Query Processing Technical Report

Marian Ricci N. Ariaga[1], Dana Louise A. Guillarte[2], Yesha P. Llorando[3], and Chrysille Grace L. So[4]
De La Salle University Manila
[1]marian_ricci_ariaga@dlsu.edu.ph, [2]dana_guillarte@dlsu.edu.ph, [3]yesha_llorando@dlsu.edu.ph, [4]chrysille_so@dlsu.edu.ph

1. Introduction

For this project, reports on game sales and performance were generated using the STEAM Games dataset, visualized through a Tableau dashboard. The data warehouse, structured with a star schema, contains key information on game performance, user engagement, and sales over time. The OLAP application supports multidimensional analysis, enabling marketing and business intelligence teams at Steam to explore trends in player behavior, genres, and sales performance.

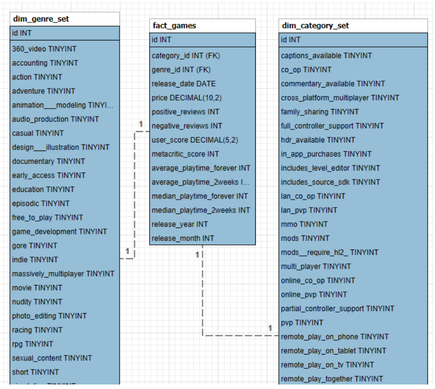
The target users are marketing and business intelligence teams, who can leverage these insights to optimize promotions and better understand player preferences. Game developers can also benefit from the reports, using them to gauge game performance as titles are released over time. This system empowers users to make data-driven decisions that enhance game strategy and drive sales growth.

2. Data Warehouse

The query processing begins with designing a schema for the Data Warehouse, intended for the OLAP (Online Analytical Processing) application. The proposed schema for the Games dataset follows a star schema design, a common and efficient approach in data warehousing due to its simplicity and performance benefits. In this design, the central fact table, fact_games, records quantitative data related to game attributes, while surrounding dimension tables provide context.

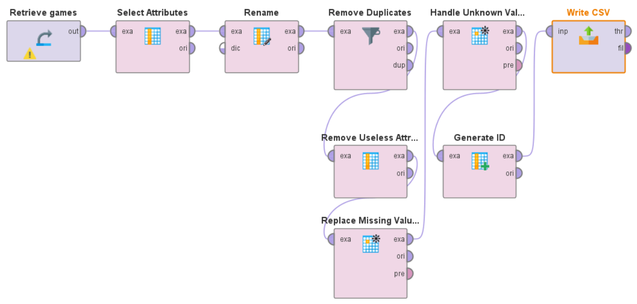
The *fact_games* table is the core component, capturing key metrics such as release date, price, reviews (positive and negative), user scores, and Metacritic scores, along with playtime statistics like average and median playtime. These fields offer a comprehensive view of each game's performance and user engagement. Foreign keys for *category_id* and *genre_id* link to the corresponding dimension tables, allowing analysis based on game genres and categories, and their impact on popularity and satisfaction.

The dimension tables, *dim_genre_set* and *dim_category_set*, represent game genres and categories, respectively, with multiple genres or categories associated with each game. The one-hot encoding format is used for these tables, converting unique values into binary columns arranged in alphabetical order for clarity. Each record in the dimension tables corresponds to a single record in *fact_games*, ensuring a strict 1:1 relationship, which adheres to the star schema's design principles and supports efficient querying and data aggregation based on game attributes.



3. ETL Script

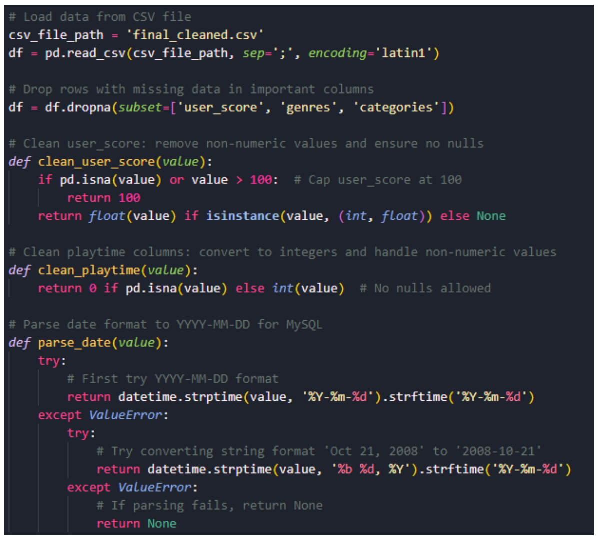
The ETL process involves using tools like Altair AI Studio (RapidMiner), and Python with libraries such as pandas and MySQL to extract, clean, transform, and load data into the OLAP application. Most of the extraction and cleaning is done in RapidMiner, as shown in the processing pipeline below.



The dataset is imported from a CSV file using RapidMiner's CSV operator for convenience. The Select Attributes operator is applied to choose relevant columns, followed by the Rename operator to align column names with the data warehouse schema. To clean the data, operators like Remove Duplicates, Remove Useless Attributes, and Replace Missing Values eliminate redundant or incomplete data. An ID is generated for each game record, and the cleaned data is exported as a CSV file.

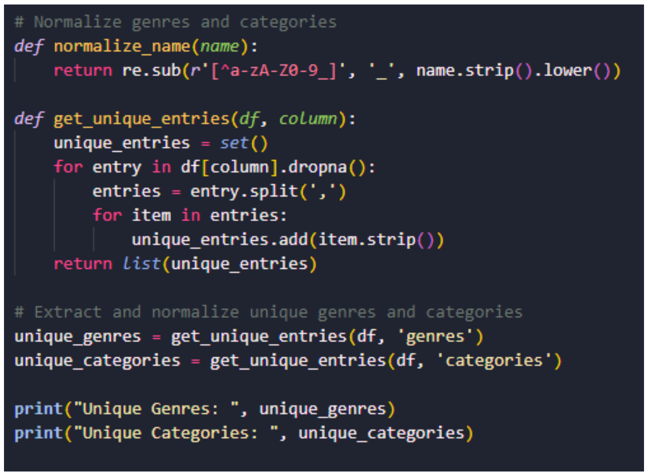
This CSV file is then processed in Python, where it is loaded into a DataFrame using pandas. The Python script further prepares the data for a MySQL database by extracting unique genres and categories, splitting them into individual attributes to form dimension tables (dim_genre_set and dim_category_set).

The functions in the image below are the additional functions we used for data pre-processing. Function parse_date converts the *release_date* from a string to a datetime variable.



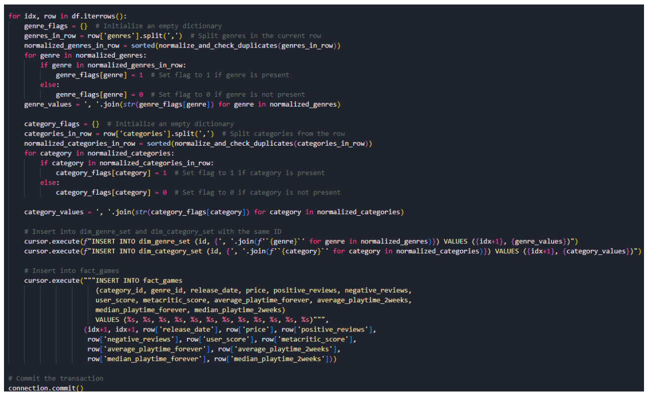
```
# Load data from CSV file
csv_file_path = 'final_cleaned.csv'
df = pd.read_csv(csv_file_path, sep=';', encoding='latin1')

# Drop rows with missing data in important columns
df = df.dropna(subset=['user_score', 'genres', 'categories'])

# Clean user_score: remove non-numeric values and ensure no nulls
def clean_user_score(value):
    if pd.isna(value) or value > 100:  # Cap user_score at 100
        return 100
    return float(value) if isinstance(value, (int, float)) else None

# Clean playtime columns: convert to integers and handle non-numeric values
def clean_playtime(value):
    return 0 if pd.isna(value) else int(value)  # No nulls allowed

# Parse date format to YYYY-MM-DD for MySQL
def parse_date(value):
    try:
        # First try YYYY-MM-DD format
        return datetime.strptime(value, '%Y-%m-%d').strftime('%Y-%m-%d')
    except ValueError:
        try:
            # Try converting string format 'Oct 21, 2008' to '2008-10-21'
            return datetime.strptime(value, '%b %d, %Y').strftime('%Y-%m-%d')
        except ValueError:
            # If parsing fails, return None
            return None
```

The functions *normalize_name()* and *get_unique_entries()* are used to transform and pivot unique entries from the genre and category fields into table columns, ensuring they follow a lowercase, underscore-separated format.



```python
# Normalize genres and categories
def normalize_name(name):
    return re.sub(r'[^a-zA-Z0-9_]', '_', name.strip().lower())

def get_unique_entries(df, column):
    unique_entries = set()
    for entry in df[column].dropna():
        entries = entry.split(',')
        for item in entries:
            unique_entries.add(item.strip())
    return list(unique_entries)

# Extract and normalize unique genres and categories
unique_genres = get_unique_entries(df, 'genres')
unique_categories = get_unique_entries(df, 'categories')

print("Unique Genres: ", unique_genres)
print("Unique Categories: ", unique_categories)
```

This loop in our code iterates through the cleaned and transformed data frames to insert each row into their respective tables.



```python
for idx, row in df.iterrows():
    genre_flags = {}  # Initialize an empty dictionary
    genres_in_row = row['genres'].split(',')  # Split genres in the current row
    normalized_genres_in_row = sorted(normalize_and_check_duplicates(genres_in_row))
    for genre in normalized_genres:
        if genre in normalized_genres_in_row:
            genre_flags[genre] = 1  # Set flag to 1 if genre is present
        else:
            genre_flags[genre] = 0  # Set flag to 0 if genre is not present
    genre_values = ', '.join(str(genre_flags[genre]) for genre in normalized_genres)

    category_flags = {}  # Initialize an empty dictionary
    categories_in_row = row['categories'].split(',')  # Split categories from the row
    normalized_categories_in_row = sorted(normalize_and_check_duplicates(categories_in_row))
    for category in normalized_categories:
        if category in normalized_categories_in_row:
            category_flags[category] = 1  # Set flag to 1 if category is present
        else:
            category_flags[category] = 0  # Set flag to 0 if category is not present

    category_values = ', '.join(str(category_flags[category]) for category in normalized_categories)

    # Insert into dim_genre_set and dim_category_set with the same ID
    cursor.execute(f"INSERT INTO dim_genre_set (id, {', '.join(f'`{genre}`' for genre in normalized_genres)}) VALUES ({idx+1}, {genre_values})")
    cursor.execute(f"INSERT INTO dim_category_set (id, {', '.join(f'`{category}`' for category in normalized_categories)}) VALUES ({idx+1}, {category_values})")

    # Insert into fact_games
    cursor.execute("""INSERT INTO fact_games
                    (category_id, genre_id, release_date, price, positive_reviews, negative_reviews,
                    user_score, metacritic_score, average_playtime_forever, average_playtime_2weeks,
                    median_playtime_forever, median_playtime_2weeks)
                    VALUES (%s, %s, %s, %s, %s, %s, %s, %s, %s, %s, %s, %s)""",
                  (idx+1, idx+1, row['release_date'], row['price'], row['positive_reviews'],
                   row['negative_reviews'], row['user_score'], row['metacritic_score'],
                   row['average_playtime_forever'], row['average_playtime_2weeks'],
                   row['median_playtime_forever'], row['median_playtime_2weeks']))

# Commit the transaction
connection.commit()
```

4. OLAP Application

The primary objective of the application is to assist decision-makers, such as game developers, marketers, and analysts, in deriving actionable insights from game-related data. This involves several key analytical tasks that are essential for understanding player behavior, market trends, and game performance. By leveraging the capabilities of OLAP (Online Analytical Processing) and visualization tools like Tableau, the application enables users to explore and interpret complex datasets with ease.

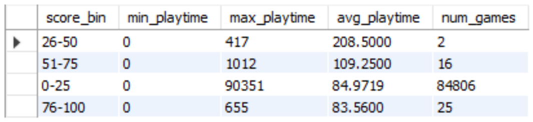
Playtime Statistics by Category and User Ratings (Roll-up and Slice)

This report provides insights into the relationship between user scores and playtime for specific game categories, aiming to identify which types of games, based on user scores, exhibit the highest player engagement through playtime metrics. By using SQL constructs like CASE, aggregate functions, joins, and grouping, the query helps developers and game designers better understand user engagement patterns.

The query calculates the minimum, maximum, and average playtime for games since their release, grouped by user score categories. User scores (0-100) are divided into four bins using a CASE clause. It also filters by game category (e.g., single-player or multiplayer) to reveal which modes attract more player engagement. These insights can guide developers in tailoring games to match user preferences, improving player retention.

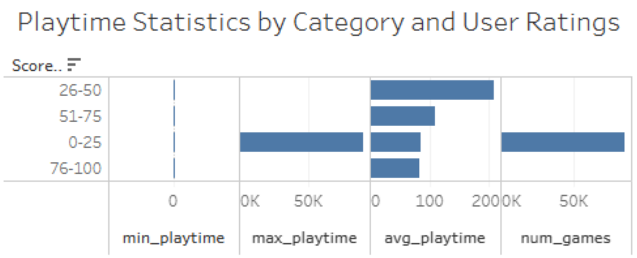
```sql
SELECT
   CASE
      WHEN f.user_score BETWEEN 0 AND 25 THEN '0-25'
      WHEN f.user_score BETWEEN 26 AND 50 THEN '26-50'
      WHEN f.user_score BETWEEN 51 AND 75 THEN '51-75'
      WHEN f.user_score BETWEEN 76 AND 100 THEN '76-100'
      ELSE 'Unknown'
   END AS score_bin,

   MIN(f.average_playtime_forever) AS min_playtime,
   MAX(f.average_playtime_forever) AS max_playtime,
   AVG(f.average_playtime_forever) AS avg_playtime,
   COUNT(f.id) AS num_games
FROM
   fact_games f
JOIN
   dim_category_set c ON f.id = c.id
WHERE
   c.single_player = 1
GROUP BY
   score_bin
ORDER BY
   avg_playtime DESC;
```

The query results in MySQL Workbench are shown below.



	score_bin	min_playtime	max_playtime	avg_playtime	num_games
▶	26-50	0	417	208.5000	2
	51-75	0	1012	109.2500	16
	0-25	0	90351	84.9719	84806
	76-100	0	655	83.5600	25

The playtime statistics are visualized in the OLAP application as such.



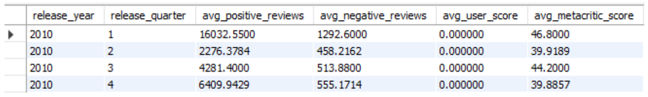
Average Game Reviews and Scores by Genre and Release Quarter (Roll-up and Dice)

This report analyzes trends in reviews and scores of games released across different quarters of the year. It helps developers and industry professionals track how games perform over time and identify patterns by genre and release dates. Through SQL time functions, joins, grouping, and aggregation, this report delivers valuable insights into user satisfaction and critical reception across various periods.

The query reports average positive and negative reviews, user scores, and Metacritic scores, grouped by release year and quarter. It also filters by genre and release year. This information enables analysts to understand which factors contribute most to user experience, helping them improve future game releases based on historical trends.

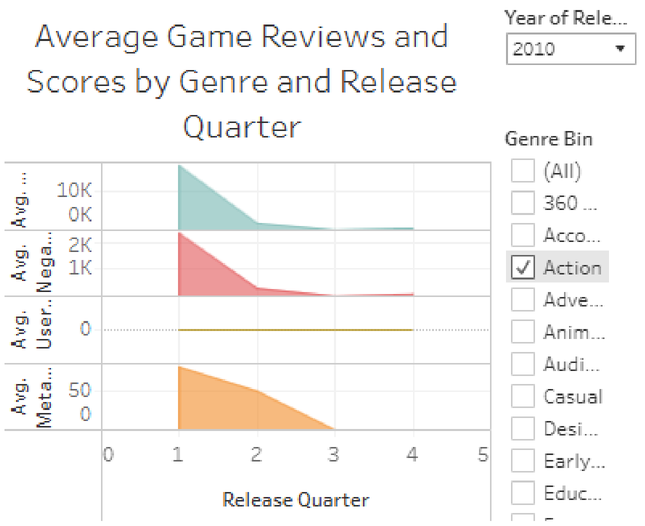
```
SELECT
   release_year,
   QUARTER(f.release_date) AS release_quarter,
   AVG(f.positive_reviews) AS avg_positive_reviews,
   AVG(f.negative_reviews) AS avg_negative_reviews,
   AVG(f.user_score) AS avg_user_score,
   AVG(f.metacritic_score) AS avg_metacritic_score
FROM
   fact_games f
JOIN
   dim_genre_set d ON f.genre_id = d.id
WHERE
   d.action = 1
GROUP BY
   release_year, release_quarter
HAVING
   release_year = 2010
ORDER BY
   release_year;
```

The query results in MySQL Workbench are shown below.



release_year	release_quarter	avg_positive_reviews	avg_negative_reviews	avg_user_score	avg_metacritic_score
2010	1	16032.5500	1292.6000	0.000000	46.8000
2010	2	2276.3784	458.2162	0.000000	39.9189
2010	3	4281.4000	513.8800	0.000000	44.2000
2010	4	6409.9429	555.1714	0.000000	39.8857

The average game reviews and scores are visualized in the OLAP application as such.



Game Release Trends (Roll-up, Drill-down, Slice)

The Game Release Trends report provides insights into how game releases fluctuate over time, helping decision-makers identify seasonal trends, peak release periods, or shifts in platform popularity. It uses time-based grouping, aggregation, and sorting to track industry growth and detect patterns or anomalies in game launches.

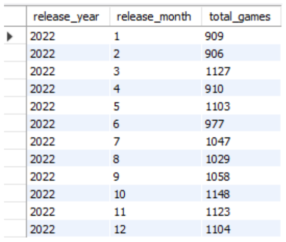
The report shows the number of games released per year by aggregating release counts through GROUP BY.

```
SELECT
   release_year,
   COUNT(*) AS total_games
   FROM fact_games f
GROUP BY
   release_year
   ORDER BY release_year;
```

The report's granularity can be increased by drilling-down reflecting monthly trends within a specific year.

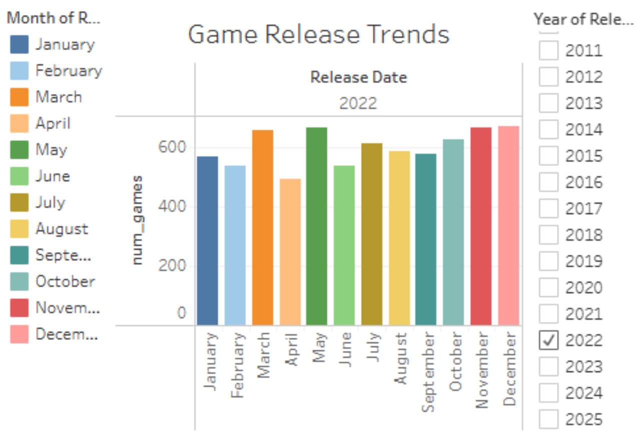
```
SELECT
   release_year,
   release_month,
   COUNT(*) AS total_games
FROM
   fact_games f
WHERE
   YEAR(f.release_date) = 2022
GROUP BY
   release_year, release_month
ORDER BY
   release_year, release_month;
```

The query results in MySQL Workbench are shown below.



release_year	release_month	total_games
2022	1	909
2022	2	906
2022	3	1127
2022	4	910
2022	5	1103
2022	6	977
2022	7	1047
2022	8	1029
2022	9	1058
2022	10	1148
2022	11	1123
2022	12	1104

The game release trends are visualized in the OLAP application as such.



Positive Reviews Rate by Genre and Release Date (Roll-up, Drill-down, Slice, Dice)

The Positive Reviews Rate by Genre report assesses the impact of user reviews within specific game genres. It helps decision-makers identify successful categories based on time-based releases, providing valuable insights for marketing teams promoting highly-rated games and development teams aiming to replicate the success of well-reviewed titles.

This query employs time-based grouping, aggregation, joins, and NULL handling to analyze the yearly positive review rate for indie games. The insights enable developers and analysts to track user sentiment trends over the years and evaluate whether user satisfaction has improved or declined.

The positive review rate is calculated as follows:

$$Positive\ Review\ Rate\ = \frac{positive\ reviews}{(positive\ reviews + negative\ reviews)}$$

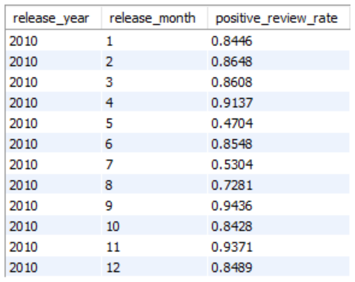
This value is rolled up by release year and filtered by the selected genre.

```
SELECT
   release_year,
   SUM(f.positive_reviews) / (SUM(f.positive_reviews) +
SUM(f.negative_reviews), 0) AS positive_review_rate
FROM
   fact_games f
JOIN
   dim_genre_set c ON f.genre_id = c.id
WHERE
   c.adventure = 1
GROUP BY
   release_year
ORDER BY
   release_year DESC;
```

For greater granularity, the report can be drilled down to a monthly aggregation, allowing further analysis. Dicing the report by genre and year leads to a more focused report.

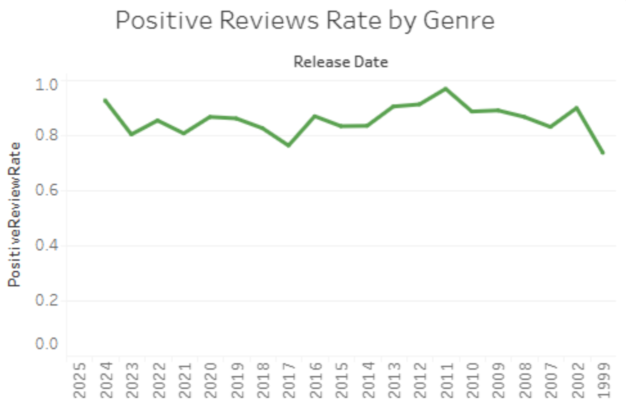
```
SELECT
   release_year,
   release_month,
   SUM(f.positive_reviews) / NULLIF(SUM(f.positive_reviews) +
SUM(f.negative_reviews), 0) AS positive_review_rate
FROM
   fact_games f
JOIN
   dim_genre_set c ON f.genre_id = c.id
WHERE
   c.adventure = 1
GROUP BY
   release_year, release_month
HAVING
   release_year = 2010
ORDER BY
   release_year DESC;
```

The query results in MySQL Workbench are shown below.



release_year	release_month	positive_review_rate
2010	1	0.8446
2010	2	0.8648
2010	3	0.8608
2010	4	0.9137
2010	5	0.4704
2010	6	0.8548
2010	7	0.5304
2010	8	0.7281
2010	9	0.9436
2010	10	0.8428
2010	11	0.9371
2010	12	0.8489

The report for positive reviews rate by genre and release date is visualized in the OLAP application as such.



The use of Tableau in these reports enhances the user experience by providing dynamic, interactive visualizations. Tableau allows decision-makers to easily slice and dice data, filter out specific variables, and roll up or drill down to different levels of detail. This interactive functionality not only simplifies data exploration but also ensures that insights are immediately actionable, allowing users to make informed decisions based on real-time data.

5. Query Processing and Optimization

Query optimization is the process of enhancing the efficiency of a database query by minimizing execution time and resource consumption. It involves selecting the best query option for executing a query, considering factors such as indexing, join costs, and data types. Query optimization is essential to ensure that queries remain efficient as the size of the database increases with time. Doing so can allow developers to easily incorporate the database to their projects, without concerning themselves over long retrieval times.

A. Optimal Database Design

To prioritize query optimization, the data warehouse schema was made to be as **denormalized** as possible, while not sacrificing the features available. The star schema is naturally denormalized, compared to the constellation schema, leading to less JOINs.

The dimension tables follow a one-hot encoding scheme to efficiently cater to a game having more than a single genre and category. This approach stores genre and category information as separate binary columns, with each column representing a specific genre or category, and a value of 1 indicating its presence. Using the **TINYINT** data type for these columns optimizes query performance by reducing storage space and memory usage, resulting in faster read and write operations. It also simplifies filtering and aggregation, allowing queries to quickly check for specific genres or categories without relying on complex joins or scanning large text-based fields.

Furthermore, the dimension tables (i.e., dim_category_set and dim_genre_set) have a **1:1 relationship** with the fact table (fact_games), adhering to the star schema design principle. This structure **minimizes the need for additional joins,** improving query performance by keeping the relationships between tables simple and efficient. Since games often belong to a 'set' of categories and genres, representing this information through one-hot encoding within dimension tables is a natural fit. It allows for more direct analysis of these sets, making the schema not only optimized for performance but also logically aligned with the data model.

Additional columns for release_year and release_month were added, as they are frequently used for aggregations in report queries. The group chose physical columns over generated virtual columns to reduce the need for re-computation.

B. Query Restructuring

To ensure the optimal performance of the final query, multiple versions of the same query were written and tested side by side. Additionally, only the necessary columns were selected before performing joins to reduce computational costs.

The different test versions of the queries from each of the OLAP reports are described below, with the full queries available in the appendix section of the paper.

Playtime Statistics by Category and User Ratings

#	Test case	Description
1	Using the WITH clause	Using the WITH clause to pre-select filtered columns in the specified category.
2	Using INNER JOIN with Subquery	Applying INNER JOIN on fact_games and a subquery (selecting filtered records for a specified category).
3	Using INNER JOIN on fact and dim table	Applying inner join between fact_games and dim_category_set on id.
4	Using WHERE clause to link 2 tables	Joining fact_games and dim_category_set indirectly using the WHERE clause ON table ids.

Average Game Reviews and Scores by Genre and Release Quarter

#	Test case	Description
1	Using INNER JOIN with alias in GROUP BY clause	Applying inner join between fact_games and dim_genre_set on id. Alias of release_year is used in the GROUP BY clause.
2	Using WHERE clause to link 2 tables	Joining fact_games and dim_genre_set indirectly using the WHERE clause ON table ids.
3	Using INNER JOIN with function in GROUP BY clause	Applying inner join between fact_games and dim_genre_set on id. YEAR(release_date) is used in the GROUP BY clause.

Game Release Trends

#	Test case	Description
1	YEAR() and MONTH() for release_year and release_month	Using year and month functions to extract release_year and release_month in attribute selection.
2	Using SUBSTRING() for release_year and release_month	Using SUBSTRING(DATE_FORMAT(f.release_date, '%Y-%m-%d'), 1, 4) to extract release_year and release_month in attribute selection.
3	Using WITH clause to define release_years	Using the WITH clause to pre-select release_year and release_month while filtering the selected year.

Positive Reviews Rate by Genre and Release Date

#	Test case	Description
1	Using INNER JOIN and GROUP BY alias	Applying inner join between fact_games and dim_genre_set on id. Alias of release_year is used in the GROUP BY clause.
2	Using INNER JOIN with function in GROUP BY clause	Applying inner join between fact_games and dim_genre_set on id. YEAR(release_date) is used in the GROUP BY clause.
3	Using WHERE to link two tables	Joining fact_games and dim_genre_set indirectly using the WHERE clause ON table ids.

C. Indexing

To optimize query performance and avoid full table scans, clustered indexes were implemented using auto-increment primary keys for all tables. In addition, several non-clustered indexes were added to further enhance performance:

```
CREATE INDEX idx_games_user_playtime_id
ON fact_games (user_score, average_playtime_forever, id);
CREATE INDEX idx_games_release_year_month
ON fact_games (release_year, release_month);
```

For genre-specific columns, individual indexes were generated to speed up slice operations. For example:

```
CREATE INDEX idx_genre_indie
ON dim_genre_set (indie);

CREATE INDEX idx_genre_action
ON dim_genre_set (action);
```

Similarly, category columns were indexed to improve filtering. For instance:

```
CREATE INDEX idx_category_single_player
ON dim_category_set (single_player);

CREATE INDEX idx_category_co_op
ON dim_category_set (co_op);
```

While this indexing strategy improves query performance, it introduces a trade-off due to the large number of indexes required by the dimension tables. Each index consumes space, with individual index values from either dim_genre_set or dim_category_set occupying 1 byte because of the TINYINT data type. With approximately 90,000 rows, this storage consumption becomes significant.

The group considered two approaches: restructuring the schema to store one record per genre for each game or maintaining the existing one-hot encoding format with indexes for every column. After evaluating the trade-offs, the group concluded that having multiple records for genres would be more expensive, as it would require using VARCHAR to store genre names, leading to redundant records across games. In contrast, the one-hot encoding approach ensures faster filtering and minimizes redundancy, making it the preferred solution.

6. Results and Analysis

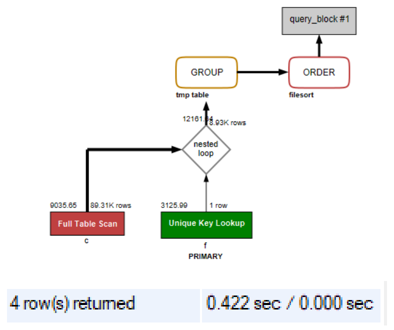
This section shows the results and analysis of the following tests: query execution plan comparison, query performance testing and evaluation, and data insert validation.

A. Query Execution Plan Comparisons

This section showcases the execution plans of each report query pre-optimization and post-optimization techniques.

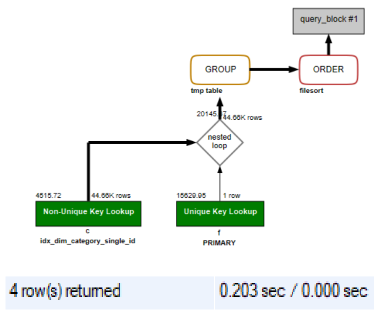
Playtime Statistics by Category and User Ratings

The images below illustrate the original execution plan for the playtime statistics query, along with its actual execution time. Initially, a full table scan is performed due to the use of roll-ups and aggregation functions.



4 row(s) returned 0.422 sec / 0.000 sec

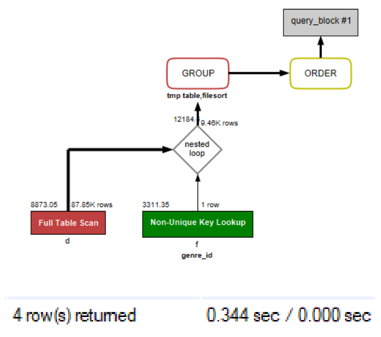
After applying the idx_games_release_year_month index and the relevant indexes for columns in dim_category_set, the query avoids a full table scan. This improved query also leverages the physical release_year and release_month columns instead of relying on the YEAR() and MONTH() functions. A comparison of the two query executions shows a noticeable increase in speed.



4 row(s) returned 0.203 sec / 0.000 sec

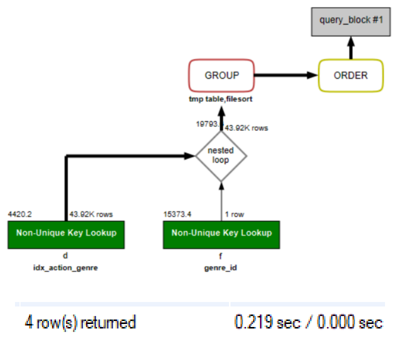
Average Game Reviews and Scores by Genre and Release Quarter

The images below illustrate the original execution plan for the average game reviews and scores query, along with its actual execution time. As with the previous case, a full table scan is used because of roll-ups and aggregation functions.



4 row(s) returned 0.344 sec / 0.000 sec

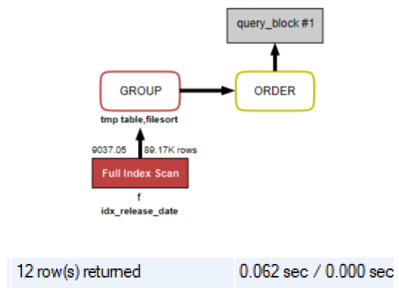
By utilizing the idx_games_release_year_month index and the corresponding indexes on columns in dim_genre_set, the query avoids performing a full table scan. A comparison of the two query executions demonstrates a visible improvement in speed.



4 row(s) returned 0.219 sec / 0.000 sec

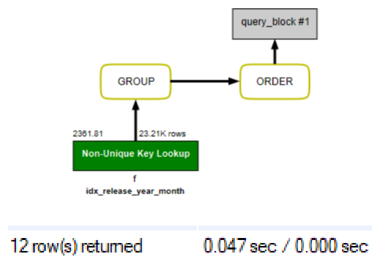
Game Release Trends

The diagram below presents the execution plan for analyzing game release trends. The initial query involves a full table scan due to the use of roll-ups and aggregation functions.



12 row(s) returned 0.062 sec / 0.000 sec

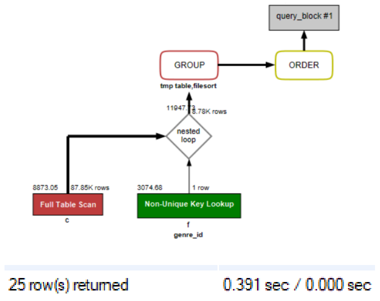
After using idx_games_release_year_month index and direct references to the physical release_year and release_month columns, the query becomes significantly more optimized in terms of both cost and execution time.



12 row(s) returned 0.047 sec / 0.000 sec

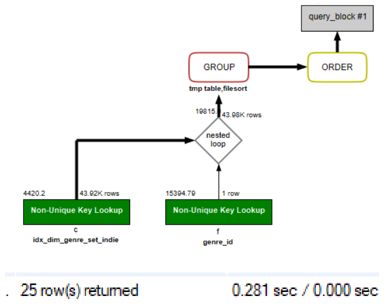
Positive Reviews Rate by Genre and Release Date

The diagram below shows the execution plan for analyzing positive review rates by genre and release date. Similar to the other queries, a full table scan is required due to the aggregation functions and roll-ups.



25 row(s) returned 0.391 sec / 0.000 sec

After incorporating the idx_games_release_year_month index and the relevant indexes for dim_genre_set columns, the query no longer performs a full table scan. A comparison of the two query executions reveals a noticeable increase in speed.



25 row(s) returned 0.281 sec / 0.000 sec

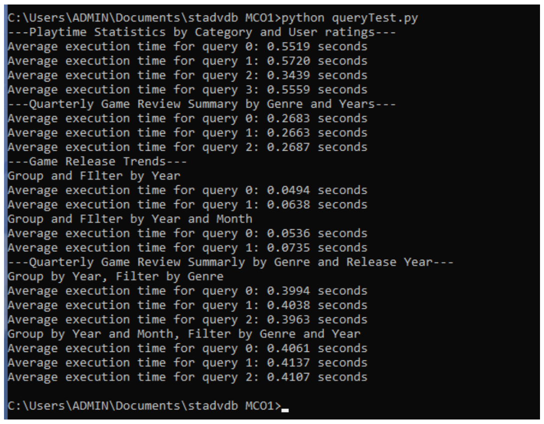
B. Query Performance Testing

This section shows the results of the automated query performance tests.

A test tool was developed in Python to benchmark the execution times of restructured test queries, as outlined in the previous section. Each test query was executed 30 times, and the average execution time from all runs was displayed in the command line.

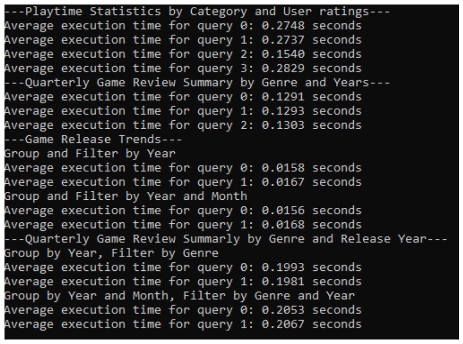
The tests were conducted in two phases: pre-optimization and post-optimization.

The image below shows the benchmarked execution times for each restructured query across the four different reports:



```
C:\Users\ADMIN\Documents\stadvdb MCO1>python queryTest.py
---Playtime Statistics by Category and User ratings---
Average execution time for query 0: 0.5519 seconds
Average execution time for query 1: 0.5720 seconds
Average execution time for query 2: 0.3439 seconds
Average execution time for query 3: 0.5559 seconds
---Quarterly Game Review Summary by Genre and Years---
Average execution time for query 0: 0.2683 seconds
Average execution time for query 1: 0.2663 seconds
Average execution time for query 2: 0.2687 seconds
---Game Release Trends---
Group and FIlter by Year
Average execution time for query 0: 0.0494 seconds
Average execution time for query 1: 0.0638 seconds
Group and FIlter by Year and Month
Average execution time for query 0: 0.0536 seconds
Average execution time for query 1: 0.0735 seconds
---Quarterly Game Review Summarly by Genre and Release Year---
Group by Year, Filter by Genre
Average execution time for query 0: 0.3994 seconds
Average execution time for query 1: 0.4038 seconds
Average execution time for query 2: 0.3963 seconds
Group by Year and Month, Filter by Genre and Year
Average execution time for query 0: 0.4061 seconds
Average execution time for query 1: 0.4137 seconds
Average execution time for query 2: 0.4107 seconds

C:\Users\ADMIN\Documents\stadvdb MCO1>
```

The following image presents the recorded execution times after applying query optimization techniques, where a significant improvement in performance can be observed. Functions were removed from GROUP BY clauses, and similar queries were merged prior to running the test.



```
---Playtime Statistics by Category and User ratings---
Average execution time for query 0: 0.2748 seconds
Average execution time for query 1: 0.2737 seconds
Average execution time for query 2: 0.1540 seconds
Average execution time for query 3: 0.2829 seconds
---Quarterly Game Review Summary by Genre and Years---
Average execution time for query 0: 0.1291 seconds
Average execution time for query 1: 0.1293 seconds
Average execution time for query 2: 0.1303 seconds
---Game Release Trends---
Group and Filter by Year
Average execution time for query 0: 0.0158 seconds
Average execution time for query 1: 0.0167 seconds
Group and Filter by Year and Month
Average execution time for query 0: 0.0156 seconds
Average execution time for query 1: 0.0168 seconds
---Quarterly Game Review Summarly by Genre and Release Year---
Group by Year, Filter by Genre
Average execution time for query 0: 0.1993 seconds
Average execution time for query 1: 0.1981 seconds
Group by Year and Month, Filter by Genre and Year
Average execution time for query 0: 0.2053 seconds
Average execution time for query 1: 0.2067 seconds
```

Using the formula below, the group calculated the optimization rate for each query.

$$Optimization\ Rate = \frac{pre\ optimized\ time - post\ optimized\ time}{pre\ optimized\ time} \times 100$$

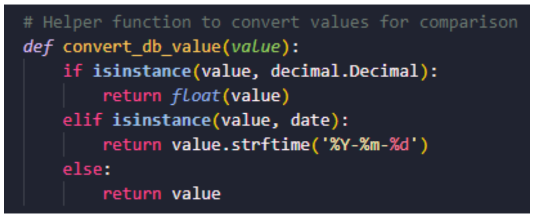
Averaging the optimization rates across all test queries, the overall **optimization rate is approximately 57.25%. This entails that the average query speed was increased by over 2 times.**

The group selected the most optimal query for each of the four reports. Queries that utilized explicit INNER JOINs performed particularly well, and this approach was implemented in the OLAP Dashboard Application.

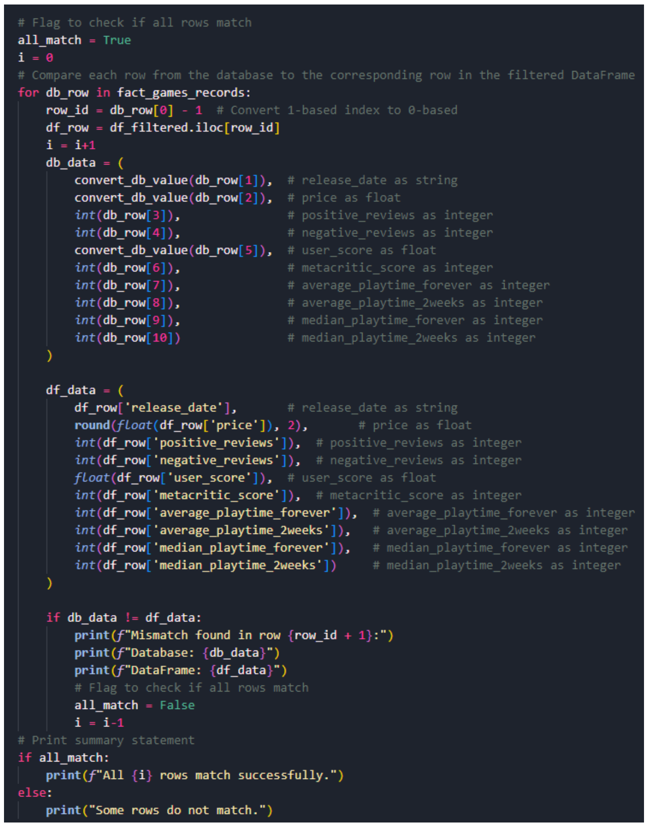
C. Data Insert Validation

This section explains the methodology for validating the ETL process. The group validated the correctness of the ETL by comparing the contents of the dataframe to the contents of the database. The ETL process was validated in 3 ways: comparing the contents of the dataframe to the database, checking for any null values in the database, and checking for any duplicate records. The test tool was coded in python, where the pandas and numpy libraries were used.

The code compares data inserted into the database with the corresponding rows in the filtered DataFrame (df_filtered) to ensure accurate insertion. A helper function, convert_db_value, converts database-specific types like decimal.Decimal and date for easy comparison with the DataFrame. Each row from the database is mapped to the corresponding DataFrame row, and key fields such as release date, price (rounded to two decimal places), user score, and playtime are compared. If any discrepancies are found, they are printed along with a flag indicating mismatches. After processing all rows, the code confirms whether all rows matched successfully or if there were mismatches.

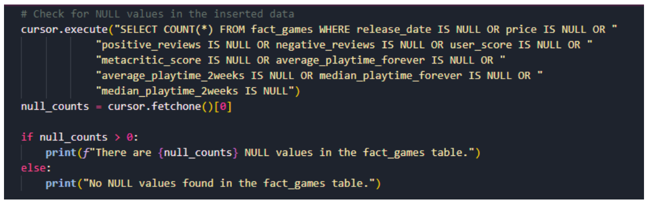


```python
# Helper function to convert values for comparison
def convert_db_value(value):
    if isinstance(value, decimal.Decimal):
        return float(value)
    elif isinstance(value, date):
        return value.strftime('%Y-%m-%d')
    else:
        return value
```



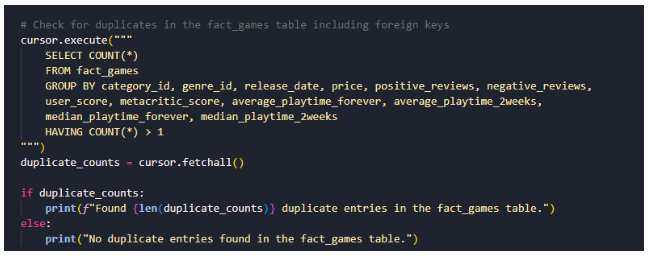
```
# Flag to check if all rows match
all_match = True
i = 0
# Compare each row from the database to the corresponding row in the filtered DataFrame
for db_row in fact_games_records:
    row_id = db_row[0] - 1  # Convert 1-based index to 0-based
    df_row = df_filtered.iloc[row_id]
    i = i+1
    db_data = (
        convert_db_value(db_row[1]),  # release_date as string
        convert_db_value(db_row[2]),  # price as float
        int(db_row[3]),               # positive_reviews as integer
        int(db_row[4]),               # negative_reviews as integer
        convert_db_value(db_row[5]),  # user_score as float
        int(db_row[6]),               # metacritic_score as integer
        int(db_row[7]),               # average_playtime_forever as integer
        int(db_row[8]),               # average_playtime_2weeks as integer
        int(db_row[9]),               # median_playtime_forever as integer
        int(db_row[10])               # median_playtime_2weeks as integer
    )

    df_data = (
        df_row['release_date'],       # release_date as string
        round(float(df_row['price']), 2),       # price as float
        int(df_row['positive_reviews']),  # positive_reviews as integer
        int(df_row['negative_reviews']),  # negative_reviews as integer
        float(df_row['user_score']),  # user_score as float
        int(df_row['metacritic_score']),  # metacritic_score as integer
        int(df_row['average_playtime_forever']),  # average_playtime_forever as integer
        int(df_row['average_playtime_2weeks']),   # average_playtime_2weeks as integer
        int(df_row['median_playtime_forever']),   # median_playtime_forever as integer
        int(df_row['median_playtime_2weeks'])     # median_playtime_2weeks as integer
    )

    if db_data != df_data:
        print(f"Mismatch found in row {row_id + 1}:")
        print(f"Database: {db_data}")
        print(f"DataFrame: {df_data}")
        # Flag to check if all rows match
        all_match = False
        i = i-1
# Print summary statement
if all_match:
    print(f"All {i} rows match successfully.")
else:
    print("Some rows do not match.")
```

The code checks for any NULL values in the database. A SQL query counts rows where any of these columns are NULL, ensuring no critical data is missing. If NULL values are found, their count is printed; otherwise, the code confirms that no NULL values are present, ensuring data completeness.



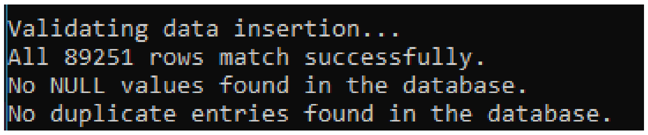
```
# Check for NULL values in the inserted data
cursor.execute("SELECT COUNT(*) FROM fact_games WHERE release_date IS NULL OR price IS NULL OR "
            "positive_reviews IS NULL OR negative_reviews IS NULL OR user_score IS NULL OR "
            "metacritic_score IS NULL OR average_playtime_forever IS NULL OR "
            "average_playtime_2weeks IS NULL OR median_playtime_forever IS NULL OR "
            "median_playtime_2weeks IS NULL")
null_counts = cursor.fetchone()[0]

if null_counts > 0:
    print(f"There are {null_counts} NULL values in the fact_games table.")
else:
    print("No NULL values found in the fact_games table.")
```

Lastly, the code checks for duplicate rows in the database, including foreign keys (*category_id* and *genre_id*). It runs a query that groups records by key fields like release date, price, and user score. The query identifies rows that appear more than once. If duplicates are found, the code prints how many duplicate entries exist; otherwise, it confirms no duplicates, ensuring data integrity.



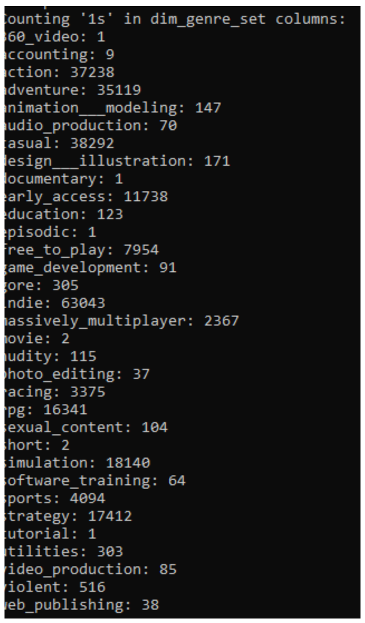
```
# Check for duplicates in the fact_games table including foreign keys
cursor.execute("""
    SELECT COUNT(*)
    FROM fact_games
    GROUP BY category_id, genre_id, release_date, price, positive_reviews, negative_reviews,
    user_score, metacritic_score, average_playtime_forever, average_playtime_2weeks,
    median_playtime_forever, median_playtime_2weeks
    HAVING COUNT(*) > 1
""")
duplicate_counts = cursor.fetchall()

if duplicate_counts:
    print(f"Found {len(duplicate_counts)} duplicate entries in the fact_games table.")
else:
    print("No duplicate entries found in the fact_games table.")
```

After completing all test phases for ETL validation, the results are printed, as shown below, confirming the integrity of the data inserted into the database.

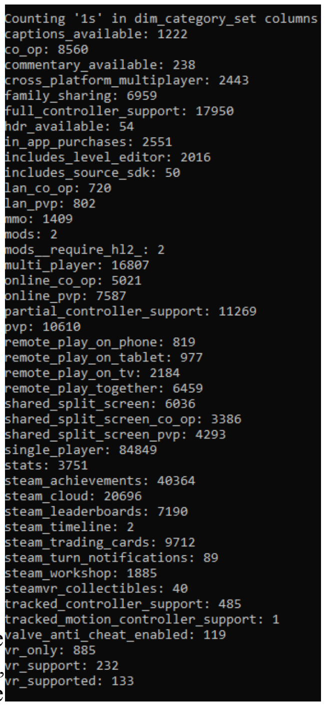


```
Validating data insertion...
All 89251 rows match successfully.
No NULL values found in the database.
No duplicate entries found in the database.
```

In addition, the 1s for each column in the dimension tables were counted using Python for validation purposes. The results of the code are as shown below.



```
ounting '1s' in dim_genre_set columns:
60_video: 1
ccounting: 9
ction: 37238
dventure: 35119
nimation___modeling: 147
udio_production: 70
asual: 38292
esign___illustration: 171
ocumentary: 1
arly_access: 11738
ducation: 123
pisodic: 1
ree_to_play: 7954
ame_development: 91
ore: 305
ndie: 63043
assively_multiplayer: 2367
ovie: 2
udity: 115
hoto_editing: 37
acing: 3375
pg: 16341
exual_content: 104
hort: 2
imulation: 18140
oftware_training: 64
ports: 4094
trategy: 17412
utorial: 1
tilities: 303
ideo_production: 85
iolent: 516
eb_publishing: 38
```



```
Counting '1s' in dim_category_set columns
captions_available: 1222
co_op: 8560
commentary_available: 238
cross_platform_multiplayer: 2443
family_sharing: 6959
full_controller_support: 17950
hdr_available: 54
in_app_purchases: 2551
includes_level_editor: 2016
includes_source_sdk: 50
lan_co_op: 720
lan_pvp: 802
mmo: 1409
mods: 2
mods__require_hl2_: 2
multi_player: 16807
online_co_op: 5021
online_pvp: 7587
partial_controller_support: 11269
pvp: 10610
remote_play_on_phone: 819
remote_play_on_tablet: 977
remote_play_on_tv: 2184
remote_play_together: 6459
shared_split_screen: 6036
shared_split_screen_co_op: 3386
shared_split_screen_pvp: 4293
single_player: 84849
stats: 3751
steam_achievements: 40364
steam_cloud: 20696
steam_leaderboards: 7190
steam_timeline: 2
steam_trading_cards: 9712
steam_turn_notifications: 89
steam_workshop: 1885
steamvr_collectibles: 40
tracked_controller_support: 485
tracked_motion_controller_support: 1
valve_anti_cheat_enabled: 119
vr_only: 885
vr_support: 232
vr_supported: 133
```

Once the integrity of the inserted data has been ensured, OLAP operations can be performed safely and with credibility.

7. Conclusion

In this project, a data warehouse was created using the most relevant columns from the Steam dataset. The schema followed a star design, with one-to-one relationships from the fact table to the dimension tables. SQL queries were used to generate reports on game performance and reviews over time, which were then optimized for efficient execution and visualized in Tableau through an OLAP dashboard.

This project demonstrates the importance of creating an efficient data warehouse as a consistent foundation for analytics, reporting, and informed decision-making. The ETL process is crucial, as it keeps the database updated by consistently loading credible data. Enabling OLAP functionality allows for complex, multidimensional data analysis, providing historical insights that support decision-making beyond the transactional capabilities of OLTP (Online Transaction Processing), which focuses on real-time transactions. Query optimization is essential for reducing processing time and resource usage, with strategies including indexing, query rewriting, and using materialized views. Custom indexes are particularly useful when queries require specific, frequently accessed columns not covered by MySQL's automatic indexing.

The findings in this report offer valuable insights for developers and business intelligence teams into Steam game performance, based on sales, reviews, and ratings. The strategies and techniques applied in this project can also serve as guidance for other database developers in designing efficient database schemas and optimized queries.

8. References

Estuary.dev. 2024. MongoDB to MySQL: Transforming NoSQL to Relational Database. https://estuary.dev/mongodb-to-mysql/

MySQL :: MySQL 8.4 Reference Manual :: 10.2.2.4 Optimizing Derived Tables, View References, and Common Table Expressions

with Merging or Materialization. (n.d.). https://dev.mysql.com/doc/refman/8.4/en/derived-table-optimization.html?fbclid=IwZXh0bgNhZW0CMTEAAR3cO-6ATCneWzOW7uagUFcG5Dwm81DnQdFJl7-PIUiyRA9agL08uWF7hbw_aem_W3UGrgFUvtqT-o5WuM0EyQ

Reference Manual :: 10.3.6 Multiple-Column Indexes. (n.d.). https://dev.mysql.com/doc/refman/8.4/en/multiple-column-indexes.html?fbclid=IwZXh0bgNhZW0CMTEAAR3ImsGMwXa51P6KZy0DZUpnOkIUfRM790EODYh4EwAC0Ct_nk8Ow68XJyw_aem_2HygjqZGjmlKR6DAetwBXA

Reference Manual :: 10.8 Understanding the Query Execution Plan. (n.d.). https://dev.mysql.com/doc/refman/8.4/en/execution-plan-information.html?fbclid=IwZXh0bgNhZW0CMTEAAR161TlnUxZoxq6rAX82KalMXTK4Bg7uzHBlkGnLJtvfs2orO4bexAT-tBU_aem_12Q91aWuiz8IxB7cK8UjRQ

8.1 Record of Contribution

Group Members:

P1: Marian Ricci N. Ariaga[1]

P2: Dana Louise A. Guillarte[2]

P3: Yesha P. Llorando[3]

P4: Chrysille Grace L. So[4]

Activity	**P1**	**P2**	**P3**	**P4**
Schema Design	20.0	20.0	20.0	40.0
ETL	50.0	50.0	0.0	0.0
Query Writing and Optimization	20.0	20.0	10.0	50.0
OLAP Application	15.0	15.0	70.0	0.0
Documentation	20.0	20.0	25.0	35.0
Raw Total	125.0	125.0	125.0	125.0
TOTAL	**25.0**	**25.0**	**25.0**	**25.0**