# Group 26: COMP 4380 Project

Colin McDonell Diroma Umukoro Eddie Wat

***Abstract*—The document presents an analysis of Amazon USA customer reviews dataset, which contains reviews and metadata of products from 1995 to 2015. The dataset is obtained from Kaggle and is stored in a DB2 database running on a virtual machine. The document describes the dataset, the queries used for analysis, and the optimizations performed to improve query runtimes. The optimizations involve creating indexes on relevant columns, rewriting the queries to make use of the indexes and reconfiguring the database for improved performance. The results show significant reductions in query runtimes after optimization, demonstrating the effectiveness of the applied techniques. Further analysis and calculations are presented in the appendix. Overall, the document provides insights into the analysis of the Amazon USA customer reviews dataset and the optimizations applied to improve query performance.**

## I. Introduction

The dataset contains Amazon USA customer reviews on their products. This is a collection of reviews with metadata, on products between the dates of 1995 to 2015. The dataset was chosen because it contains a large amount of interesting data that will require optimized queries in order for their runtime to be reasonable. Another reason is the width of each table allows for varied and interesting queries and analysis. Ten queries were performed on the dataset and optimized to improve runtime. Various optimization strategies were used such as indexes, rewriting queries, creating temporary tables, and reconfiguring the database. We found that for all the queries, we were able to show some improvement to the runtimes through optimization. On average, we were able to reduce the runtime of the queries by 65%.

## II. Data

### A. Description of Data

The subset that we used as our dataset contains 7 tables, with 15 columns of data for each. The table below shows the file size and number of rows for each table in the database

TABLE I
Table Metadata

| Table Name | Metadata | |
|---|---|---|
| | *Size (MB)* | *Number of Records* |
| Apparel | 4108 | 5874693 |
| Appliances | 112 | 96792 |
| Automotive | 2680 | 3597760 |
| Multilingual | 6688 | 6882088 |
| Music | 6400 | 4722502 |
| PC | 6843 | 6902673 |
| Wireless | 7911 | 8985135 |

### B. Data Source

The dataset we used was downloaded from Kaggle. [1] The source of the dataset was from Amazon and was compiled by Kaggle User Cynthia Rempel. More information about the dataset can be found on the Amazon AWS website. [2] [3]

### C. DBMS

The DBMS that we are using for the project is DB2 for Linux, Unix and Windows V11.1. It is running as a virtual machine with 4 cores of a Xeon E5-2640v4 CPU @ 2.4GHz, 16 GB memory, and 100 GB of disk on a SATA SSD

## III. Experiments/Analysis

We will optimize/analyze queries to improve their runtime and optimize the database

The tables are divided by product category and each contains 15 columns. Since most queries are executed on all of the records, most queries consist of unioning a projection of each table. As a shorthand, we will show the first projection statement and the last projection statement. For example, a statement that counts the number of records from all tables would be written as:

```
WITH all AS (
SELECT product_id FROM MCDONELC.APPAREL
UNION
...
UNION
SELECT product_id FROM MCDONELC.WIRELESS)
SELECT count(*) FROM all;
```

The `SELECT` statement for tables `APPLIANCES`, `AUTOMOTIVE`, `MULTILINGUAL`, `MUSIC` and `PC` are implied by `...`.

### *Query 1*

The query below shows the product categories that have the greatest percentage of 5-star reviews.

```
WITH review_count as (
    SELECT product_category,
    sum(r) as reviews
    FROM (
        SELECT product_category,
        count(review_id) as r
        FROM MCDONELC.apparel
        GROUP BY product_category
        UNION ALL
        ...
        UNION ALL
```

```
        SELECT product_category,
        count(review_id) as r
        FROM MCDONELC.wireless
    GROUP BY product_category
    )
    GROUP BY product_category
),
fstar_count as (
    SELECT product_category,
    sum(five) as five_stars
    FROM (
        SELECT product_category,
        count(STAR_RATING) as five
        FROM MCDONELC.apparel
        WHERE STAR_RATING = 5
        GROUP BY product_category
        UNION ALL
        ...
        UNION ALL
        SELECT product_category,
        count(STAR_RATING) as five
        FROM MCDONELC.wireless
        WHERE STAR_RATING = 5
           GROUP BY product_category
    )
    GROUP BY product_category
)
SELECT review_count.product_category,
reviews, five_stars,
DEC(DEC(five_stars,9,2)/
DEC(reviews,9,2)*100,3,1)
AS five_stars_per_review FROM
review_count
JOIN fstar_count
ON review_count.product_category
LIKE fstar_count.product_category
ORDER BY five_stars_per_review DESC
LIMIT 10;
```

*Optimization:* In order to improve the runtime of the query, an index was created on star_rating and an index was created on product_category.

The original runtime of this query before optimization averaged 56 seconds. Once the query optimizations were implemented, the average time improved to 33 seconds. This significant improvement results from removing the requirement of any table scans and is completed with index-only scans.

According to the documentation, DB2 will use available indexes whenever possible to avoid accessing tuples in the table. [4] Since there is an index on review_id automatically (created when declared as the primary key), all of the data we need is contained in indexes.

Additional analysis on this query is contained in the APPENDIX. Through database reconfiguration, a 12% reduction in page I/Os can be achieved. The calculations are to lengthy to be included in the main part of the report.

*Query 2*

The query below counts the number of reviews per product category between 1995-2000.

```
SELECT product_category, COUNT(review_id) AS count
FROM (
    SELECT product_category, review_id
    FROM MCDONELC.APPAREL
    WHERE review_date
    BETWEEN '1995-01-01'
    AND '2000-12-31'
    UNION
    ...
    UNION
    SELECT product_category, review_id
    FROM MCDONELC.WIRELESS
    WHERE review_date
    BETWEEN '1995-01-01'
    AND '2000-12-31'
)
GROUP BY product_category
ORDER BY count DESC
LIMIT 10;
```

*Optimization:* In order to improve the runtime of the query, an index was created on review_date and an index was created on product_category.

The original runtime of this query before optimization averaged 24 seconds. Once the query optimizations were implemented, the average time improved to 2 seconds. This significant improvement results from removing the requirement of any table scans and is completed with index-only scans.

*Query 3*

The query below finds the top 10 customers who have written the most reviews. The original SQL query is written below.

```
WITH all AS (
    SELECT customer_id, review_id
    FROM MCDONELC.APPAREL
    UNION
    ...
    UNION
    SELECT customer_id, review_id
    FROM MCDONELC.WIRELESS)
SELECT customer_id, count(review_id)
AS number_of_reviews FROM all
GROUP BY customer_id
ORDER BY number_of_reviews DESC
LIMIT 10;
```

*Optimization:* Adding an index on customer_id did not improve the running speed of query 3 as it was originally written in Phase 2 of the project. The reason it did not improve runtime

is that the aggregation of the count of review_id grouped by customer_id is performed by sorting by customer_id and totalling over each group for each customer_id. The initial way that the query was written performed this grouping step after all tables (APPAREL,..., WIRELESS) were joined. No index existed on this large table so the aggregate function could not make use of any index scan. The aggregate function, therefore, was much slower because it had to sort a lot of records.

In order to make use of the index on customer_id, an aggregate over each subtable was taken before joining them. Doing the aggregation in this manner makes use of the index and made the query much quicker. The final aggregation of the sum of review_id for each customer_id still could not use an index (because no index existed on the joined table) but there were much fewer rows to work with.

Below, is an better version of the query that is able to make use of the indexes.

```
WITH all AS (
    SELECT customer_id, COUNT(review_id)
    AS reviews FROM MCDONELC.APPAREL
    GROUP BY customer_id
    UNION
    ...
    UNION
    SELECT customer_id, COUNT(review_id)
    AS reviews FROM MCDONELC.WIRELESS
    GROUP BY customer_id),
SELECT customer_id, SUM(reviews)
AS total_reviews FROM all
GROUP BY customer_id
ORDER BY total_reviews DESC
LIMIT 10;
```

Table II shows the differences between the original and optimized version of the query.

TABLE II
ORIGINAL VS. OPTIMIZED QUERY

| | **Original Query** | **Optimized Query** |
|---|---|---|
| Rows from `apparel` | 5874693 | 3216362 |
| Rows from `appliances` | 96792 | 91212 |
| Rows from `automotive` | 3497760 | 1901331 |
| Rows from `multilingual` | 6882248 | 4089251 |
| Rows from `music` | 4722502 | 1932921 |
| Rows from `pc` | 6902673 | 4053822 |
| Rows from `wireless` | 8985135 | 5191173 |
| Rows after `UNION` | 36961803 | 20476072 |
| Average Runtime (s) | 60 | 16 |
| Table aggregation (s) | 44 | 8 |
| Final aggregation (s) | 16 | 8 |

The results in Table II show that there are many more rows of results returned from each table by the original SQL statement vs. the improved one. This makes for more costly union operation. In fact, the union operation accounts for a bigger portion of the I/O cost than the aggregation operation. Rewriting the SQL query an creating an index on customer_id improved runtime from 60 seconds to 16 seconds. This is a 73% speed increase.

*Query 4*

The query finds the top 10 customers with the most helpful votes

```
WITH all AS (
    SELECT customer_id, review_id,
    helpful_votes
    FROM MCDONELC.APPAREL
    UNION
    ...
    UNION
    SELECT customer_id, review_id,
    helpful_votes
    FROM MCDONELC.WIRELESS),
votes AS (
    SELECT customer_id,
    COUNT(review_id)
    AS number_of_reviews,
    SUM(helpful_votes)
    AS total_votes FROM all
    GROUP BY customer_id)
SELECT votes.customer_id,
votes.number_of_reviews,
votes.total_votes,
votes.total_votes/
votes.number_of_reviews
AS votes_per_review
FROM votes
ORDER BY votes_per_review DESC
LIMIT 10;
```

*Optimization:* Similarly to Query 3, the index on customer_id did not improve the runtime. When the SQL statement was rewritten to aggregate before the UNION it took even longer than the original query (65s vs 90s). This can be attributed to the fact that each table had to be read to get the helpful_votes of each review so a full table scan was needed and the index was not used. The increased number of aggregation actions resulted in a slower query overall.

In order to improve the runtime of the query, an index was created on (customer_id, helpful_votes). This meant that all data required by the query was stored in an index making it an index-only scan. This resulted in a significant speed increase from 60 seconds to 20 seconds.

The optimized query is shown below

```
WITH all AS (
    SELECT customer_id, count(review_id)
    AS reviews, sum(helpful_votes)
    AS votes
    FROM MCDONELC.APPAREL
    GROUP BY customer_id
    UNION
    ...
```

```
    UNION
    SELECT customer_id, count(review_id)
    AS reviews, sum(helpful_votes)
    AS votes
    FROM MCDONELC.WIRELESS
    GROUP BY customer_id),
totals AS (
    SELECT customer_id, sum(reviews)
    AS total_reviews,
    sum(votes) AS total_votes
    FROM all
    GROUP BY customer_id)
SELECT customer_id, total_reviews,
total_votes,
total_votes/total_reviews as vpr
FROM totals ORDER BY vpr DESC
LIMIT 10;
```

*Query 5*

The query below returns the top ten most reviewed products.

```
WITH all AS (
SELECT review_id, product_id,
product_title FROM MCDONELC.apparel
UNION
...
UNION
SELECT review_id, product_id,
product_title from MCDONELC.wireless)
SELECT product_title,
count(review_id) as review_count FROM all
GROUP BY product_title
ORDER BY review_count DESC
LIMIT 10;
```

*Optimization:* We will show through the theoretical analysis, the potential optimization of the query using a temporary table in IBM DB2. We were given permission to only perform an analysis, as we do not have access to the database to make the implementation of this optimization technique.

Temporary tables are used to store intermediate results, reducing the amount of data processing needed during query execution. Complex calculations like the one above which took 3 minutes can be performed once and stored. Subsequent queries can then reference the temporary table instead of recomputing the results. This will result in better query performance.

Just like the materialized query table, the temporary table can also be used for query rewrite in DB2 and it can be indexed. This will further speed up the data retrieval based on indexed columns. By leveraging indexes and query rewrite, the query performance can be further optimized.

Temporary tables can help us to reduce the complexity of the original query. Instead of performing multiple UNION operations on multiple tables, the query can be simplified to operate on a single temporary table. This will make the query easier to understand, maintain and optimize, leading to improved query performance. Temporary tables can help reduce the I/O cost drastically as it can store the result on memory or disk depending on the database configuration and accessing data from a temporary table in memory can be faster compared to repeatedly reading data from multiple tables. This will have a high impact on our query performance since we are working data size.

*Query 6*

The query below returns the total number of reviews for each star rating

```
WITH all AS (
    SELECT review_id, star_rating
    FROM MCDONELC.apparel
    UNION
    ...
    UNION
    SELECT review_id, star_rating
    FROM MCDONELC.wireless)
SELECT star_rating,
count(review_id) AS num_reviews
FROM all
GROUP BY star_rating
```

*Optimization:* To improve the runtime of the above query, a B+-tree index was added to the star_rating column. This optimization made no difference after executing the above query. The way the above query was written did not use the index.

The above query combines data from multiple tables using the UNION operator and then sorts the result by the star_rating column before grouping. Since the sorting happens after the union, the index for each individual table is not used.

In order to improve the performance of the query, we took the approach of rewriting the above query to group before the join. This also allows for index-only access which has significant speed improvements.

The improved query is written below.

```
SELECT star_rating, sum(r) AS num_reviews
FROM (
    SELECT count(review_id) as r,
    star_rating FROM MCDONELC.apparel
    GROUP BY star_rating
    UNION ALL
    ...
    UNION ALL
    SELECT count(review_id) as r,
    star_rating FROM MCDONELC.wireless
    GROUP BY star_rating
)
GROUP BY star_rating;
```

The optimized query also uses the UNION ALL operator instead which eliminates the overhead of eliminating duplicates.

The optimized query along with the index on star_rating resulted in a significant improvement over the original query. The original query had an execution time of 60 seconds, while the optimized query had an execution of 2 seconds. This is a reduction in execution time of approximately 97%.

*Query 7*

```
SELECT t1.customer_id,
SUM(COALESCE(t1.helpful_votes, 0)
+ COALESCE(t2.helpful_votes, 0))
AS total_helpful_votes
FROM MCDONELC.apparel t1
RIGHT OUTER JOIN MCDONELC.music t2
ON t1.customer_id = t2.customer_id
GROUP BY t1.customer_id
ORDER BY total_helpful_votes DESC;
```

*Optimization:* The query was optimized by adding indexes on customer_id to increase the speed of the JOIN. The runtime was improved from 10 seconds to 4 seconds on average. The customer_id column is not sorted because the table's primary key is on the review_id column so the created index is not clustered. In the original query without an index, the join condition would require searching the inner table for a matching customer_id, (unsorted column) which requires a table scan. Sorting tables is expensive and should be avoided, instead we add an index on customer_id to improve performance greatly.

*Query 8*

The query below gives the number of reviews with a star_rating = 1 and have at least one total_vote.

```
SELECT product_category,
COUNT(*) as count,
SUM(TOTAL_VOTES) as total_votes
FROM (
    SELECT product_category,
    TOTAL_VOTES
    FROM MCDONELC.apparel
    WHERE TOTAL_VOTES > 0
    AND STAR_RATING = 1
    UNION ALL
    ...
    UNION ALL
    SELECT product_category,
    TOTAL_VOTES
    FROM mcdonelc.wireless
    WHERE TOTAL_VOTES > 0
    AND STAR_RATING = 1
) AS all_reviews
GROUP BY product_category
ORDER BY COUNT(*) DESC
LIMIT 10;
```

*Optimization:* An index on the TOTAL_VOTES and STAR_RATING columns were added to improve runtime. However, they did not make a difference. It actually resulted in an increase in runtime from 21s to 149s. This is because the order is not very effective. Since most reviews have greater than zero TOTAL_VOTES, most of the index had to be scanned for records where STAR_RATING = 1. Creating indexes on individual columns resulted in a slight increase in runtime from 21s to 19s. This slight improvement results from all the data required by the query being in an index. Therefore the query can use an index-only scan to get the data.

Since we are running the DB2 database on a server hosted by the University of Manitoba, the slight performance gain could be due to either performance gains or load on the server as fluctuations in results are close enough to be uncertain. Creating indexes for a potential 10% gain in execution speed might not be worth the cost of maintaining the indexes since having no indexes implemented on those two attributes makes little difference. The attribute of STAR_RATING ranges from 1 to 5. With such limited values, an index would also have a limited effect on improving runtime. For our specific query, TOTAL_VOTES condition can be any value other than 0 or null which similarly makes an index on this column ineffective.

Modifications to use join operators for this query instead of UNION ALL were ineffective and produced no improvements.

A simplification of this query and many other queries in our database can be achieved by merging all tables into one large table as all our columns are identical, and to add another column to mark which table category every entry belongs to. This would get rid of the need for all the UNION operators and simplify the query block.

*Query 9*

The query below returns information about customers who reviewed both automotive and computer products.

```
SELECT * FROM MCDONELC.automotive t1
INNER JOIN
MCDONELC.pc t2
ON t1.customer_id=t2.customer_id;
```

The query was optimized by adding a hash index on the customer_id column.

From the different join algorithms covered in class, the above query is most likely using a block-nested-loop join which does not require any indexes as it can be used with any kind of join conditions. It is expensive in terms of I/O cost since it scans every page of both tables to join them.

We added a hash index on the customer_id column which enables the database management system to quickly locate the matching record and can help with the query performance by reducing the cost of execution in time.

The hash index hashed values in the customer_id column and uses the resulting hash values as pointers to the corresponding rows in the table. This makes searching for the matching records much faster, eliminating the need for a full table scan.

The hash index optimization on customer_id resulted in a significant reduction in execution time from 46 seconds to 8

seconds. Therefore, the optimized query runs over 81% faster with the hash index.

*Query 10*

The query below gets review_id, helpful_votes, product_category, product_title, product_id, customer_id, product_parent, star_rating from all tables in the database

```
WITH all AS (
    SELECT review_id,
    helpful_votes, product_category,
    product_title, product_id,
    customer_id, product_parent,
    star_rating
    FROM MCDONELC.apparel
    UNION
    ...
    UNION
    SELECT review_id,
    helpful_votes, product_category,
    product_title, product_id,
    customer_id, product_parent,
    star_rating
FROM MCDONELC.wireless
)
SELECT *
FROM all;
```

*Query Optimization Using a Materialized Query Table:* The optimization for this query is making a materialized query table for the result of the query. This is a theoretical analysis on potential optimization of the below query using materialized query table in IBM DB2.

Permission was taken to make only analysis, as we do not have access to the database to make the implementation of this optimization technique.

Materialize query tables (MQT) are permanent database objects which can be created by the database user and can be used to store the result of a query or the portion of a query as a physical table in the database. They are used to improve query performance by pre-computing the result of expensive query like the one above so that the same result can be retrieved faster in subsequent query execution.

This means that the results are already computed and stored and subsequent queries can directly access the pre-computed results without having to recompute them. This can significantly reduce the amount of time spent during query execution since all of our queries UNION multiple tables together.

MQT can also be indexed which will further speed up the query execution time by allowing for efficient data retrieval based on the indexed columns. This will help us to optimize the query performance. DB2 also has a query rewrite feature that automatically rewrites queries so they can use MQT when appropriate. This means that even if the original query we are executing does not explicitly reference the materialized query table, the database can automatically rewrite the query to use the pre-computed result stored in the MQT if it determines that it will result in a faster query execution. This modifies the query performance without modifying the original query.

MQT can reduce the I/O cost of queries since it stores the precomputed result in a physical table. This can result in lower I/O cost during query execution, leading to an improved performance, especially for the large data sets we are working with, which took about 90s to execute the above query.

The difference between MQT from temporary tables is that MQT are permanent database objects that store precomputed results of queries to optimize performance while temporary tables are created temporarily during a transaction and are used to store intermediate results.

## IV. Discussion

Table III shows a summary of the runtime improvements to the queries through optimization. The queries that are not shown in the table are the queries where only hypothetical analysis was done. Through optimization, an average of 65% increase in speed was achieved.

TABLE III
Query Runtime Improvement Summary

| Query | Original Runtime (s) | Optimized Runtime (s) | Percentage Improvement (%) |
|---|---|---|---|
| Query 1 | 56 | 33 | 41 |
| Query 2 | 24 | 2 | 92 |
| Query 3 | 60 | 16 | 73 |
| Query 4 | 60 | 20 | 67 |
| Query 6 | 60 | 2 | 97 |
| Query 7 | 10 | 4 | 60 |
| Query 8 | 21 | 19 | 10 |
| Query 9 | 46 | 8 | 83 |
| **Average** | **42** | **13** | **65** |

Attempts were made to create temporary tables and materialized query tables with the `CREATE TEMPORARY TABLE` and `CREATE TABLE` operation but we did not have authorization to do so in the DB2 server. Unfortunately this wasn't able to be resolved in time with troubleshooting, so we unfortunately had to avoid this operation. In the process of experimentation and analysis, we also found that the default page size is 4096 bytes through the SQL query of `SELECT bpname.pagesize FROM syscat.bufferpools`.

Further optimizations by changing the schema were considered but not implemented due to shortage of time and uncertainty of results. Extracting the review associated attributes and the product attributes into their own tables could cut down on the size of each row (On average review_body is 332 bytes at the largest compared to product_title at 69 bytes at the second largest) and provide some flexibility for how tables are sorted.

## References


[1] C. Rempel, "Amazon US Customer Reviews Dataset." kaggle.com. https://www.kaggle.com/datasets/cynthiarempel/amazon-us-customer-reviews-dataset (accessed April 12, 2023)

[2] "Amazon Customer Reviews Dataset README." https://s3.amazonaws.com/amazon-reviews-pds/readme.html (accessed April 12, 2023)
[3] "Amazon Customer Reviews Dataset INFO." https://s3.amazonaws.com/amazon-reviews-pds/tsv/index.txt (accessed April 12, 2023)
[4] IBM Documentation, "Types of index access." https://www.ibm.com/docs/en/db2/11.5?topic=methods-types-index-access (accessed April 12, 2023)
[5] An email. (Jeff Durston, private communication, March 20, 2023) stating specifications of DB2 database.
[6] IBM Documentation, "Table page sizes." https://www.ibm.com/docs/en/db2/11.5?topic=tables-table-page-sizes (accessed April 12, 2023)
[7] IBM Documentation, "Space requirements for tables." https://www.ibm.com/docs/en/db2/11.5?topic=tables-space-requirements (accessed April 12, 2023)
[8] Columbia University, "Buffer Pool Size (buffpage)." https://www.columbia.edu/sec/acis/db2/db2help/sqld0001.htm (accessed April 12, 2023)

## APPENDIX A: CALCULATIONS TO COMPARE DATABASE CONFIGURATIONS

### *Current Configuration*

Currently, our database consists of 7 tables, loosely divided based on product category. This is how the data was configured in the files we downloaded to populate the database. All tables are 15 columns wide. The average file size of each column (in bytes) is shown in table IV. The command `AVG(LENGTH([Attribute]))` was used to get the average length for the text columns since each character in `VARCHAR` is a byte (Octet).

TABLE IV
AVERAGE ATTRIBUTE SIZES

| Attribute | Size (Bytes) |
|---|---|
| Market_place | 2 |
| Customer_id | 8 |
| Review_id | 13 |
| Product_id | 10 |
| Product_parent | 8 |
| Product_title | 69 |
| Product_category | 10 |
| Star_rating | 4 |
| Helpful_votes | 4 |
| Total_votes | 4 |
| Vine | 4 |
| Verified_purchase | 4 |
| Review_headline | 34 |
| Review_body | 332 |
| Review_date | 10 |
| **Record Size** | **516** |

The page size of our database is the default size of 4096 bytes. [6] This information was found by running the SQL command

```
SELECT bpname,pagesize
FROM syscat.bufferpools
```

Another simplifying assumption that we will make is that the data for review_body is stored along with the record. However, the db2 documentation states the CLOB data is not stored along with the records, only the description of the data is stored along with the records. [7] We will ignore this fact to simplify calculations and to exagerate the speed improvement when we reconfigure the database as the reduction in page size due to moving review_body to a separate table will be dramatic.

For the calculations, B = 1000. This is the default number created for UNIX-based platforms by version 5 of DB2. [8] We were unable to determine the number of buffer pages for Version 11 that we are using.

Table V shows information about the database over all tables.

TABLE V
CURRRENT DATABASE CONFIGURATION

| | |
|---|---|
| **Total records** | 36961803 |
| **Record size (Bytes)** | 516 |
| **Page size (Bytes)** | 4096 |
| **Records/page** | 7 |
| **Pages** | 5280258 |

### *New Configuration*

In order to reduce the width of each table and combine all the tables, the data would be separated into 3 large tables. a products table, review_details table, and reviews table as shown below:

**products**(product_id, product_parent, product_title, product_category) sorted by product_id

**reveiw_details**(review_id, market_place, star_rating, helpful_votes, total_votes,vine, verified_purchase, review_headline, review_body, review_date) sorted by review_id.

**reviews**(review_id, customer_id,product_id) sorted by review_id.

Table VI shows information about the above tables. We keep our assumptions made for original attribute sizes.

TABLE VI
NEW DATABASE CONFIGURATION

| Table | Records | Size (Bytes) | Records/page | Pages |
|---|---|---|---|---|
| **Products** | 5257914 | 97 | 42 | 125189 |
| **Review_details** | 36961803 | 411 | 9 | 4106867 |
| **Reviews** | 36961803 | 31 | 132 | 280014 |

To compare the two table configurations, We will calculate the I/O cost for a query for both configurations. We will assume that no index-only scans are done.

### *Query*

The query calculating the percentage of reviews that are have 5 stars per product category will be used to compare the I/O cost for the two database configurations.

*Query for Current Database Configuration:*

```
WITH review_count as (
    SELECT product_category,
    sum(r) as reviews
    FROM (
        SELECT product_category,
        count(review_id) as r
        FROM MCDONELC.apparel
        GROUP BY product_category
        UNION ALL
        ...
        UNION ALL
        SELECT product_category,
        count(review_id) as r
        FROM MCDONELC.wireless
        GROUP BY product_category
    )
    GROUP BY product_category
),
fstar_count as (
    SELECT product_category,
    sum(five) as five_stars
    FROM (
        SELECT product_category,
        count(STAR_RATING) as five
        FROM MCDONELC.apparel
        WHERE STAR_RATING = 5
        GROUP BY product_category
        UNION ALL
        ...
        UNION ALL
        SELECT product_category,
        count(STAR_RATING) as five
        FROM MCDONELC.wireless
        WHERE STAR_RATING = 5
        GROUP BY product_category
    )
    GROUP BY product_category
)
SELECT review_count.product_category,
reviews, five_stars,
DEC(DEC(five_stars,9,2)/
DEC(reviews,9,2)*100,3,1)
AS five_stars_per_review FROM
review_count
JOIN fstar_count
ON review_count.product_category
LIKE fstar_count.product_category
ORDER BY five_stars_per_review DESC
LIMIT 10;
```

*I/O Cost of Review_count Common Table Expression (CTE):* Below are the steps that will be executed to generate review_count CTE.

1) Do sorting-based projection for each table to get product_category and review_id
2) Aggregate each table on product_category to get count of review_id
3) Union all tables
4) Sort combined table on product_category
5) Aggregate combined table

Sample calculations will be shown for the `apparel` table.

The number of pages in the apparel table is:

$$pages = 5874693\, records \times \frac{1\, page}{7\, records} = 839242$$

The number of pages in the apparel table after projection is:

$$pages = 5874693\, records \times \frac{1\, page}{240\, records} = 24478$$

The I/O cost of sorting-sorting based projection (including writing result) is:

$$\begin{aligned} I/O\ Cost &= 839242 + 24478 \\ &\quad + 2 \times 24478 \times (\lceil log_{999} \lceil 24478/1000 \rceil \rceil - 1) \\ &\quad + 2 \times 24478 \\ &= 912676 \end{aligned}$$

The cost of aggregating on product_category is the cost of a scan plus the cost of writing the results. There is 1 distinct product categories in the `apparel` table. The I/O cost is $24478 + 1 = 24479$.

Table VII shows the number of distinct categories per subtable. Only the multilingual table has multiple product categories.

TABLE VII
NUMBER OF DISTINCT CATEGORIES PER TABLE

| Table | Distinct Categories |
|---|---|
| apparel | 1 |
| appliances | 1 |
| automotive | 1 |
| multilingual | 38 |
| music | 1 |
| pc | 1 |
| wireless | 1 |

The above calculations were performed on all the tables to obtain the results in Table VIII.

When we union all the tables, we can fit all data in the buffer pages so the cost is reading each subtable, sorting them by product category, computing aggregations, and then writing result. There are 39 distinct product categories in the dataset so once aggregated we have 39 records which fit on one page. The I/O cost is

$$7 + 1 = 8$$

The total cost of the review_count CTE is

$$5741489 + 154019 + 8 = 5,895,516$$

TABLE VIII
I/O COSTS PER TABLE

| Table | Records | Pages | Pages After Proj. | Proj. I/O Cost | Agg. I/O Cost |
|---|---|---|---|---|---|
| apparel | 5874693 | 839242 | 24478 | 912676 | 24479 |
| appliances | 96792 | 13828 | 404 | 14232 | 405 |
| automotive | 3497760 | 499680 | 14574 | 543402 | 14575 |
| multilingual | 6882248 | 983179 | 28677 | 1069210 | 28678 |
| music | 4722502 | 674644 | 19678 | 733678 | 19679 |
| pc | 6902673 | 986097 | 28762 | 1072383 | 28763 |
| wireless | 8985135 | 1283591 | 37439 | 1395908 | 37440 |
| **Total** | | | | 5741489 | 154019 |

*I/O Cost of fstar_count Common Table Expression (CTE):* Below are the steps that will be executed to generate fstar_count CTE.

1) Do sorting-based projection for each table to get product_category and star_rating
2) Aggregate each table on product_category to get count of star_rating
3) Apply where clause
4) Union all tables
5) Sort combined table on product_category
6) Aggregate combined table

Sample calculations will be shown for the `apparel` table.

The number of pages in the apparel table is:

$$pages = 5874693\, records \times \frac{1\, page}{7\, records} = 839242$$

The number of pages in the apparel table after projection is:

$$pages = 5874693\, records \times \frac{1\, page}{292\, records} = 20119$$

The I/O cost of sorting-sorting based projection (including writing result) is:

$$\begin{aligned} I/O\ Cost &= 839242 + 20119 \\ &\quad + 2 \times 20119 \times (\lceil log_{999} \lceil 24478/1000 \rceil \rceil - 1) \\ &\quad + 2 \times 20119 \\ &= 899599 \end{aligned}$$

Applying where STAR_RATING = 5 is the cost of scanning the projection and writing the result. Using an index here slows down this operation because there are more applicable records than pages. A review can have a star rating of 1,2,3,4, or 5 stars.

$$I/O\ Cost = 20119 + \frac{1}{5} \times 20119 = 20119 + 4024 = 24143$$

Since the table was sorted in the sorting-based projection phase, we do not need to sort again.

The cost of aggregating on product_category is the cost of a scan plus the cost of writing the results. There is 1 distinct product category in the `apparel` table. The I/O cost is $4024 + 1 = 4025$.

The above calculations were performed on all the tables to obtain the results in Table IX.

TABLE IX
I/O COSTS PER TABLE

| Table | Pages | Pages After Proj. | Proj. I/O Cost | Where Cost | Agg. I/O Cost |
|---|---|---|---|---|---|
| apparel | 839242 | 20119 | 899599 | 24143 | 4025 |
| appliances | 13828 | 332 | 14160 | 399 | 68 |
| automotive | 499680 | 11979 | 535617 | 14375 | 2397 |
| multilingual | 983179 | 23570 | 1053889 | 28284 | 4715 |
| music | 674644 | 16173 | 723163 | 19408 | 3236 |
| pc | 986097 | 23640 | 1057017 | 28368 | 4729 |
| wireless | 1283591 | 30772 | 1375907 | 36927 | 6156 |
| **Total** | | | 5659352 | 151904 | 25326 |

As before, when we union all the tables, we can fit all data in buffer pages so the cost is reading each subtable, we can sort them by product category and compute aggregations while holding them in memory. There are 39 distinct product categories in the dataset so once each is aggregated we have 39 records which fit on one page. The I/O cost is

$$7 + 1 = 8$$

The total cost of the fstar_count CTE is

$$5659362 + 151904 + 25326 + 8 = 5,836,600$$

To join the two CTE's, the I/O cost is reading each CTE, joining them, aggregating, and outputting the results. Luckily, both tables fit in memory so the only I/O cost is reading each of them. The I/O cost is

$$1 + 1 = 2$$

The total cost is therefore

$$5,895,516 + 5,836,600 + 2 = 11,732,118$$

### A. I/O Cost of New Database Configuration

*Query for New Database Configuration:*

```
WITH review_count AS (
    SELECT product_category,
    count(review_id) as reviews
    FROM products p
    JOIN reviews r
    ON p.product_id = r.product_id
    GROUP BY product_category
),
fstar_count AS (
    SELECT products_category,
    count(star_rating) as five_stars
```

```
    FROM products p
    JOIN reviews r
    ON p.product_id = r.product_id
    JOIN review_details d
    ON r.review_id = d.review_id
    WHERE star_rating = 5
    GROUP BY product_category
)
SELECT review_count.product_category,
reviews, five_stars,
DEC(DEC(five_stars,9,2)/
DEC(reviews,9,2)*100,3,1)
AS five_stars_per_review FROM
review_count JOIN fstar_count
ON review_count.product_category
LIKE fstar_count.product_category
ORDER BY five_stars_per_review DESC
LIMIT 10;
```

Recall the new database configuration.

TABLE X
NEW DATABASE CONFIGURATION

| Table | Records | Size (Bytes) | Records/page | Pages |
|---|---|---|---|---|
| **Products** | 5257914 | 97 | 42 | 125189 |
| **Review_details** | 36961803 | 411 | 9 | 4106867 |
| **Reviews** | 36961803 | 31 | 132 | 280014 |

*I/O Cost of Review_count Common Table Expression (CTE):* Below are the steps that will be executed to generate review_count CTE.

1) Sort reviews table by product_id
2) Join products and reviews table on product_id
3) Perform sorting-based projection on product_category, review_id
4) Aggregate combined table (table is already sorted from projection)
5) Write results to memory

The cost to sort the reveiws table by product_id is

$$I/O\ Cost = 2 \times 280014 \times (1 + \lceil log_{999} \lceil 280014/1000 \rceil \rceil) = 1,120,056$$

The cost of joining the reviews table with the products table is the cost of scanning both of them since they are both now sorted by product_id. Joining them will not produce addition records since product_id is unique in the products table.

The number of pages of the joined table is:

$$pages = 36961803 rec. \times \frac{118\ bytes}{rec.} \times \frac{1\ page}{4096\ bytes} = 1087111$$

The cost of the join (including writing results to memory) is:

$$I/O\ Cost = 125189 + 280014 + 1087111 = 1492314$$

The I/O cost of getting the projection of product_category and review_id is calculated as below.

The number of pages of results is:

$$1087111 \times \frac{23}{118} = 211895$$

The I/O cost of sorting-based projection is

$$\begin{aligned} I/O\ Cost &= 1087111 + 2 \times 211895 \\ &\quad + 2 \times 211895 \times (\lceil log_{999} \lceil \frac{211895}{1000} \rceil \rceil - 1) + 211895 \\ &= 1722796 \end{aligned}$$

Since the table is already sorted by product_category, the cost of the aggregation is reading all pages and writing results. There are 39 distinct product categories so we will only have one page of results. The I/O cost is

$$211895 + 1 = 211896$$

Total I/O cost of review_count CTE is

$$1120056 + 1492314 + 1722796 + 211896 = 4547062$$

*I/O Cost of fstar_count Common Table Expression (CTE):* Below are the steps that will be executed to generate fstar_count CTE.

1) Perform a projection of review_details table on reveiw_id and star_rating
2) Scan table and only keep records where star_rating = 5
3) Join with reviews on review_id using sorting-based join since both are sorted by reveiw_id.
4) Sort result on product_id
5) Join result with products table on product_id
6) Perform sorting-based projection on product_category and star_rating
7) Aggregate on product_category
8) Write results to memory

I/O cost of performing projection of review_details on review_id and star_rating. Since review_id is unique we do not need to sort to eliminate duplicates.

$$\begin{aligned} I/O\ Cost &= 4106867 + 36961803\ rec. \times \frac{1\ page}{241\ rec.} \\ &= 4106867 + 153369 = 4260236 \end{aligned}$$

Now we will scan the table and only keep the records where star_rating = 5. The I/O cost is

$$\begin{aligned} I/O\ Cost &= 153369 + 153369 \times \frac{1}{5} \\ &= 153369 + 30674 = 184043 \end{aligned}$$

The cost of joining reveiws on review_id is the cost of scanning both tables and writing the result since they are already sorted by review_id. The result will have the same number of results as in the projected review table. The number of records is $36961803/5 = 7392361$

$$I/OCost = 280014 + 30674 + 7392361rec. \times \frac{1\,page}{117\,rec.}$$
$$= 310688 + 63183 = 373871$$

Next, we must sort this table on product_id so it can be joined with products table. The I/O cost is

$$I/O\ Cost = 2 \times 63183 \times (1 + \lceil log_{999} \lceil 63183/1000 \rceil \rceil)$$
$$= 252732$$

The I/O cost to join this table with the products table is the cost of scanning both of them. We also need to write the result to memory. There will be the same number of records (since product_id is unique in products table).

The number of resulting pages is

$$pages = 7392361 \times \frac{1}{\frac{4096}{114}} = 211211$$

I/O cost of join is

$$I/O\ Cost = 63183 + 125189 + 211211 = 399583$$

Now, we need to perform a projection on this table to get product_category and star_rating.

The pages of results will be

$$pages = 7392361 \times \frac{1}{\frac{4096}{14}} = 25317$$

The I/O cost is

$$I/O\ Cost = 211211 + 2 \times 25317$$
$$+ 2 \times 25317 \times (\lceil log_{999} \lceil \frac{25317}{1000} \rceil \rceil - 1) + 25317$$
$$= 287162$$

Aggregating on product_category is the cost of scanning the table. As before the results will fit on one page.

$$I/O\ Cost = 25317 + 1 = 25318$$

Total I/O cost of fstar_count CTE is

$$I/O\ Cost = 4260236 + 184043 + 373871 + 252732$$
$$+ 399583 + 287162 + 25318$$
$$= 5,782,945$$

To join the two CTE's, the I/O cost is reading each CTE, joining them, aggregating, and outputting the results. Luckily, both tables fit in memory so the only I/O cost is reading each of them. The I/O cost is

$$I/OCost = 1 + 1 = 2$$

The total cost is therefore

$$I/O\ Cost = 4,547,062 + 5,782,945 + 2 = 10,330,009$$

### B. Conclusion of Analysis

The difference in I/O cost was between the two database configurations for this query was

$$I/O\ Difference = 11,732,118 - 10,330,009 = 1,402,109$$

The new configuration results in a 12% improvement in I/O cost. The margin off improvement would depend on the query, the new database would likely require less I/O's. For instance, any queries that only need one of the table information would be substantially faster in the improved database configuration since no join operations would be needed.