MCO1: Query Processing Technical Report

Clarissa Albarracin[1], Amor De Guzman[2], Reina Althea Garcia[3], Miko Santos[4]

[1]clarissa_albarracin@dlsu.edu.ph, [2]amor_deguzman@dlsu.edu.ph, [3]reina_garcia@dlsu.edu.ph, [4]miko_santos@dlsu.edu.ph

ABSTRACT

In today's rapidly evolving digital landscape, organizations are burdened with large amounts of data gathered daily from various sources. Effective management of such databases through appropriate design and query optimization have become critical for deriving meaningful analytics and aiding strategic decision-making. This paper delves into key aspects of modern data infrastructure and analysis, including data warehousing, Extract-Transform-Load (ETL) pipelines, Online Analytical Processing (OLAP) operations, and query processing and optimization techniques. Throughout this project, industry-standard tools such as AI Studio (formerly known as RapidMiner) for extracting, transforming and loading data into the data warehouse, MySQL Workbench for running SQL scripts, and Tableau Public for visualizing data analytics were used. Using the Steam Games dataset, the group seeks to develop an interactive and dynamic dashboard application to support end-users in exploring games of their interest. By exploring these interconnected components, the group aims to provide insights into building robust data ecosystems that can transform raw information into actionable business intelligence.

Keywords

Data Warehouse, ETL, OLAP, Query Processing, Query Optimization

1. Data Warehouse

Large volumes of structured data from multiple sources can be stored centrally in a data warehouse, making data readily available for reporting and analysis [6]. Combining data makes analysis and querying more effective. In contrast with traditional databases, which are designed for everyday use, data warehouses are made especially for complex queries and the analysis of historical data. Before being loaded into the warehouse, data is cleaned and processed using the Extract, Transform, Load (ETL) process, ensuring accuracy and consistency for analysis.

1.1 Source Dataset

The source dataset was obtained from Kaggle and consists of a dataset related to Steam, the largest gaming platform on PC. This dataset contains 39 columns that provide essential information about various games, including attributes such as the game name, release date, developers, genres, categories, and more.

The dataset is available in two file formats: JSON and CSV. The JSON format includes additional information, specifically packages, which were not present in the CSV format. However, the CSV file had alignment issues, since values did not correspond correctly to their respective columns (e.g., values meant for column 1 were placed under column 2). Despite this, the group chose to work with the CSV format due to the challenges in importing the large JSON file into MongoDB, which required more complex coding solutions rather than a straightforward MongoDB import.

To address the misalignment issue in the CSV file, the group manually fixed the column headers to match the underlying values by cross-referencing the data with the JSON format, using a text editor (VSCode) for comparison. Additionally, the CSV file lacked a header for the last column, which contained information about movie links where the games were featured. This missing header caused the movie-related data to be excluded during extraction in the ETL process. To resolve this, the group added the appropriate header, "movies", to ensure that all data would be captured correctly in subsequent steps.

Afterward, the group created a relational database for the source data, implemented in MySQL. The database consists of 11 tables. Some attributes, specifically those with values as links or text, such as screenshots and movies where the games were featured, were not included as they were deemed insignificant for the group's intended analysis. The tables are as follows:

Table 1. Stream Games Dataset

Tables	Description
`Games`	List of games in the dataset.
`Ratings`	Ratings data for each game.
`Playtime`	Average and median playtime for each game.
`Developers`	List of developers.
`Games_has_Developers`	Links games to their respective developers.
`Publishers`	List of publishers.
`Games_has_Publishers`	Links games to their respective publishers.
`Genre`	List of genres.
`Games_has_Genre`	Links games to their respective genres.
`Categories`	List of categories.
`Games_has_Categories`	Links games to their respective categories.

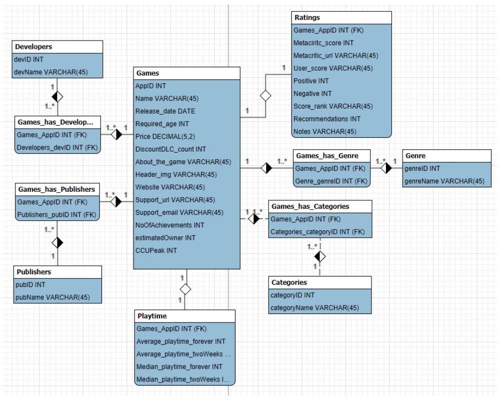


Figure 1. Relational Database Schema for the Steam Games Dataset

1.2 Dimensional Model

The dimensional model used for this data warehouse is essential for enabling OLAP operations, such as slice, dice, pivot, roll-up, and drill-down. The group has chosen a `snowflake schema` for its structured approach and effectiveness in supporting these types of operations. In a snowflake schema, the fact table holds quantitative data and is linked to a core dimension table, which in turn connects to other dimension tables. This approach normalizes the data, reducing redundancy while still maintaining the necessary context for analysis.

While some tables such as genres, categories, developers, and publishers, have many-to-many relationships with the central game dimension, directly referencing them within a single dimension table would lead to data redundancy and inefficiencies. To address this, the group employed `bridge tables`—intermediary tables that manage these many-to-many relationships between dimensions. The bridge tables connect the core dimension (e.g., `dim_game`) to other dimension tables (e.g., genre, category) without duplicating data [2]. This allows the fact table to indirectly link to multiple related dimensions, maintaining a clean and efficient design.

The `FACT_STREAM` table serves as the center of the data warehouse, containing quantitative metrics that provide insights into game performance based on user rating and user engagement. The primary contents of this fact table is shown in Table 2.

Table 2. FACT_STREAM Table Attributes

Attributes	Description
`ave_playtime_forever`	Indicates the average amount of time players have spent on a game.
`recommendation_count`	Reflects the number of recommendations received by the game.
`metacritic_score`	Score from metacritic.
`positive_count`	Number of players that have rated the game positively.
`negative_count`	Number of players that have rated the game negatively.
`peak_CCU`	Indicates the highest number of players who were playing the game simultaneously.
`Price`	The retail price of the game.
`Estimated_owners`	Estimates the number of unique owners of the game.
`AppID`	Foreign key that links each record in the fact table to the corresponding game in the DIM_GAME dimension.

The dimensions in this schema provide descriptive context for the fact table and allow for meaningful analysis by categorizing and organizing the data. Each dimension represents a different aspect of the games and their attributes, enabling detailed insights and breakdowns of the quantitative data in the fact table. The primary dimensions in the schema are described in Table 3.

Table 3. Dimension Tables of Stream Games Dataset

Table	Description
`DIM_GAME`	This contains information about the games, with attributes such as, `AppID`, `Name`, and `Release_date`
`DIM_GENRE`	This contains all the genres of a game. Each genre has a unique `genreID` and `genreName`
`DIM_CATEGORY`	This contains all the categories a game may belong to. Each category has a unique `categoryID` and `categoryName`
`DIM_DEVELOPER`	This contains information about developers, such as `devID` and `devName`.
`DIM_PUBLISHER`	This contains information about publishers, such as `pubID` and `pubName`

Each dimension in this schema has a flat hierarchy, meaning there are no multiple levels within the dimensions themselves. However, bridge tables (e.g., `BRIDGE_GAME_GENRE`, `BRIDGE_GAME_CATEGORY`) are used to manage the many-to-many relationships between games and their attributes, such as genres, categories, developers, and publishers. This approach ensures that a game can be associated with multiple values in these dimensions without duplicating data.

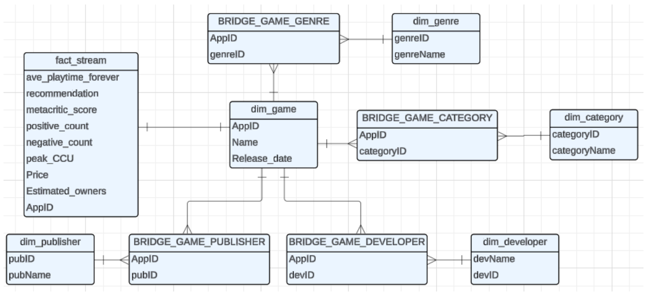


Figure 2. Dimensional Model: Snowflake schema

The dimensions chosen for this schema allow for a structured and meaningful analysis of the games and their key characteristics. The `DIM_GAME` dimension is essential as it provides the basic information about each game, such as its name and release date.

The `DIM_GENRE` dimension is included to capture the type of game, such as action, adventure, or strategy. Understanding which genres are popular aids in analyzing player preferences and market trends. Similarly, the `DIM_CATEGORY` dimension allows us to classify games by categories, such as whether they are multiplayer, single-player, or VR-compatible, therefore aiding the analysis of how different game types perform.

The `DIM_DEVELOPER` and `DIM_PUBLISHER` dimensions provide important information about who develops and distributes the games. By including these dimensions, we can assess whether certain developers or publishers consistently release successful games, providing insights for future game releases.

The `FACT_STREAM` table focuses on the key performance metrics of the games, such as average playtime, recommendation counts, and review scores. These numbers give a clear view of how well a game is performing and how engaged users are. By linking these metrics to the dimensions, the schema supports a wide range of analysis, allowing us to determine which games performed well based on their metric scores and which genres or categories, or which developers, created popular games, and vice versa.

Initially we opted to use star schema instead of snowflake schema; however while creating the design, we realized that there are many-to-many relationships between game dimensions and other dimensions. Initially, it was all placed in one table; however, the issue of data redundancy occurred and would greatly affect the analysis later on. Therefore, we decided to use bridge tables, as stated in the beginning of this section, to normalize these many-to-many relationships. This approach helped reduce data redundancy and ensured that each game dimension (e.g., genre, category, developer, and publisher) could be linked efficiently without duplicating data across tables. By normalizing the schema with bridge tables, we improved the accuracy and scalability of the schema, which in turn supports more efficient data analysis and minimizes the risk of inconsistent or duplicated data during query operations.

2. ETL Script

At first, the group decided to use the JSON format for data loading; however, it turned out to be too large for **`mongoimport`** (a MongoDB utility for importing data) to handle efficiently. This led the group to switch to using the CSV format for loading into the source database. After creating the source database with a relational model and the data warehouse with a snowflake schema, the group's chosen file format, CSV, was loaded into the source database. The first ETL pipeline was designed to fix misalignment issues, as mentioned in Section 2. The columns were mismatched with their values, as the values for column X were incorrectly placed in column Y, causing a shift in the subsequent columns. The first pipeline consists of reading the CSV, renaming columns to their appropriate names to match the values, and selecting the attributes to be written on CSV.

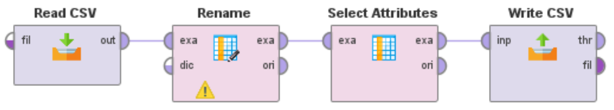


Figure 3. ETL Pipeline for Fixing CSV Issues

After correcting the file, it was then loaded into the source database using various operators in AI Studio (RapidMiner), such as `Split`, `De-Pivot`, `Join`, and others. These operators were employed to ensure the data was accurately transformed and organized for analysis. Figure 4 shows a snippet of the ETL pipeline created and utilized for this purpose, highlighting each step involved in processing the data to achieve a clean and consistent format.

For **extraction**, `Read CSV` was used to load and encode the corrected CSV file, which contained key information such as game details, developer information, genres, and user ratings. This data is important for the model, as it populates the dimensions of the snowflake schema, enabling a comprehensive analysis of the games' performance.

For **transformation**, `Select Attributes` was applied to choose the necessary columns for specific tables, which would later be populated using a `Write Database` operator. `Filter` Expression was then used to filter out and remove null values from the final result sets. `Replace` was employed for developers and publishers because their names often included terms like 'Inc,' 'LLC,' or 'Ltd,' which contain commas. This caused issues when using the `Split` operator, as these company names were mistakenly separated. The `Split` operator was necessary to handle multiple developers or publishers listed in a single column, but we needed to ensure that names with 'Inc,' 'LLC,' or 'Ltd' were not unintentionally split, enabling better separation and analysis of individual entities. Since the `Split` operator caused duplicate columns of the same type or attribute, `De-Pivot` was used to combine them back into a single table, ensuring that the `AppID` remained consistent in all entries. This allowed for a clean representation of data where each game and its associated attributes were aligned correctly, facilitating accurate analysis and reporting.

Additionally, the `JOIN` operator was used for tables with many-to-many relationships, allowing the effective combining of data from different columns while maintaining the integrity of the relationships between entities. For example, this operator facilitated the linking of games with their respective developers, publishers, categories and genres, ensuring that all relevant data was accessible in a unified format. By employing the `JOIN` operator, we were able to create a more comprehensive dataset that supported in-depth analysis, allowing us to explore correlations and trends across multiple attributes within the data warehouse.

For **loading**, the `Write Database` operator was used to load the processed data into the source database, while maintaining key constraints such as primary key integrity (ensuring the `AppID` remained consistent across all tables) and referential integrity between related tables.

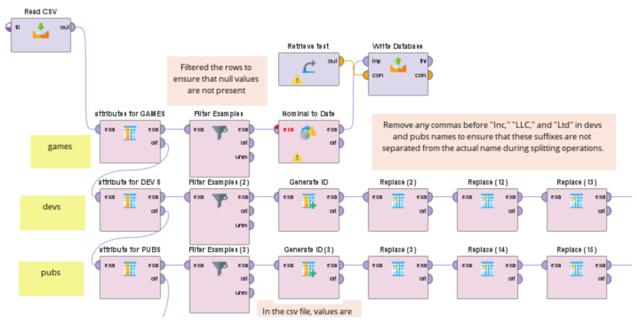


Figure 4. Snippet of ETL Pipeline for Loading into SourceDB

After loading into the source database, the data was then transferred to the data warehouse, which is structured using a snowflake schema consisting of *1 fact table, 5 dimension tables, and 4 bridging tables*. The ETL pipeline for this step is relatively straightforward, as the primary transformations and data fixes were already handled in the earlier ETL process. This pipeline focuses on efficiently mapping and loading the cleaned data into the fact and dimension tables, ensuring that relationships are preserved. The `fact table` stores the key metrics and performance data, while the dimension tables contain descriptive attributes, such as game details, developers, publishers, genres, and categories. The `bridging tables`, on the other hand, manage the many-to-many relationships between different entities, like games and their associated developers or genres, ensuring a normalized and optimized schema for querying.

The ETL process involved several key operators. First, the `Read Database` operator was used to connect to the source database in `MySQL`, enabling the extraction of the previously loaded and cleaned data. Next, the `Select Attributes` operator was employed to filter and select only the necessary columns required for populating the data warehouse, ensuring that only relevant information was transferred. Finally, the `Write to Database` operator was utilized to load the selected data into the snowflake schema in the data warehouse. This step mapped the data to the corresponding fact and dimension tables, ensuring the relationships between entities were accurately reflected.

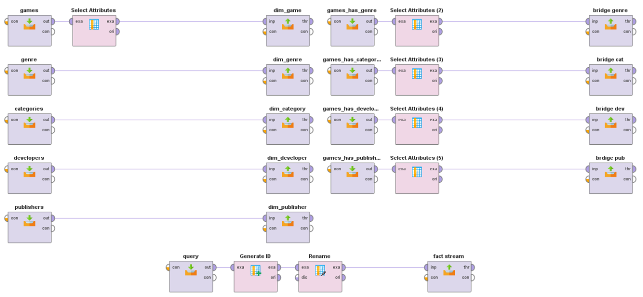
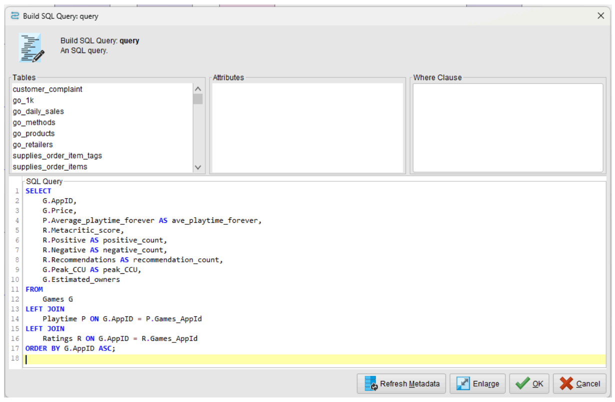


Figure 5. ETL Pipeline for Loading into Data Warehouse

To add to that, when loading data into the fact table, instead of simply reading the entire table as was done with other `Read Database` operators, a more efficient approach was taken by utilizing a `query`. This query allowed for precise extraction of only the necessary data needed for the fact table, filtering out irrelevant information and improving the overall performance of the ETL process.



customer_complaint
go_1k
go_daily_sales
go_methods
go_products
go_retailers
supplies_order_item_tags
supplies_order_items

```sql
SELECT
    G.AppID,
    G.Price,
    P.Average_playtime_forever AS ave_playtime_forever,
    R.Metacritic_score,
    R.Positive AS positive_count,
    R.Negative AS negative_count,
    R.Recommendations AS recommendation_count,
    G.Peak_CCU AS peak_CCU,
    G.Estimated_owners
FROM
    Games G
LEFT JOIN
    Playtime P ON G.AppID = P.Games_AppId
LEFT JOIN
    Ratings R ON G.AppID = R.Games_AppId
ORDER BY G.AppID ASC;
```

Figure 6. Query-based Extraction for Loading into Fact Table

The issues weren't that many, as the group was already familiar with the usage of `AI Studio` (RapidMiner), which made it relatively easy for us to establish connections and use operators. However, there were still a few issues that we encountered during the process. One of the key issues involved generating a new ID attribute. Initially, we used the Generate ID operator to create unique identifiers for each record. However, we noticed a significant problem: each time the operator was used, it would overwrite or remove the previously generated IDs, leading to the loss of earlier data and inconsistencies in the records, which disrupted the process.

To address this issue, we decided to switch from using the Generate ID operator to the Generate Attributes operator. By doing so, we were able to create unique identifiers that remained intact across multiple operations. This approach allowed us to define our custom logic for generating IDs, ensuring that no previously generated IDs were overwritten or deleted. As a result, the records remained consistent, and we successfully avoided any issues related to the loss of previous ID values.

Additionally, we encountered an issue with the split operator, which incorrectly separated publisher and developer names that contained commas, such as "Example, Inc." This led to unintended splits and data integrity problems. To address this, the group implemented a regex replacement using the replace operator to modify the commas in these names. Specifically, comma was replaced with a space when it was followed by terms like "LLC" or "Inc." This transformation ensured that names like "Nintendo, LLC" were converted to "Nintendo LLC," effectively preventing the split operator from misinterpreting these names as separate fields and maintaining the integrity of the data throughout the ETL process.

In terms of progressive loading, our group's ETL process requires running the entire ETL script to load updates into the data warehouse. This involves re-reading the CSV file format and rewriting all existing records each time an update is performed. This approach ensures that the data in the warehouse remains synchronized with the source file, capturing any changes or new entries to maintain accuracy and completeness. By re-reading the CSV for each update, we make sure the data warehouse reflects the most current information available.

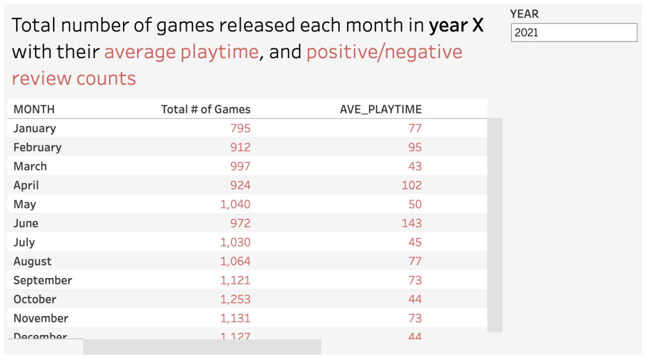
3. OLAP Application

The purpose of the OLAP application is to generate comprehensive analytics that could aid stakeholders in the gaming industry through the analysis of game popularity, user feedback and different performance metrics across genres and metrics tailored specifically for the developers. Through the application, stakeholders could evaluate how specific games, and by extension, how game developers perform, enabling them to make informed business decision strategies in relation to marketing, development and investment in their own business environment.

The analytical tasks involved in the application are identification of which: (1) games released each month in year X, along with their average playtime and positive/negative review counts, indicating which games have the potential to have further investment; (2) games have performed well critically in specific genre through utilizing the metacritic scores; (3) developers have consistently developed and produced games that are well-received and reviewed; and lastly, (4) games garner a high potential in terms of popularity ranking. The four reports listed below facilitates generating of such insightful analytics as stated.

The corresponding SQL statements, along with the advanced SQL constructs utilized and their purposes, as well as an explanation of how OLAP operations (roll-up, drill-down, dice, slice) were employed in the queries, can be found in Section 4.

3.1 Total number of games released each month in year X with their average playtime, and positive/negative review counts



Total number of games released each month in **year X** with their average playtime, and positive/negative review counts

MONTH	Total # of Games	AVE_PLAYTIME
January	795	77
February	912	95
March	997	43
April	924	102
May	1,040	50
June	972	143
July	1,030	45
August	1,064	77
September	1,121	73
October	1,253	44
November	1,131	73
December	1,127	44

Figure 7. Tableau Dashboard query for the total

number of games released each month in year X, including their average playtime, and counts of positive and negative reviews.

This Tableau Dashboard query in Figure 7 displays the number of games released each month given a year X (in the example, 2021) along with the average playtime of users, the positive review counts, and the negative review counts of all these games. This information is extracted from tables FACT_STREAM and DIM_GAME, which are joined on AppID. This query employs the `DRILL-DOWN, ROLL-UP` and `SLICE` OLAP operations. The slice operation is achieved through the WHERE clause, which filters the data to only include games released in X year. This restricts the dataset to a specific subset based on the Release_date dimension, focusing solely on games from that year. Next, the query performs a drill-down operation by breaking down the yearly data into monthly summaries with the GROUP BY release_month clause. Lastly, because it summarizes data for each month, this query also demonstrates a roll-up operation, aggregating total games, average playtime, and review counts by month. This query allows for the analysis of whether games released on a specific month are more likely to become popular compared to games released on different months.

3.2 Games with a Metacritic score greater than X in the Y genre, and with estimated owners greater than Z.

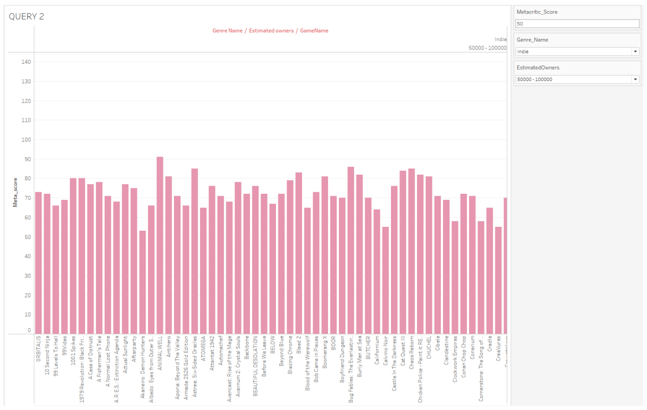


Figure 8. Tableau Dashboard Query for Games with a Metacritic score greater than X in the Y genre, and with estimated owners greater than Z.

The query retrieves video games by selecting those with a Metacritic score greater than **X** (in the example, 50), belonging to the **Y** genre (Indie, for instance), and having an estimated owner count within the range of **Z** (specifically between 50,000 and 100,000). It extracts the game name, release date, Metacritic score, estimated owners, and genre name from multiple tables, including `FACT_STREAM`, `DIM_GAME`, `BRIDGE_GAME_GENRE`, and `DIM_GENRE`, joined on their respective identifiers. The query employs the `SLICE` OLAP operation by filtering the dataset to include only games categorized as "Indie," thereby narrowing the focus to a specific genre. Additionally, it utilizes the `DICE` OLAP operation to further refine the data by considering only those records with a Metacritic score exceeding Y and estimated owners within the specified range Z, resulting in a more relevant subset for analysis. This targeted approach provides insights into the performance of **genre X** games, helping developers and consumers understand which genres are the most sought after.

3.3 Developers with Game/s of Genre *X* that have at least *Y* positive reviews

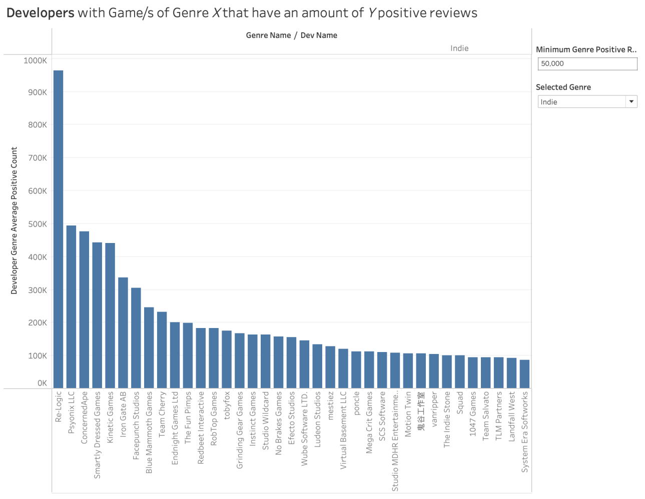


Figure 9. Tableau Dashboard Query for Developers with Game/s of Genre X that have at least Y positive reviews

Given a specific genre and target amount of positive reviews, Figure 9 shows all game developers whose game/s in Genre *X* have an average amount of *Y* positive reviews. The query initially utilizes a `SLICE` operation on the game genre dimension. Rows are filtered using the criterion where the genre is that of the one specified (in this case, it is Indie) and have a minimum of *Y* positive reviews (set to 50,000 in the sample query). Results are then grouped by developer and genre to obtain the average positive reviews among all apps for each developer. Once grouped, `ROLLUP` is implemented when the average game-genre positive reviews are aggregated. Finally, another `SLICE` operation is then used to filter averages which match the set positive review target.

3.4 Top 100 Games in X genre based on Popularity

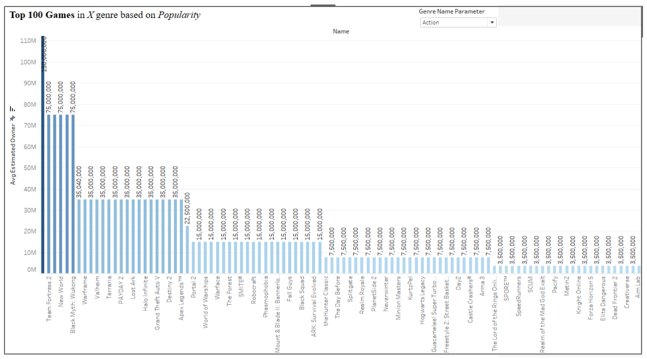


Figure 10. Tableau Dashboard Query for Top 100 Games Y in X genre based on Popularity

Given a specific genre Y, Figure 4 shows the list of games X belong in the top 100 in regards to the popularity metric. For filtering the top 100 games, an adjustment was conducted to the value for the `Top` attribute on `Filter By Field`, which was assigned a numeric value of 124 instead of raw 100 in order to return exactly 100 (rows) unique games after filtering. Failure to do so will return a result of 74 rows only which is not what is expected. The `SLICE` operation was applied for this query to select a single dimension from the multidimensional dataset through the `WHERE` clause which filters the result to a subset of data specifically only including games belonging to the genre Y, which in the figure is the `Action` genre . However, the selection for specific genre Y is not limited to the mentioned one, it could still be facilitated dynamically within the application dashboard in `Tableau` by editing the parameter for `Gender Name Parameter`.

4. Query Processing and Optimization

`Query optimization` is defined as the process of improving both the efficiency and performance of fetching database queries. When dealing with small-scale databases, fetch time may be negligible varying from milliseconds. However, larger scale databases with complex models can significantly slow down query performance resulting in a slower overall application. For this project, query optimization strategies are applied such as using indexes, restructuring queries, and using `WHERE` instead of `HAVING`, among other strategies.

4.1 Database Level Optimization

In this project, database level optimization was achieved through indexing and normalization. Indexing enhanced data retrieval speed by allowing quick access to records, whereas normalization improved data integrity and reduced redundancy by organizing data into related tables.

4.2 Query Formulation

The subsequent optimization phase involves applying effective SQL querying strategies which entails enhancing and structuring queries to improve the query performance at the database level. This consists of simplifying complex queries and fixing structured subqueries, and addition of indexes.

4.2.1 Total number of games released each month in year X average playtime, and positive/negative review counts

The unoptimized query utilized the `JOIN` operation only to associate the `FACT_STREAM` and `DIM_GAME` tables which is essential for integrating the gameplay metric with the game-specific information. Relying solely on the mentioned operation, considering that our data source contains a large volume of data, may result in an execution with full table scan for matching rows between fact and dimensional table, slowing down the data retrieval time. Further, without the use of indices, performance could degrade further.

Listing 1. Unoptimized SQL Script for Query 4.2.1

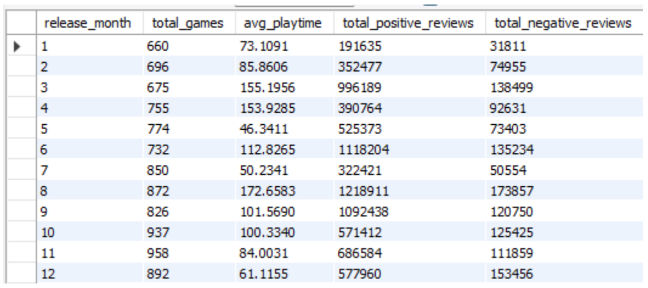
```
SELECT
    MONTH(g.Release_date) AS release_month,
    COUNT(fs.fact_id) AS total_games,
    AVG(fs.ave_playtime_forever) AS avg_playtime,
    SUM(fs.positive_count) AS total_positive_reviews,
    SUM(fs.negative_count) AS total_negative_reviews
FROM FACT_STREAM fs JOIN DIM_GAME g ON fs.AppID = g.AppID
WHERE YEAR(g.Release_date) = XXXX
GROUP BY release_month
ORDER BY release_month ASC;
```

On the other hand, considering the actual number of rows in the two tables, which total 97,268, the optimized version employs indexing as an optimization technique on critical attributes such as AppID and Release_date. This enhancement improves query performance by facilitating rapid retrieval and minimizing full table scans. The query implements a drill-down analysis by aggregating data on a monthly basis. It filters the results for the total number of games released in year X, providing insights into total games, average playtime, and review counts (both positive and negative) for each month. By grouping the results by release month, the query allows for a detailed view of game performance over time.

Listing 2. Optimized SQL Script for Query 4.2.1

```
CREATE INDEX idx_release_date ON DIM_GAME(Release_date);
CREATE INDEX idx_fact_stream_appid ON FACT_STREAM(AppID);
CREATE INDEX idx_dim_game_appid ON DIM_GAME(AppID);

SELECT
    MONTH(g.Release_date) AS release_month,
    COUNT(fs.fact_id) AS total_games,
    AVG(fs.ave_playtime_forever) AS avg_playtime,
    SUM(fs.positive_count) AS total_positive_reviews,
    SUM(fs.negative_count) AS total_negative_reviews
FROM FACT_STREAM fs JOIN DIM_GAME g ON fs.AppID = g.AppID
WHERE YEAR(g.Release_date) = 2020
GROUP BY release_month
ORDER BY release_month ASC;
```



release_month	total_games	avg_playtime	total_positive_reviews	total_negative_reviews
1	660	73.1091	191635	31811
2	696	85.8606	352477	74955
3	675	155.1956	996189	138499
4	755	153.9285	390764	92631
5	774	46.3411	525373	73403
6	732	112.8265	1118204	135234
7	850	50.2341	322421	50554
8	872	172.6583	1218911	173857
9	826	101.5690	1092438	120750
10	937	100.3340	571412	125425
11	958	84.0031	686584	111859
12	892	61.1155	577960	153456

Figure 11. Results snippet for optimized SQL Script for Query 4.2.1

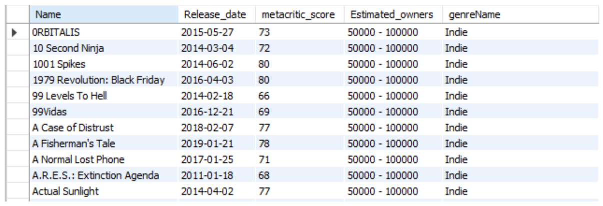
This query retrieves a comprehensive overview of games released each month in year X, summarizing key metrics such as total games, average playtime, and positive and negative review counts. By indexing critical attributes like Release_date and AppID, the analysis facilitates efficient data retrieval. The results provide insights into user engagement and satisfaction, highlighting trends in game performance that could present valuable marketing opportunities for stakeholders.

4.2.2 Games with a Metacritic score greater than X in the Y genre, and with estimated owners greater than Z.

The query retrieves a dynamic list of games with a Metacritic score greater than 50, belonging to the "Indie" genre, and with estimated owners between 50,000 and 100,000. The values for the Metacritic score, genre, and ownership range are not fixed and can be adjusted as needed for different analyses. It joins the FACT_STREAM, DIM_GAME, BRIDGE_GAME_GENRE, and DIM_GENRE tables to obtain game names, release dates, Metacritic scores, estimated owners, and genre names. However, this version of the query is unoptimized, potentially leading to slower performance, especially with large datasets. Optimization through indexing and query restructuring can improve execution time.

Listing 3. Unoptimized SQL Script for Query 4.2.2

```
SELECT g.Name,
       g.Release_date,
       f.metacritic_score,
       f.Estimated_owners,
       d.genreName
FROM   FACT_STREAM f JOIN DIM_GAME g ON f.AppID = g.AppID
          JOIN BRIDGE_GAME_GENRE bg ON g.AppID = bg.AppID
          JOIN DIM_GENRE d ON bg.genreID = d.genreID
WHERE f.metacritic_score > 50 AND d.genreName = 'Indie'
            AND f.Estimated_owners IN ('50000 - 100000');
```



	Name	Release_date	metacritic_score	Estimated_owners	genreName
▶	0RBITALIS	2015-05-27	73	50000 - 100000	Indie
	10 Second Ninja	2014-03-04	72	50000 - 100000	Indie
	1001 Spikes	2014-06-02	80	50000 - 100000	Indie
	1979 Revolution: Black Friday	2016-04-03	80	50000 - 100000	Indie
	99 Levels To Hell	2014-02-18	66	50000 - 100000	Indie
	99Vidas	2016-12-21	69	50000 - 100000	Indie
	A Case of Distrust	2018-02-07	77	50000 - 100000	Indie
	A Fisherman's Tale	2019-01-21	78	50000 - 100000	Indie
	A Normal Lost Phone	2017-01-25	71	50000 - 100000	Indie
	A.R.E.S.: Extinction Agenda	2011-01-18	68	50000 - 100000	Indie
	Actual Sunlight	2014-04-02	77	50000 - 100000	Indie

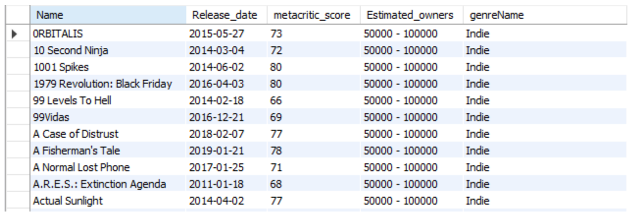
Figure 12. Results snippet for unoptimized SQL Script for Query 5.2.2

The optimized version of the query enhances performance by creating **indexes** on the columns most frequently used in the WHERE clause: metacritic_score, Estimated_owners, and genreName. These indexes allow the database to quickly retrieve relevant records, reducing the need for full table scans. The query retrieves a dynamic list of games based on user-defined conditions, such as a Metacritic score greater than 50, belonging to the "Indie" genre, and with estimated owners between 50,000 and 100,000. These values can be easily changed depending on what the user is looking for, making the query flexible for different analysis needs. With the addition of indexing, the execution time is significantly reduced, especially when dealing with larger datasets.

Listing 4. Optimized SQL Script for Query 4.2.2

```
CREATE INDEX idx_factstream_metacritic_score ON
FACT_STREAM(metacritic_score);
CREATE INDEX idx_factstream_estimated_owners ON
FACT_STREAM(Estimated_owners);
CREATE INDEX idx_dimgenre_genrename ON
DIM_GENRE(genreName);

SELECT g.Name,
       g.Release_date,
       f.metacritic_score,
       f.Estimated_owners,
       d.genreName
FROM   FACT_STREAM f JOIN DIM_GAME g ON f.AppID = g.AppID
          JOIN BRIDGE_GAME_GENRE bg ON g.AppID = bg.AppID
          JOIN DIM_GENRE d ON bg.genreID = d.genreID
WHERE f.metacritic_score > 50 AND d.genreName = 'Indie'
            AND f.Estimated_owners IN ('50000 - 100000');
```



	Name	Release_date	metacritic_score	Estimated_owners	genreName
▶	0RBITALIS	2015-05-27	73	50000 - 100000	Indie
	10 Second Ninja	2014-03-04	72	50000 - 100000	Indie
	1001 Spikes	2014-06-02	80	50000 - 100000	Indie
	1979 Revolution: Black Friday	2016-04-03	80	50000 - 100000	Indie
	99 Levels To Hell	2014-02-18	66	50000 - 100000	Indie
	99Vidas	2016-12-21	69	50000 - 100000	Indie
	A Case of Distrust	2018-02-07	77	50000 - 100000	Indie
	A Fisherman's Tale	2019-01-21	78	50000 - 100000	Indie
	A Normal Lost Phone	2017-01-25	71	50000 - 100000	Indie
	A.R.E.S.: Extinction Agenda	2011-01-18	68	50000 - 100000	Indie
	Actual Sunlight	2014-04-02	77	50000 - 100000	Indie

Figure 13. Results snippet for optimized SQL Script for Query 4.2.2

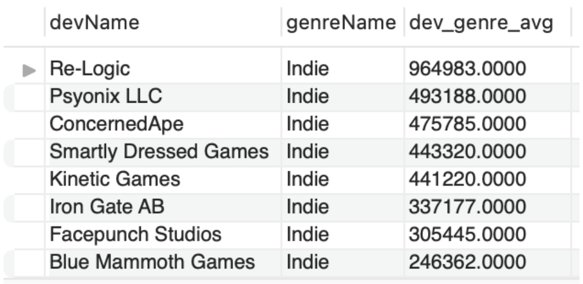
4.2.3 Developers with Game/s of Genre *X* that have at least *Y* positive reviews

To facilitate the user's search on well-known developers, this query retrieves all developers found to have game/s of a specified genre which have at least a *Y* amount of positive reviews. For this sample, games of the Indie genre with a minimum of 50,000 positive reviews are queried.

The query combines data from five tables (`FACT_STREAM`, `BRIDGE_GAME_GENRE`, `DIM_GENRE`, `BRIDGE_GAME_DEVELOPER`, and `DIM_DEVELOPER`) using multiple `JOIN` operations. This approach, while comprehensive, potentially generates a very large intermediate dataset. The size of this dataset can lead to efficiency issues, primarily due to the extensive data scanning and processing required.

Listing 5. Unoptimized SQL Script for Query 4.2.3

```
SELECT dev.devname,
       genre.genrename,
       AVG(fs.positive_count)
FROM   FACT_STREAM AS fs
       JOIN BRIDGE_GAME_GENRE AS bgg
         ON fs.appid = bgg.appid
       JOIN DIM_GENRE AS genre
         ON bgg.genreid = genre.genreid
       JOIN BRIDGE_GAME_DEVELOPER AS bgd
         ON fs.appid = bgd.appid
       JOIN DIM_DEVELOPER AS dev
         ON bgd.devid = dev.devid
GROUP  BY dev.devname,
          genre.genrename
HAVING AVG(fs.positive_count) > 50000
       AND genre.genrename = 'Indie'
ORDER  BY AVG(fs.positive_count) DESC,
          dev.devname,
          genre.genrename;
```



	devName	genreName	dev_genre_avg
▶	Re-Logic	Indie	964983.0000
	Psyonix LLC	Indie	493188.0000
	ConcernedApe	Indie	475785.0000
	Smartly Dressed Games	Indie	443320.0000
	Kinetic Games	Indie	441220.0000
	Iron Gate AB	Indie	337177.0000
	Facepunch Studios	Indie	305445.0000
	Blue Mammoth Games	Indie	246362.0000

Figure 14. Results snippet for unoptimized SQL Script for Query 4.2.3

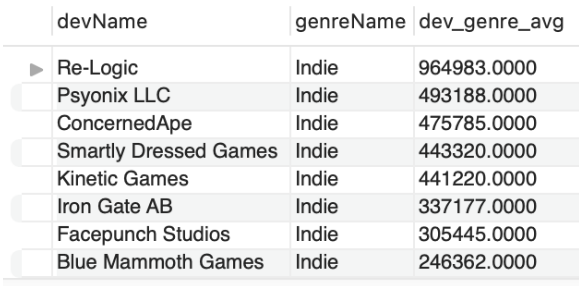
[5] The first step of optimization is the creation of an index on the `genreID` column of the `BRIDGE_GAME_GENRE` table. This will facilitate faster lookup and eliminate full table scans. To reduce large initial dataset size, a subquery is implemented to limit `JOIN` operations. The subquery combines all bridge tables with the fact table and performs a `DICE` operation to only select the columns relevant to the query (`appID`, `positive_count`, `genreID`, and `devID`).

The main query deals with the filtering of the data to conform with the conditions set using both WHERE and HAVING clauses. The genre condition is moved to a WHERE clause to pre-aggregate data before performing other operations.

Listing 6. Optimized SQL Script for Query 4.2.3

```
CREATE INDEX idx_bgg_genreID
  ON BRIDGE_GAME_GENRE(genreID);

SELECT dev.devname,
       genre.genrename,
       AVG(sub_fs.positive_count) AS dev_genre_avg
FROM   (SELECT fs.appID,
               fs.positive_count,
               bgg.genreID,
               bgd.devID
        FROM   FACT_STREAM AS fs
               JOIN BRIDGE_GAME_GENRE AS
                    bgg
                 ON fs.appID = bgg.appID
               JOIN BRIDGE_GAME_DEVELOPER
                    AS bgd
                 ON fs.appID = bgd.appID)
         AS sub_fs
       JOIN DIM_GENRE AS genre
         ON sub_fs.genreID = genre.genreID
       JOIN DIM_DEVELOPER AS dev
         ON sub_fs.devID = dev.devID
WHERE  genre.genrename = 'Indie'
GROUP  BY dev.devname,
          genre.genrename
HAVING dev_genre_avg > 50000
ORDER  BY dev_genre_avg DESC,
          dev.devname,
          genre.genrename;
```



devName	genreName	dev_genre_avg
Re-Logic	Indie	964983.0000
Psyonix LLC	Indie	493188.0000
ConcernedApe	Indie	475785.0000
Smartly Dressed Games	Indie	443320.0000
Kinetic Games	Indie	441220.0000
Iron Gate AB	Indie	337177.0000
Facepunch Studios	Indie	305445.0000
Blue Mammoth Games	Indie	246362.0000

Figure 15. Results snippet for optimized SQL Script for Query 4.2.3

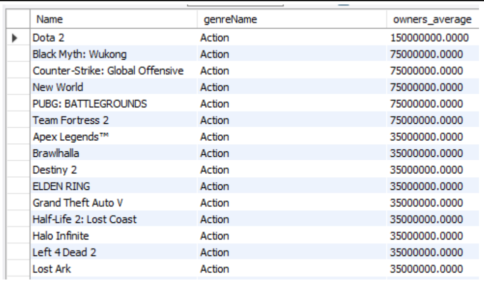
4.2.4 Top 100 Games in X genre based on Popularity

For the analysis of game ownership statistics, this query obtains the list of action games alongside their estimated ownership averages. The unoptimized query only utilizes the JOIN operation to associate games names with their respective estimated ownership counts, segregated by genre. However, this approach can result in excessive data retrieval which could negatively impact query performance. Further, the use of this operation could result in unnecessary records being returned which further hinders performance.

In constructing the SQL script, advanced SQL constructs such as JOIN and SUBSTRING_INDEX function were utilized to generate the aforementioned report. The former was used to combine the tables FACT_STREAM, DIM_GAME, BRIDGE_GAME_GENRE and DIM_GENRE. The latter parsed the ESTIMATED_OWNERS column since it contains both the upper and lower range for that attribute, which is necessary for calculating the average estimated ownership. By averaging the estimated number of owners, a more balanced estimate of the popularity is created by considering both extreme bounds rather than sticking to just one bound.

Listing 7. Unoptimized SQL Script for Query 4.2.4

```
SELECT
    G.Name,
    GR.genreName,
    (CAST(SUBSTRING_INDEX(F.Estimated_owners,'-', 1)
    AS SIGNED) + CAST(SUBSTRING_INDEX(
    F.Estimated_owners,'-', -1) AS SIGNED))/2
    AS owners_average
FROM FACT_STREAM F
JOIN DIM_GAME G ON F.AppID = G.AppID
JOIN BRIDGE_GAME_GENRE BG ON BG.AppID= G.AppID
JOIN DIM_GENRE GR ON GR.genreID = BG.genreID
WHERE GR.genreName = 'Action'
ORDER BY owners_average DESC, G.Name
LIMIT 100;
```



Name	genreName	owners_average
Dota 2	Action	150000000.0000
Black Myth: Wukong	Action	75000000.0000
Counter-Strike: Global Offensive	Action	75000000.0000
New World	Action	75000000.0000
PUBG: BATTLEGROUNDS	Action	75000000.0000
Team Fortress 2	Action	75000000.0000
Apex Legends™	Action	35000000.0000
Brawlhalla	Action	35000000.0000
Destiny 2	Action	35000000.0000
ELDEN RING	Action	35000000.0000
Grand Theft Auto V	Action	35000000.0000
Half-Life 2: Lost Coast	Action	35000000.0000
Halo Infinite	Action	35000000.0000
Left 4 Dead 2	Action	35000000.0000
Lost Ark	Action	35000000.0000

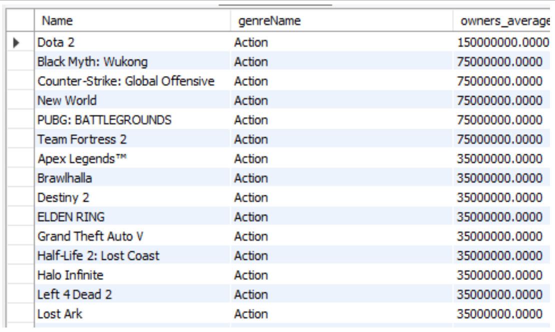
Figure 16. Results snippet for unoptimized SQL Script for Query 4.2.4

The creation of the indexes on columns on DIM_GENRE(genreName), DIM_GAME(AppID), and FACT_STREAM(AppID) allows the database engine to quick lookup and locate rows in involved tables, greatly reducing the data retrieval time. [3] With this additional optimization technique on top of the advanced SQL constructs, this script becomes more efficient compared to the former script. The breakdown of the query test result in seconds are indicated in Table 8 and 9 for reference.

Listing 8. Optimized SQL Script for Query 4.2.4

```
CREATE INDEX idx_genre ON DIM_GENRE(genreName);
CREATE INDEX idx_game ON DIM_GAME(AppID);
CREATE INDEX idx_fact_stream ON FACT_STREAM(AppID);

SELECT
    G.Name,
    GR.genreName,
    (CAST(SUBSTRING_INDEX(F.Estimated_owners,'-', 1)
    AS SIGNED) + CAST(SUBSTRING_INDEX(
    F.Estimated_owners,'-', -1) AS SIGNED))/2
    AS owners_average
FROM FACT_STREAM F
JOIN DIM_GAME G ON F.AppID = G.AppID
JOIN BRIDGE_GAME_GENRE BG ON BG.AppID= G.AppID
JOIN DIM_GENRE GR ON GR.genreID = BG.genreID
WHERE GR.genreName = 'Action'
ORDER BY owners_average DESC, G.Name
LIMIT 100;
```



Name	genreName	owners_average
Dota 2	Action	150000000.0000
Black Myth: Wukong	Action	75000000.0000
Counter-Strike: Global Offensive	Action	75000000.0000
New World	Action	75000000.0000
PUBG: BATTLEGROUNDS	Action	75000000.0000
Team Fortress 2	Action	75000000.0000
Apex Legends™	Action	35000000.0000
Brawlhalla	Action	35000000.0000
Destiny 2	Action	35000000.0000
ELDEN RING	Action	35000000.0000
Grand Theft Auto V	Action	35000000.0000
Half-Life 2: Lost Coast	Action	35000000.0000
Halo Infinite	Action	35000000.0000
Left 4 Dead 2	Action	35000000.0000
Lost Ark	Action	35000000.0000

Figure 17. Results snippet for optimized SQL Script for Query 5.2.4

5. Results and Analysis

This section showcases the results of both functional and performance testing for the ETL Pipelines using `AI Studio (RapidMiner)`, OLAP queries using `SQL` constructs, as well as the dashboard application developed using `Tableau`.

5.1 Functional Testing

This section presents the validation procedure implemented on both ETL Script and OLAP queries, ensuring that gathered data works as expected and is consistent.

5.1.1 ETL Pipeline

To verify the validity of the resulting data sources from each ELT pipeline, `process logs` generated from `AI Studio` (RapidMiner) were assessed in checking the completion of the process execution for both pipelines. See the Appendix Section II for the process log references for ETL on source database and the data warehouse. The total number of resulting rows per table on the data warehouse are as follows:

- for `DIM_GAME` and `FACT_STREAM` tables, it returned 97, 268 rows;
- for `DIM_GENRE` table, it returned 38 rows;
- for `DIM_CATEGORY`, it returned 49 rows;
- for `DIM_DEVELOPER`, it returned 60, 431 rows;
- for `DIM_PUBLISHER` table, it returned 49, 837 rows;
- for `BRIDGE_GAME_CATEGORY` table, it returned 297, 841 rows;
- for `BRIDGE_GAME_PUBLISHER` table, it returned 95, 376 rows;
- for the `BRIDGE_GAME_DEVELOPER` table, it returned 100, 635 rows;
- for `BRIDGE_GAME_GENRE` table, it returned 265, 854 rows.

5.1.2 OLAP Queries

The functional testing for each of the four queries was facilitated by executing the SQL queries in MySQL Workbench and the OLAP application, and comparing the number of results returned by each. Listed below are the different test cases for each query consisting of the expected results corresponding to SQL queries, the actual results from the OLAP application, and status of Pass or Fail for each test case.

Table 4. Test Results for OLAP Query 4.2.1

Test Cases	Expected	Actual	Pass/Fail
No games in given year	Returns 0 rows	Returns 0 rows	PASS
Valid Year Query (year = 2020)	Return monthly data for all games released in 2020, with appropriate counts and averages.	Return monthly data for all games released in 2020, with appropriate counts and averages.	PASS
Year with No Releases from October to December (year = 2024)	No games and counts from October to December.	No games and counts from October to December.	PASS

Table 5. Test Results for OLAP Query 4.2.2

Test Cases	Expected	Actual	Pass/Fail
Games with a Metacritic score > 50 in the Indie genre, and with estimated owners > 50000 - 100000	327	327	PASS
Games with a Metacritic score > 40 in the Casual genre, and with estimated owners > 20000 - 50000	79	79	PASS
Games with a Metacritic score > 80 in the Action genre, and with estimated owners > 50000 - 100000	27	27	PASS

Table 6. Test Results for OLAP Query 4.2.3

Test Cases	Expected	Actual	Pass/Fail
Developers with Game/s of Genre Indie that have an amount of at least 50,000 positive reviews	63	63	PASS
Developers with Game/s of Genre Strategy that have an amount of at least 40,000 positive reviews	31	31	PASS
Developers with Game/s of Genre Casual that have an amount of at least	71	71	PASS

15,000 positive reviews

Table 7. Test Results for OLAP Query 4.2.4

Test Cases	Expected	Actual	Pass/Fail
Top 100 Games in `Action` genre based on Popularity	100	100	PASS
Top 100 Games in `Sports` genre based on Popularity	100	100	PASS
Top 100 Games in `Adventure` genre based on Popularity	100	100	PASS

`Tables 4, 5, 6,` and `7` list the test results of evaluating the expected output using `MySQL` and the actual results from the `OLAP` application. All tables validate that each script has passed the three test cases. The test cases for `Table 1` revolve around the different specified positive review count. For `Table 2`, test cases vary in different metacritic score, genre, and estimated ownership. For `Table 3,` it differs in specified genre, and number of positive review counts. Lastly, for `Table 4`, it varies in specified genre. These test cases were tested through modification of `WHERE` and `HAVING` clauses. These functional tests were performed by everyone in the group to verify that there are no inconsistencies in the resulting outcomes per test case per query.

5.2 Performance Testing

After completing the functional testing for the four queries, the goal was to evaluate the performance of each SQL script on two different hardware configurations by executing said script in MySQL Workbench using a:

- a MacBook Air M1,
  - 8-core CPU (4 performance and 4 efficiency cores)
  - 8.00 GB RAM
- a Windows 10 Pro Desktop
  - Intel Core i5-10400 CPU 2.90GHz
  - 8.00 GB RAM

The evaluation considered the average execution times in determining the effect of the optimization of each queries. The evaluation follows a consistent approach such that both unoptimized and optimized versions were executed on each system. Further, each query had multiple runs, at least 5, performed to average out any variation in the resulting execution time that may result in systems' load or caching effects. All queries were tested using exact data sources on the data warehouse.

The input data is composed of a fact, a dimension, and many sub dimension tables with ranges from `38` rows (for `DIM_GENRE`) up to `97, 268` rows (for `DIM_GAME` and `FACT_STREAM`). Each query produces a consistent result ranging from a hundred to a few thousand rows depending on the filters applied. The returned number of attributes varied from `two` to `five` depending on the query, with Query `4.2.2` having the largest number of attributes returned.

Table 8. Average Query Performance of each script using MacBook Air M1

Query	Unoptimized SQL Script	Optimized SQL Script
1	0.058	0.057
2	0.097	0.038
3	0.698	0.311
4	0.472	0.394

Table 9. Average Query Performance of each script using Intel Core Processor

Query	Unoptimized SQL Script	Optimized SQL Script
1	0.250	0.225
2	0.244	0.166
3	3.437	0.916
4	0.284	0.259

The resulting reports indicated a substantial improvement with the optimized queries essentially for those that have subquery. Upon optimization, `Query 1` saw a slight improvement on its performance with its unoptimized version having an average fetch time of 0.250s on the Desktop and 0.058s on the Macbook and the optimized version 0.225s on the Desktop and 0.057s on the Macbook. `Query 2` saw a notable improvement from `0.097s` unoptimized to `0.038s` optimized on the MacBook, and `0.244s` to `0.166s` on the Desktop. With the most complex query among others, `Query 3` showed the largest performance gain, reducing from `3.437s` to `0.916s` optimized on Desktop, and from `0.698s` to `0.311s` in MacBook. Lastly, `Query 4` saw a modest performance improvement.

Fact and dimension tables on the schema were normalized and indexed for an efficient join. `Primary indexes` frequently used on `AppID` served as the primary key and link between tables in joins. On the other hand, `secondary indexes` are employed on repeated queries columns such as `positive_count`, `metacritic_score` and `genreName`, speeding up both the `WHERE` and the `ORDER BY` operations. For most queries, both join condition and filter condition having an index merited faster lookups instead of scanning the entire table for data matching, making the database engine quickly retrieve relevant rows using indexes employed. As discussed above, the optimized queries were proven to allow for a great reduction to query execution time. Without the usage of indexing, any aggregation operation employed such as `AVG` or `GROUP BY` would perform full scans over the large datasource which is a resource-intensive process.

Further, reduction in redundant calculations and computational cost of string manipulation was granted by narrowing down the data before employing any aggregate function through the use of index. The use of multiple joins, particularly those that span from fact to dimension table from `Query 5.2.2` and `5.2.3`, saw an improvement after optimization by the number of rows processed on each join was reduced due to indexing the join columns. However, while the use of index aids in reduction of the execution time, it is not always the case when it is applied in a column with a large volume of data which will require an additional storage resulting in increased database size. The more indexes a script has, the more time is necessary in maintaining some operation such as row insertion and deletion. The rule of thumb is to only apply it on primary key, secondary keys which are columns that are frequently used in JOINs.

6. Conclusion

In this project, AI Studio (RapidMiner) was utilized to facilitate ETL (Extract, Transform, Load) processes, allowing for the loading of data from various sources, specifically CSV files, into a data warehouse built in MySQL Workbench. The ability of a data warehouse to store and arrange huge volumes of data, enabling effective data retrieval and analysis, makes its development and maintenance important in analysis. Combined with this, ETL plays an important role in being able to load huge amounts of data, transforming them to align with the structure and requirements of the data warehouse. Using OLAP operations such as dice, slice, and roll-up, aside from OLTP operations, allows for the detailed reporting and analysis of data from the data warehouse, providing essential information that may affect business decisions made by those analyzing said data.

Creating custom indexes is necessary to create queries with faster access times than queries that use only automatically generated indexes. Query optimization is essential for enhancing performance and ensuring that analytics are generated swiftly and efficiently with strategies such as indexing and normalization. The application of these two strategies demonstrated significantly faster querying times compared to querying without them. This enables efficient, comprehensive analytics for stakeholders in the gaming industry, helping them identify which games have the highest positive reviews, Metacritic scores, or estimated owners. By demonstrating the convenience of using data warehouses like AI Studio, revealing the intuitivity of data visualization with Tableau Public, and reporting the amount of time saved by employing optimization strategies, this report may encourage businesses to become more well-informed and strategic in their decision-making through better analysis of the data they currently have. Since this report also demonstrates the relationships between users such as data analysts and businessmen and software such as data warehouses, databases, and data visualization tools, developers may be better informed on how to improve user experience by considering these relationships during development.

7. References

[1] Sheldon, R. (2023). snowflaking (snowflake schema). Data Management; TechTarget. https://www.techtarget.com/searchdatamanagement/definition/snowflaking

[2] Felix, P. B. (2017, February 14). Bridge Tables. LeapFrogBI. https://www.leapfrogbi.com/blog/bridge-tables/

[3] MySQL :: MySQL 8.4 Reference Manual :: 15.1.15 CREATE INDEX Statement. (2019). Mysql.com. https://dev.mysql.com/doc /refman/8.4/en/create-index.html

[4] MySQL Connector - Tableau. (2024). Tableau.com. https://help.tableau.com/current/pro/desktop/en-us/examples_mysql.htm

[5] *MySQL 8.4 Reference Manual :: 10.3.5 column indexes*. MySQL 8.4 Reference Manual. (n.d.). https://dev.mysql.com/doc/refman/8.4/en/column-indexes.html

[6] *What is a Data Warehouse?* (n.d.). https://www.oracle.com/ph/database/what-is-a-data-warehouse/

8. Declarations

8.1 Declaration of Generative AI in Scientific Writing.

During the preparation of this work the author(s) used ChatGPT in order to refine explanations, and assist with drafting content for this report. After using this tool/service, the author(s) reviewed and edited the content as needed and take(s) full responsibility for the content of the publication.

8.2 Record of Contribution.

P1: Clarissa Albarracin	**P2:** Amor De Guzman
P3: Reina Althea Garcia	**P4:** Miko Santos

Activity	**P1**	**P2**	**P3**	**P4**
A. Introduction	95.00	5.00		
B. Data Warehouse	10.00	10.00	70.00	10.00
C. ETL Script		10.00	90.00	
D. OLAP Application	25.00		25.00	50.00
E. Query Processing and Optimization	25.00		25.00	50.00
F. Results and Analysis	10.00	10.00		80.00
G. Conclusion		40.00	60.00	
Raw Total	165.00	75.00	270.00	190.00
TOTAL	23.57	10.71	38.57	28.57

A. APPENDIX

SECTION I

Table. Query Test Results (in seconds) with **Unoptimized SQL Script** on the Original Schema [**`MacBook Air M1`**]

Query	T1	T2	T3	T4	T5	Avg
1	0.060	0.063	0.062	0.055	0.052	**0.058**
2	0.110	0.109	0.062	0.079	0.125	**0.097**
3	0.754	0.689	0.697	0.657	0.692	**0.698**
4	0.641	0.437	0.422	0.421	0.437	**0.472**

Table. Query Test Results (in seconds) with **Optimized SQL Script** on the Original Schema [`MacBook Air M1`]

Query	T1	T2	T3	T4	T5	Avg
1	0.058	0.057	0.056	0.053	0.059	**0.057**
2	0.047	0.032	0.047	0.031	0.031	**0.038**
3	0.313	0.332	0.320	0.297	0.295	**0.311**
4	0.375	0.391	0.407	0.406	0.391	**0.394**

Table. Query Test Results (in seconds) with **Unoptimized SQL Script** on the Original Schema [`Intel Core`]

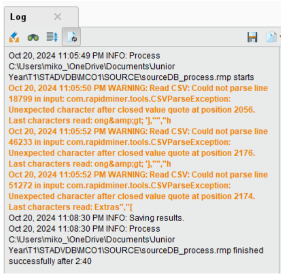
Query	T1	T2	T3	T4	T5	Avg
1	0.234	0.265	0.296	0.219	0.234	**0.250**
2	0.296	0.235	0.234	0.235	0.219	**0.244**
3	3.687	3.328	3.422	3.359	3.390	**3.437**
4	0.328	0.282	0.265	0.281	0.266	**0.284**

Table. Query Test Results (in seconds) with **Optimized SQL Script** on the Original Schema [`Intel Core`]

Query	T1	T2	T3	T4	T5	Avg
1	0.188	0.265	0.203	0.218	0.250	**0.225**
2	0.172	0.157	0.172	0.156	0.171	**0.166**
3	0.953	0.937	0.891	0.890	0.907	**0.916**
4	0.265	0.250	0.266	0.265	0.250	**0.259**

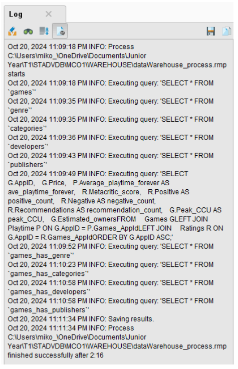
SECTION II

Process Log 1: (sourceDB)



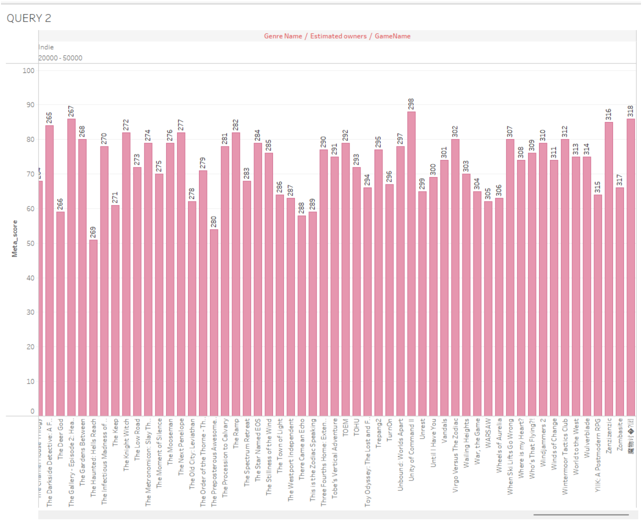
```
Oct 20, 2024 11:05:49 PM INFO: Process
C:\Users\miko_\OneDrive\Documents\Junior
Year\T1\STADVDB\MCO1\SOURCE\sourceDB_process.rmp starts
Oct 20, 2024 11:05:50 PM WARNING: Read CSV: Could not parse line
18799 in input: com.rapidminer.tools.CSVParseException:
Unexpected character after closed value quote at position 2056.
Last characters read: ong&amp;gt; '],"","h
Oct 20, 2024 11:05:52 PM WARNING: Read CSV: Could not parse line
46233 in input: com.rapidminer.tools.CSVParseException:
Unexpected character after closed value quote at position 2176.
Last characters read: ong&amp;gt; '],"","h
Oct 20, 2024 11:05:52 PM WARNING: Read CSV: Could not parse line
51272 in input: com.rapidminer.tools.CSVParseException:
Unexpected character after closed value quote at position 2174.
Last characters read: Extras","[
Oct 20, 2024 11:08:30 PM INFO: Saving results.
Oct 20, 2024 11:08:30 PM INFO: Process
C:\Users\miko_\OneDrive\Documents\Junior
Year\T1\STADVDB\MCO1\SOURCE\sourceDB_process.rmp finished
successfully after 2:40
```

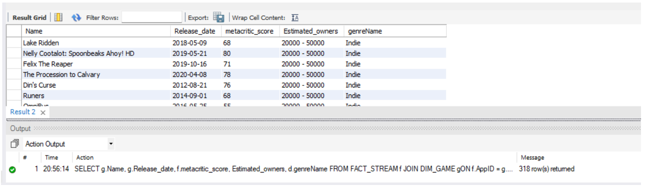
Process Log 2: (dataWarehouse)



```
Oct 20, 2024 11:09:18 PM INFO: Process
C:\Users\miko_\OneDrive\Documents\Junior
Year\T1\STADVDB\MCO1\WAREHOUSE\dataWarehouse_process.rmp
starts
Oct 20, 2024 11:09:18 PM INFO: Executing query: 'SELECT * FROM
`games`'
Oct 20, 2024 11:09:35 PM INFO: Executing query: 'SELECT * FROM
`genre`'
Oct 20, 2024 11:09:35 PM INFO: Executing query: 'SELECT * FROM
`categories`'
Oct 20, 2024 11:09:36 PM INFO: Executing query: 'SELECT * FROM
`developers`'
Oct 20, 2024 11:09:43 PM INFO: Executing query: 'SELECT * FROM
`publishers`'
Oct 20, 2024 11:09:49 PM INFO: Executing query: 'SELECT
G.AppID,    G.Price,    P.Average_playtime_forever AS
ave_playtime_forever,    R.Metacritic_score,    R.Positive AS
positive_count,    R.Negative AS negative_count,
R.Recommendations AS recommendation_count,    G.Peak_CCU AS
peak_CCU,    G.Estimated_ownersFROM    Games GLEFT JOIN
Playtime P ON G.AppID = P.Games_AppIdLEFT JOIN    Ratings R ON
G.AppID = R.Games_AppIdORDER BY G.AppID ASC;'
Oct 20, 2024 11:09:52 PM INFO: Executing query: 'SELECT * FROM
`games_has_genre`'
Oct 20, 2024 11:10:23 PM INFO: Executing query: 'SELECT * FROM
`games_has_categories`'
Oct 20, 2024 11:10:58 PM INFO: Executing query: 'SELECT * FROM
`games_has_developers`'
Oct 20, 2024 11:10:58 PM INFO: Executing query: 'SELECT * FROM
`games_has_publishers`'
Oct 20, 2024 11:11:34 PM INFO: Saving results.
Oct 20, 2024 11:11:34 PM INFO: Process
C:\Users\miko_\OneDrive\Documents\Junior
Year\T1\STADVDB\MCO1\WAREHOUSE\dataWarehouse_process.rmp
finished successfully after 2:16
```

The images provided are examples from Query 2, demonstrating that both the SQL query and Tableau returned 318 rows, confirming they produce the same results.





Name	Release_date	metacritic_score	Estimated_owners	genreName
Lake Ridden	2018-05-09	68	20000 - 50000	Indie
Nelly Cootalot: Spoonbeaks Ahoy! HD	2019-05-21	80	20000 - 50000	Indie
Felix The Reaper	2019-10-16	71	20000 - 50000	Indie
The Procession to Calvary	2020-04-08	78	20000 - 50000	Indie
Din's Curse	2012-08-21	76	20000 - 50000	Indie
Runers	2014-09-01	68	20000 - 50000	Indie

#	Time	Action	Message
1	20:56:14	SELECT g.Name, g.Release_date, f.metacritic_score, Estimated_owners, d.genreName FROM FACT_STREAM f JOIN DIM_GAME gON f.AppID = g....	318 row(s) returned

SECTION III

OLAP Application Dashboard

https://public.tableau.com/app/profile/reina.althea.garcia/viz/MCO1-SteamGamesDatasetAnalysisV2/Dashboard1?publish=yes