Stage 3 Part 2

— Query 1 —

```
EXPLAIN ANALYZE
SELECT p.Plan_ID, p.Plan_Name, u.User_ID, u.Name, pc.Total_Sess
FROM Plan p
JOIN Users u ON p.User_ID = u.User_ID
JOIN (
    SELECT Plan_ID, COUNT(Session_ID) AS Total_Sessions
    FROM Plan_Contains
    GROUP BY Plan_ID
) pc ON p.Plan_ID = pc.Plan_ID
ORDER BY pc.Total_Sessions DESC
LIMIT 15;
```

Explain Analyze Output

```
| -> Limit: 15 row(s)  (cost=13463.00 rows=15) (actual time=8.2
    -> Nested loop inner join  (cost=13463.00 rows=5700) (actual
        -> Nested loop inner join  (cost=11468.00 rows=5700) (ac
            -> Sort: pc.Total_Sessions DESC  (cost=9473.00..947
                -> Table scan on pc  (cost=1717.51..1791.25 rows
                    -> Materialize  (cost=1717.50..1717.50 rows=
                        -> Group aggregate: count(Plan_Contains
                            -> Covering index scan on Plan_Conta
            -> Filter: (p.User_ID is not null)  (cost=0.25 rows=
                -> Single-row index lookup on p using PRIMARY (P
        -> Single-row index lookup on u using PRIMARY (User_ID=p
 |
```

Analysis

Costs: `13463.00`

Execution Time: `8.280ms..10.74ms`

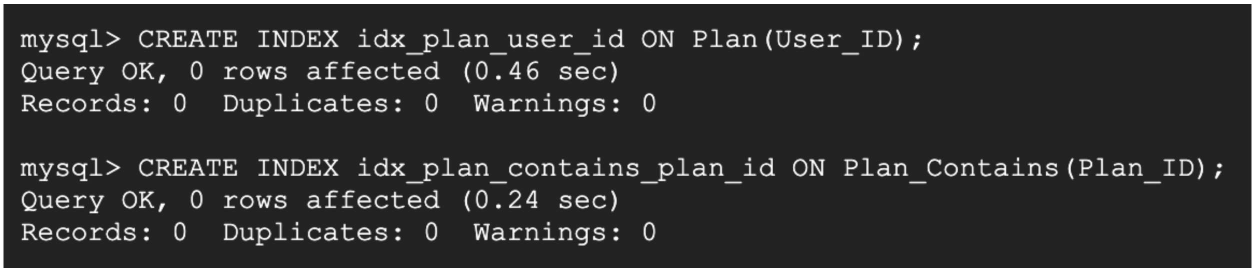
Total

- **High Cost in Sorting and Grouping**:
  - The GROUP BY in the subquery where it counts sessions per plan, is costing a lot per the output above
  - Sorting also contributed a lot to the execution time by processing with 0 indexing
- **Joins on User_ID and Plan_ID**:
  - We have Nested loop inner joins which is a indicator that indexing User_ID and Plan_ID in Plan and Plan_Contains respectively, should improve performance by avoiding full scans

Post-Analysis Additions

We are going to index these two columns.

```
CREATE INDEX idx_plan_user_id ON Plan(User_ID);
CREATE INDEX idx_plan_contains_plan_id ON Plan_Contains(Plan_ID
```



```
mysql> CREATE INDEX idx_plan_user_id ON Plan(User_ID);
Query OK, 0 rows affected (0.46 sec)
Records: 0  Duplicates: 0  Warnings: 0

mysql> CREATE INDEX idx_plan_contains_plan_id ON Plan_Contains(Plan_ID);
Query OK, 0 rows affected (0.24 sec)
Records: 0  Duplicates: 0  Warnings: 0
```

New *Explain Analyze* Output

```
| -> Limit: 15 row(s)  (cost=13463.00 rows=15) (actual time=4.3
    -> Nested loop inner join  (cost=13463.00 rows=5700) (actua
        -> Nested loop inner join  (cost=11468.00 rows=5700) (a
            -> Sort: pc.Total_Sessions DESC  (cost=9473.00..947
                -> Table scan on pc  (cost=1717.51..1791.25 rows
                    -> Materialize  (cost=1717.50..1717.50 rows=
                        -> Group aggregate: count(Plan_Contains
                            -> Covering index scan on Plan_Conta
            -> Filter: (p.User_ID is not null)  (cost=0.25 rows=
                -> Single-row index lookup on p using PRIMARY (P
        -> Single-row index lookup on u using PRIMARY (User_ID=p
 |
```

Post Analysis

- **Execution Time** :
  - **Before Indexing**: `8.280..10.741 ms`
  - **After Indexing**: `4.307..5.552 ms`

  The added indexes significantly reduced query time.

- **Optimized Grouping and Sorting**:
  - With idx_plan_contains_plan_id, the Group aggregate in the subquery now uses an indexed scan instead of a table scan.
  - **Cost Improvement**: Group aggregate on Plan_Contains.Session_ID improved from `0.308..2.258 ms` to `0.046..1.326 ms`.
  - This addition now allows faster sorting and grouping operations in Plan_Contains.

- **Improved Join Efficiency**:
  - The new indexes allows for faster lookups for User_ID in Plan and Plan_ID in Plan_Contains, this allows MySQL to use indexed lookups instead of table scans, resulting in a more efficient query.

— Query 2 —

```
EXPLAIN ANALYZE
WITH RankedExercises AS (
    SELECT e.Exercise_Name, e.Muscle_Group, e.Difficulty,
           ROW_NUMBER() OVER (PARTITION BY e.Muscle_Group ORDER
    FROM Exercises e
    JOIN Sets s ON e.Exercise_ID = s.Set_ID
    JOIN Users u ON s.User_ID = u.User_ID
    WHERE e.Difficulty > (
        SELECT AVG(e2.Difficulty)
        FROM Exercises e2
        WHERE e2.Muscle_Group = e.Muscle_Group
    )
)
SELECT Exercise_Name, Muscle_Group, Difficulty
FROM RankedExercises
WHERE rn = 1
ORDER BY Difficulty DESC
LIMIT 15;
```

OUTPUT

```
| -> Limit: 15 row(s)  (cost=0.10..0.10 rows=0) (actual time=23
    -> Sort: RankedExercises.Difficulty DESC, limit input to 15
        -> Index lookup on RankedExercises using <auto_key0> (r
            -> Materialize CTE RankedExercises  (cost=0.00..0.0
                -> Window aggregate: row_number() OVER (PARTITI
                    -> Sort: e.Muscle_Group, e.Difficulty DESC
                        -> Stream results  (cost=2337.10 rows=2
                            -> Nested loop inner join  (cost=23
                                -> Nested loop inner join  (cos
                                    -> Filter: (e.Difficulty >
                                        -> Table scan on e  (co
                                        -> Select #3 (subquery
                                            -> Aggregate: avg(e
                        -> Filter: (e2.M
                          -> Table sca
                  -> Filter: (s.User_ID is not
                    -> Single-row index look
                -> Single-row covering index loo
 |
```

Analysis

Costs: `2337.10`

Execution Time: `2331.449 ms .. 2331.450 ms`

- Window Aggregate and Sort Operations:
    - The row_number() window function has a high cost, along with all of the sorting operations of muscle group and difficulty
- **Subquery:**
    - The subquery that calculates the average difficulty for each Muscle_Group also has a high cost.
    - The subquery is executed for each row in the main query, resulting in a large number of rows and loops (2917 loops).
    - A table scan on Exercises (e2) occurs in the subquery, which further increased the cost.

Post Analysis Addition

```
CREATE INDEX idx_muscle_difficulty ON Exercises (Muscle_Group, D
CREATE INDEX idx_muscle_group ON Exercises (Muscle_Group);
```

New *Explain Analyze* Output

```
| -> Limit: 15 row(s)  (cost=0.10..0.10 rows=0) (actual time=387
    -> Sort: RankedExercises.Difficulty DESC, limit input to 15
        -> Index lookup on RankedExercises using <auto_key0> (r
            -> Materialize CTE RankedExercises  (cost=0.00..0.0
                -> Window aggregate: row_number() OVER (PARTITI
                    -> Sort: e.Muscle_Group, e.Difficulty DESC
                        -> Stream results  (cost=2337.10 rows=2
                            -> Nested loop inner join  (cost=233
                                -> Nested loop inner join  (cos
                                    -> Filter: (e.Difficulty > (
                                        -> Table scan on e  (co
                                        -> Select #3 (subquery
                                            -> Aggregate: avg(e
                                                -> Covering inde
                                    -> Filter: (s.User_ID is no
                                        -> Single-row index loo
                                -> Single-row covering index loo
 |
```

Post Analysis

- **Reduced Cost in the Window Aggregate and Sort Operations**
  - The addition of idx_muscle_difficulty and idx_muscle_group helps in efficiently partitioning and sorting:
    - **Window aggregate:** `2,300 ms to 385.601..386.261` `ms.`
    - **Sort on e.Muscle_Group and e.Difficulty**: 385.590..385.732 ms.
  - The index on (Muscle_Group, Difficulty) optimized these operations by allowing MySQL to efficiently sort and partition the data based on indexed values.
- **Improved Efficiency in Select #3**
  - The subquery now uses a covering index (idx_muscle_difficulty) to look up Muscle_Group and Difficulty, reducing the subquery cost:
    - Now **cost=26.06 and rows=172** with actual time=`0.017..0.104 ms` per loop, significantly lower than before.

— Query 3 —

```
EXPLAIN ANALYZE
SELECT p.Plan_ID, p.Plan_Name, COUNT(DISTINCT e.Muscle_Group) A
FROM Plan p
JOIN Plan_Contains pc ON p.Plan_ID = pc.Plan_ID
JOIN Session s ON pc.Session_ID = s.Session_ID
JOIN Session_Contains sc ON s.Session_ID = sc.Session_ID
JOIN Sets se ON sc.Set_ID = se.Set_ID
JOIN Set_Contains sec ON se.Set_ID = sec.Set_ID
JOIN Exercises e ON sec.Exercise_ID = e.Exercise_ID
GROUP BY p.Plan_ID, p.Plan_Name
ORDER BY Muscle_Groups_Targeted DESC, p.Plan_ID DESC
LIMIT 15;
```

OUTPUT

```
| -> Limit: 15 row(s)  (actual time=298.304..298.306 rows=15 lo
    -> Sort: Muscle_Groups_Targeted DESC, p.Plan_ID DESC, limit
        -> Stream results  (cost=49824.74 rows=53023) (actual ti
            -> Group aggregate: count(distinct e.Muscle_Group)
                -> Nested loop inner join  (cost=44522.42 rows=
                    -> Nested loop inner join  (cost=25964.28 r
                        -> Nested loop inner join  (cost=16347.
                            -> Nested loop inner join  (cost=10
                                -> Nested loop inner join  (cos
                                    -> Nested loop inner join
                                        -> Sort: p.Plan_ID, p.P
                                            -> Table scan on p
                                        -> Covering index looku
                                    -> Single-row covering inde
                                -> Covering index lookup on sc
                            -> Single-row covering index lookup
                        -> Covering index lookup on sec using P
                    -> Single-row index lookup on e using PRIMA
 |
```

Analysis

- **Total Cost:** `49824.74`
- **Execution Time:** `298.304 ms`
- The query performs multiple nested loop joins across, each contributing to high costs.
- The GROUP BY and COUNT(DISTINCT e.Muscle_Group) on Exercises would benefit from a sort and aggregate operation

To improve efficiency, let's index columns frequently used in joins

- **Plan_Contains.Plan_ID** (Plan)
- **Session_Contains.Session_ID** (Session and Plan_Contains)
- **Set_Contains.Set_ID** (Sets and Session_Contains)

Post Analysis Addition

```
CREATE INDEX idx_plan_contains_plan_id ON Plan_Contains(Plan_ID
CREATE INDEX idx_session_contains_session_id ON Session_Contain
CREATE INDEX idx_set_contains_set_id ON Set_Contains(Set_ID);
```

New *Explain Analyze* Output

```
| -> Limit: 15 row(s)  (actual time=211.554..211.556 rows=15 lo
    -> Sort: Muscle_Groups_Targeted DESC, p.Plan_ID DESC, limit
        -> Stream results  (cost=46641.21 rows=53023) (actual t
            -> Group aggregate: count(distinct e.Muscle_Group)
                -> Nested loop inner join  (cost=41338.88 rows=
                    -> Nested loop inner join  (cost=22780.75 r
                        -> Nested loop inner join  (cost=13163.
                            -> Nested loop inner join  (cost=68
                                -> Nested loop inner join  (cos
                                    -> Nested loop inner join
                                        -> Sort: p.Plan_ID, p.Pl
                                            -> Table scan on p
                                        -> Covering index looku
                                    -> Single-row covering inde
                                -> Covering index lookup on sc
                            -> Single-row covering index lookup
                        -> Covering index lookup on sec using P
                    -> Single-row index lookup on e using PRIMA
 |
```

Post Analysis

COST: Lowers by about 6.4%,

- **Before Indexing:** `49824.74`
- **After Indexing:** `46641.21`

TIME: Lowers by about 29%,

- **Before Indexing:** `298.304 ms`
- **After Indexing:** `211.554 ms`

Summary:

Session_Contains.Session_ID and Set_Contains.Set_ID lookups are now using indexes, reducing time from row-by-row access to direct lookups.

- Session_Contains.Session_ID decreased from `0.018` per lookup to `0.003`, and Set_Contains.Set_ID from `0.006` to `0.002`.

— Query 4 —

```
EXPLAIN ANALYZE
WITH ExerciseFrequency AS (
    SELECT e.Exercise_ID, e.Exercise_Name, e.Muscle_Group, COUN
    FROM Exercises e
    JOIN Set_Contains sc ON e.Exercise_ID = sc.Exercise_ID
    GROUP BY e.Exercise_ID, e.Exercise_Name, e.Muscle_Group
), RankedExercises AS (
    SELECT Exercise_Name, Muscle_Group,
           ROW_NUMBER() OVER (PARTITION BY Muscle_Group ORDER B
    FROM ExerciseFrequency
)
SELECT Exercise_Name, Muscle_Group
FROM RankedExercises
WHERE rn = 1
ORDER BY Muscle_Group
LIMIT 15;
```

OUTPUT

```
| -> Limit: 15 row(s)  (cost=0.10..0.10 rows=0) (actual time=20
    -> Sort: RankedExercises.Muscle_Group, limit input to 15 ro
        -> Index lookup on RankedExercises using <auto_key0> (r
            -> Materialize CTE RankedExercises  (cost=0.00..0.0
                -> Window aggregate: row_number() OVER (PARTITI
                    -> Sort: ExerciseFrequency.Muscle_Group, Exe
                        -> Table scan on ExerciseFrequency  (co
                            -> Materialize CTE ExerciseFrequenc
                                -> Table scan on <temporary>  (
                                    -> Aggregate using temporar
                                        -> Nested loop inner jo
                                            -> Table scan on e
                                            -> Covering index l
 |
```

Analysis

- **Total Cost:** `5645.34`
- **Execution Time:** `209.003 ms` for the limit operation

- Joins on Set_Contains.Exercise_ID and Exercises.Exercise_ID are costly

We should index the columns that are used in JOINS and SORTING operations

- **Set_Contains.Exercise_ID**: Used to join with Exercises on Exercise_ID.
- **Exercises.Muscle_Group** and **Exercises.Exercise_ID**: Used in grouping and ordering operations in the ExerciseFrequency

Post Analysis Addition

```
CREATE INDEX idx_set_contains_exercise_id ON Set_Contains(Exerci
CREATE INDEX idx_exercises_muscle_group_exercise_id ON Exercises
```

New *Explain Analyze* Output

```
| -> Limit: 15 row(s)  (cost=0.10..0.10 rows=0) (actual time=10
    -> Sort: RankedExercises.Muscle_Group, limit input to 15 ro
        -> Index lookup on RankedExercises using <auto_key0> (r
            -> Materialize CTE RankedExercises  (cost=0.00..0.0
                -> Window aggregate: row_number() OVER (PARTITI
                    -> Sort: ExerciseFrequency.Muscle_Group, Exe
                        -> Table scan on ExerciseFrequency  (co
                            -> Materialize CTE ExerciseFrequency
                                -> Table scan on <temporary>  (a
                                    -> Aggregate using temporary
                                        -> Nested loop inner jo
                                            -> Table scan on e
                                            -> Covering index l
 |
```

Post Analysis

Total Cost: Cost went up in indexing, I believe this is due to the increased operations that are involved in cost calculations, meaning indexing hurt cost more than it helped.

- **Before Indexing:** `5645.34`

- **After Indexing:** `7845.56`

Execution Time: Execution time has decreased by approximately 51%, showing greater retrieval speeds to the indexing

- **Before Indexing:** `209.003 ms`
- **After Indexing:** `102.212 ms`

Nested Joins and Table Access:

- Set_Contains.Exercise_ID lookup decreased to `0.005..0.007 ms` from `0.015..0.019 ms`.

Summary: Although cost time went up, our execution time went down dramatically