning_Techniques/links/586253ce08ae6eb871ab0748/Prediction-of-Movies-popularity-Using-Machine-Learning-Techniques.pdf

[2] Ali, A. 2020. *Understanding SQL Server Physical Joins.* (July, 2020). Retrieved from https://www.mssqltips.com/sqlservertip/2115/understanding-sql-server-physical-joins/

[3] Apache Software Foundation. n.d. *Apache NiFi*. Retrieved from https://nifi.apache.org/

[4] Apache Software Foundation. n.d. *PutSQL.* Retrieved from https://nifi.apache.org/docs/nifi-docs/components/org.apache.nifi/nifi-standard-nar/1.6.0/org.apache.nifi.processors.standard.PutSQL/

[5] Apache Software Foundation. n.d. *GenerateFlowFile.* Retrieved from https://nifi.apache.org/docs/nifi-docs/components/org.apache.nifi/nifi-standard-nar/1.5.0/org.apache.nifi.processors.standard.GenerateFlowFile/index.html

[6] Balke, W. and Maarry, K. n.d. *Data Warehousing & Data Mining*. Institute for Information Systems. Technische Universität Braunschweig. Retrieved from http://www.ifis.cs.tu-bs.de/webfm_send/2082

[7] Bioglio, L. and Pensa, R. 2018. *Identification of key films and personalities in the history of cinema from a Western perspective.* Applied Network Science, 3, 50, (Nov. 2018). Retrieved from https://appliednetsci.springeropen.com/articles/10.1007/s41109-018-0105-0

[8] Buehler, C. 2020. *What are the Phases of Film Production.* (Apr. 2020). Retrieved from https://www.ipr.edu/blogs/digital-video-and-media-production/what-are-the-phases-of-film-production/

[9] Cloudera Inc. 2020. *User Guide.* Retrieved from https://docs.cloudera.com/HDPDocuments/HDF3/HDF-3.1.0/bk_user-guide/content/scheduling-tab.html

[10] Combaudon, S. 2012. *Full Table Scan vs Full Index Scan Performance.* (Nov, 2012). Retrieved from https://www.percona.com/blog/2012/11/23/full-table-scan-vs-full-index-scan-performance/

[11] Demeter, R. 2021. *Which Movie Genres Earned the Most At the Box Office Between 1980 and 2020?*. (Apr. 2021). Retrieved from https://cuttingroommusic.com/2021/04/20/which-movie-genres-earned-the-most-at-the-box-office-between-1980-and-2020/

[12] Gibbs, J. and Pye, D. 2006. *Style and Meaning: Studies in the Detailed Analysis of Film.* Manchester University Press, Manchester. Retrieved from https://www.nottingham.ac.uk/scope/documents/2006/october-2006/book-rev-oct-2006.pdf

[13] Han, J., Kamber, M. and Pei, J. 2012. *Data Mining: Concepts and Techniques (3rd ed.). The Morgan Kaufmann Series in Data Management Systems, Vol 3.* Elsevier Inc., Amsterdam. DOI:10.1016/B978-0-12-381479-1.00017-4

[14] Hobbs, L., Hillson, S., Lawande, S. and Smith, P. 2005. *Oracle 10g Data Warehousing*. Elsevier Inc., Amsterdam. DOI:10.1016/B978-1-55558-322-4.X5000-0

[15] Hoogland, F. 2011. *How Oracle secretly changed multiblock reads.* (Nov. 2011). Retrieved from https://www.doag.org/formes/pubfiles/5053011/2013-DB-Frits_Hoogland-About_multiblock_reads-Manuskript.pdf

[16] IBM. 2014. *Table Access Strategy*. (Sept. 2014). Retrieved from https://www.ibm.com/docs/en/iodg/11.3?topic=pes-table-access-strategy

[17] IMDb.com, Inc. 2021. *What is IMDb?*. Retrieved from https://help.imdb.com/article/imdb/general-information/what-is-imdb/G836CY29Z4SGNMK5?ref_=helpart_nav_1#

[18] Johnson, E. and Jones, J. 2008. *A Developer's Guide to Data Modeling for SQL Server: Covering SQL Server 2005 and 2008 (1st. ed.)*. Addison-Wesley Professional, Boston.

[19] Kimball, R. and Ross, M. 2013. *The Data Warehouse Toolkit: The Definitive Guide to Dimensional Modeling (3rd ed.).* John Wiley & Sons, Inc., Indianapolis.

[20] Kubrak T. 2020. *Impact of Films: Changes in Young People's Attitudes after Watching a Movie.* Behav Sci (Basel) 10, 5 (May 2020). Retrieved from https://www.ncbi.nlm.nih.gov/pmc/articles/PMC7288198/

[21] Leone, S. 2020. *IMDb movies extensive dataset.* Retrieved from https://www.kaggle.com/stefanoleone992/imdb-extensive-dataset

[22] Liberty, D. 2019. *How To Calculate Average Sales.* (Nov. 2019). Retrieved from https://www.sisense.com/blog/how-to-calculate-average-sales/

[23] Luján-Mora, S., Trujillo, J. and Song, I. 2006. *A UML profile for multidimensional modeling in data warehouses.* Data & Knowledge Engineering, 59, 3 (Dec. 2006), 725-769. DOI:10.1016/j.datak.2005.11.004

[24] Malinowski E. and Zimányi E. 2004. *OLAP Hierarchies: A conceptual perspective.* Biomedical Sciences Instrumentation 3084: 477-491. (June 2004). Retrieved from https://link.springer.com/content/pdf/10.1007%2F978-3-540-25975-6_34.pdf

[25] Mapbox. 2018. *Maps in Tableau | Mapbox.* (Oct. 2018). Retrieved from https://www.mapbox.com/tableau/

[26] Motion Picture Association of America. 2016. *Theatrical Market Statistics.* Retrieved from: https://www.motionpictures.org/wp-content/uploads/2018/03/MPAA-Theatrical-Market-Statistics-2016_Final-1.pdf

[27] Motl, J., and Schulte, O. 2018. *The CTU Prague Relational Learning Repository.* arXiv:1511.03086v1. (Sept. 2018). Retrieved from https://arxiv.org/pdf/1511.03086.pdf

[28] Nash Information Services, LLC. 2021. *Domestic Movie Theatrical Market Summary 1995 to 2021*. Retrieved from https://www.the-numbers.com/market/

[29] Oracle Corporation. 2021. *Buffering and Caching.* Retrieved from https://dev.mysql.com/doc/refman/8.0/en/buffering-caching.html

[30] Oracle Corporation. 2021. *CHECKSUM TABLE Statement* Retrieved from https://dev.mysql.com/doc/refman/5.7/en/checksum-table.html

[31] Oracle Corporation. 2021. *CREATE TEMPORARY TABLE Statement.* Retrieved from https://dev.mysql.com/doc/refman/8.0/en/create-temporary-table.html

[32] Oracle Corporation. 2021. *Index Merge Optimization.* Retrieved from https://dev.mysql.com/doc/refman/8.0/en/index-merge-optimization.html

[33] Oracle Corporation. 2021. *InnoDB Startup Options and System Variables.* Retrieved from https://dev.mysql.com/doc/refman/5.7/en/innodb-parameters.html#sysvar_innodb_log_buffer_size

[34] Oracle Corporation. 2021. *ORDER BY Optimization.* Retrieved from https://dev.mysql.com/doc/refman/8.0/en/order-by-optimization.html#order-by-filesort

[35] Oracle Corporation. 2021. *Server System Variables.* Retrieved from https://dev.mysql.com/doc/refman/8.0/en/server-system-variables.html

[36] Oracle Corporation. 2021. *The Physical Structure of an InnoDB Index.* Retrieved from https://dev.mysql.com/doc/refman/8.0/en/innodb-physical-structure.html

[37] Pandas Development Team. 2021. *pandas.DataFrame.explode.* (Aug. 2021). Retrieved from https://pandas.pydata.org/docs/reference/api/pandas.DataFrame.explode.html

[38] Perovšek, M., Vavpetič, A., Kranjc, J., and Lavrač, N. 2015. *ILP Datasets*. (April 2015). Retrieved from http://kt.ijs.si/janez_kranjc/ilp_datasets/

[39] Redfern, N. n.d. *Genre trends in five European countries, 2006 to 2010.* Retrieved from https://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.393.7628&rep=rep1&type=pdf

[40] Sanders, G. L. and Shin, S.K. 2005. *Denormalization strategies for data retrieval from data warehouses.* Decision Support Systems 46, 1 (Oct. 2006), 267-282. DOI:10.1016/j.dss.2004.12.004

[41] Schwartz, B., Tkachenko, V. and Zaitsev, P. 2012. *Chapter 4. Optimizing Schema and Data Types*. *High Performance MySQL (3rd ed.).* O'Reilly Media, Inc., Massachusetts. Retrieved from https://www.oreilly.com/library/view/high-performance-mysql/9781449332471/ch04.html

[42] Silberschatz, A., Korth, H.F. and Sudarshan, S. 2020. *Database System Concepts (7th ed.).* McGraw-Hill Education, New York.

[43] Stanford University IT Department. 2005. *Oracle® Database Data Warehousing Guide (10.2)*. (Dec. 2005). Retrieved from https://web.stanford.edu/dept/itss/docs/oracle/10gR2/server.102/b14223.pdf

[44] Statista. n.d. *Most popular movie genres in the United States and Canada from 1995 to 2021, by total box office revenue.* Retrieved from https://www.statista.com/statistics/188658/movie-genres-in-north-america-by-box-office-revenue-since-1995/

[45] Tableau Software LLC. 2015. *Create Sets.* (Jan. 2015). Retrieved from https://help.tableau.com/current/pro/desktop/en-us/sortgroup_sets_create.htm

[46] Tableau Software LLC. 2020. *How Relationships Differ from Joins.* (May 2020). Retrieved from https://help.tableau.com/current/online/en-us/datasource_relationships_learnmorepage.htm

[47] Winand, M. 2012. *Hash Join.* (May 2012). From https://use-the-index-luke.com/sql/join/hash-join-partial-objects

[48] Yaddow, W. 2019. *ETL Test Automation Planning for DW/BI Projects.* (Mar. 2019). Retrieved from https://tdwi.org/Articles/2019/03/22/DIQ-ALL-ETL-Test-Automation.aspx?Page=3

[49] Yudelson, J. 2011. *The impact of actors and producers in studio-financed movie deals.* Journal of Behavioral Studies in Business, 3, 9 (Apr. 2011), 98-112. Academic and Business Research Institute, Florida. Retrieved from https://www.aabri.com/manuscripts/10698.pdf

9. Appendix

The link to the OLAP dashboard is given below:
https://cggl-imdb-dashboard.herokuapp.com/