Stage 3 Part 2

— Query 1 —

```
mysql> EXPLAIN ANALYZE
    -> SELECT p.Plan_ID, p.Plan_Name, u.User_ID, u.Name, pc.Total_Sessions
    -> 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;
```

```
| -> Limit: 15 row(s)  (cost=13463.00 rows=15) (actual time=9.929..12.771 rows=15 loops=1)
    -> Nested loop inner join  (cost=13463.00 rows=5700) (actual time=9.912..12.751 rows=15 loops=1)
        -> Nested loop inner join  (cost=11468.00 rows=5700) (actual time=9.790..12.033 rows=15 loops=1)
            -> Sort: pc.Total_Sessions DESC  (cost=9473.00..9473.00 rows=5700) (actual time=9.637..9.642 rows=15 loops=1)
                -> Table scan on pc  (cost=1717.51..1791.25 rows=5700) (actual time=6.716..6.859 rows=1419 loops=1)
                    -> Materialize  (cost=1717.50..1717.50 rows=5700) (actual time=6.713..6.713 rows=1419 loops=1)
                        -> Group aggregate: count(Plan_Contains.Session_ID)  (cost=1147.50 rows=5700) (actual time=0.316..6.157 rows=1419 loops=1)
                            -> Covering index scan on Plan_Contains using PRIMARY  (cost=577.50 rows=5700) (actual time=0.282..5.524 rows=5700 loops=1)
            -> Filter: (p.User_ID is not null)  (cost=0.25 rows=1) (actual time=0.156..0.157 rows=1 loops=15)
                -> Single-row index lookup on p using PRIMARY (Plan_ID=pc.Plan_ID)  (cost=0.25 rows=1) (actual time=0.156..0.156 rows=1 loops=15)
        -> Single-row index lookup on u using PRIMARY (User_ID=p.User_ID)  (cost=0.25 rows=1) (actual time=0.047..0.047 rows=1 loops=15)
 |
```

```
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
```

```
| -> Limit: 15 row(s)  (cost=13463.00 rows=15) (actual time=1.918..1.972 rows=15 loops=1)
    -> Nested loop inner join  (cost=13463.00 rows=5700) (actual time=1.917..1.970 rows=15 loops=1)
        -> Nested loop inner join  (cost=11468.00 rows=5700) (actual time=1.910..1.932 rows=15 loops=1)
            -> Sort: pc.Total_Sessions DESC  (cost=9473.00..9473.00 rows=5700) (actual time=1.895..1.896 rows=15 loops=1)
                -> Table scan on pc  (cost=1717.51..1791.25 rows=5700) (actual time=1.534..1.669 rows=1419 loops=1)
                    -> Materialize  (cost=1717.50..1717.50 rows=5700) (actual time=1.533..1.533 rows=1419 loops=1)
                        -> Group aggregate: count(Plan_Contains.Session_ID)  (cost=1147.50 rows=5700) (actual time=0.042..1.410 rows=1419 loops=1)
                            -> Covering index scan on Plan_Contains using idx_plan_contains_plan_id  (cost=577.50 rows=5700) (actual time=0.037..1.030 rows=5700 loops=1)
            -> Filter: (p.User_ID is not null)  (cost=0.25 rows=1) (actual time=0.002..0.002 rows=1 loops=15)
                -> Single-row index lookup on p using PRIMARY (Plan_ID=pc.Plan_ID)  (cost=0.25 rows=1) (actual time=0.002..0.002 rows=1 loops=15)
        -> Single-row index lookup on u using PRIMARY (User_ID=p.User_ID)  (cost=0.25 rows=1) (actual time=0.002..0.002 rows=1 loops=15)
 |
```

```
mysql> CREATE INDEX idx_plan_contains_session_id ON Plan_Contains(Session_ID);
Query OK, 0 rows affected (0.25 sec)
Records: 0  Duplicates: 0  Warnings: 0
```

```
| -> Limit: 15 row(s)  (cost=13463.00 rows=15) (actual time=5.541..6.831 rows=15 loops=1)
    -> Nested loop inner join  (cost=13463.00 rows=5700) (actual time=5.522..6.811 rows=15 loops=1)
        -> Nested loop inner join  (cost=11468.00 rows=5700) (actual time=5.384..5.595 rows=15 loops=1)
            -> Sort: pc.Total_Sessions DESC  (cost=9473.00..9473.00 rows=5700) (actual time=5.186..5.187 rows=15 loops=1)
                -> Table scan on pc  (cost=1717.51..1791.25 rows=5700) (actual time=3.713..3.846 rows=1419 loops=1)
                    -> Materialize  (cost=1717.50..1717.50 rows=5700) (actual time=3.710..3.710 rows=1419 loops=1)
                        -> Group aggregate: count(Plan_Contains.Session_ID)  (cost=1147.50 rows=5700) (actual time=0.101..3.047 rows=1419 loops=1)
                            -> Covering index scan on Plan_Contains using idx_plan_contains_plan_id  (cost=577.50 rows=5700) (actual time=0.085..2.458 rows=5700 loops=1)
            -> Filter: (p.User_ID is not null)  (cost=0.25 rows=1) (actual time=0.023..0.024 rows=1 loops=15)
                -> Single-row index lookup on p using PRIMARY (Plan_ID=pc.Plan_ID)  (cost=0.25 rows=1) (actual time=0.023..0.023 rows=1 loops=15)
        -> Single-row index lookup on u using PRIMARY (User_ID=p.User_ID)  (cost=0.25 rows=1) (actual time=0.081..0.081 rows=1 loops=15)
 |
```

```
mysql> CREATE INDEX idx_plan_contains_plan_id_session_id_composite
    ->     ON Plan_Contains(Plan_ID, Session_ID);
Query OK, 0 rows affected (0.28 sec)
Records: 0  Duplicates: 0  Warnings: 0
```

```
| -> Limit: 15 row(s)  (cost=13463.00 rows=15) (actual time=2.769..3.371 rows=15 loops=1)
    -> Nested loop inner join  (cost=13463.00 rows=5700) (actual time=2.758..3.359 rows=15 loops=1)
        -> Nested loop inner join  (cost=11468.00 rows=5700) (actual time=2.702..2.854 rows=15 loops=1)
            -> Sort: pc.Total_Sessions DESC  (cost=9473.00..9473.00 rows=5700) (actual time=2.618..2.619 rows=15 loops=1)
                -> Table scan on pc  (cost=1717.51..1791.25 rows=5700) (actual time=2.097..2.244 rows=1419 loops=1)
                    -> Materialize  (cost=1717.50..1717.50 rows=5700) (actual time=2.095..2.095 rows=1419 loops=1)
                        -> Group aggregate: count(Plan_Contains.Session_ID)  (cost=1147.50 rows=5700) (actual time=0.166..1.778 rows=1419 loops=1)
                            -> Covering index scan on Plan_Contains using idx_plan_contains_plan_id  (cost=577.50 rows=5700) (actual time=0.096..1.359 rows=5700 loops=1)
            -> Filter: (p.User_ID is not null)  (cost=0.25 rows=1) (actual time=0.014..0.014 rows=1 loops=15)
                -> Single-row index lookup on p using PRIMARY (Plan_ID=pc.Plan_ID)  (cost=0.25 rows=1) (actual time=0.014..0.014 rows=1 loops=15)
        -> Single-row index lookup on u using PRIMARY (User_ID=p.User_ID)  (cost=0.25 rows=1) (actual time=0.033..0.033 rows=1 loops=15)
 |
```

1. Before Any Indexes

EXPLAIN ANALYZE Output:

- **Total cost:** `13463.00`
- **Actual time**: 9.929–12.771 ms for 15 rows.
- **Scan method**: Full table scan on `Plan_Contains` for the aggregation (COUNT) operation.
- **Joins**: Nested loop join between `Plan` and `Users`, and between `Plan` and the subquery.
- **Sorting**: Sort by `Total_Sessions` in descending order.

2. Adding Index on `Plan_Contains(Plan_ID)`

EXPLAIN ANALYZE Output:

- **Total cost:** `13463.00`
- **Actual time**: 1.918–1.972 ms for 15 rows (significantly improved).
- **Scan method**: A **covering index scan** is used on `Plan_Contains` using the `idx_plan_contains_plan_id` index.
- **Join performance**: The join between `Plan` and `Plan_Contains` is much faster because the query no longer has to perform a full table scan; instead, it uses the index for faster access to `Plan_ID`.

- **Performance improvement**: The actual time improved drastically, from ~9.9 ms to ~1.9 ms, which is a significant performance gain.

3. Adding Index on `Plan_Contains(Session_ID)`

EXPLAIN ANALYZE Output:

- **Total cost**: `13463.00`
- **Actual time**: 5.541–6.831 ms for 15 rows (worse than the previous case).
- **Scan method**: A **covering index scan** on `Plan_Contains` using `idx_plan_contains_plan_id`. The `Session_ID` index doesn't seem to provide any additional benefit here because the primary benefit from this index would be for speeding up the aggregation (`COUNT(Session_ID)`).
- **Performance degradation**: The query's time increased significantly (from 1.9 ms to around 5.5–6.8 ms). This could be due to the overhead of using two indexes (`idx_plan_contains_plan_id` and `idx_plan_contains_session_id`) when the `Plan_ID` index alone is sufficient.

4. Adding Composite Index on `Plan_Contains(Plan_ID, Session_ID)`

EXPLAIN ANALYZE Output:

- **Total cost**: `13463.00`
- **Actual time**: 2.769–3.371 ms for 15 rows (slight improvement compared to the previous test).
- **Scan method**: A **covering index scan** on `Plan_Contains` using `idx_plan_contains_plan_id`. The composite index did not seem to provide a significant improvement for this particular query.
- **Performance change**: The actual time slightly decreased (compared to the previous query with just `Session_ID`), but the change is marginal. This is likely because the query already benefits from the `Plan_ID` index, and adding the composite index didn't significantly impact the performance of the `COUNT()` operation.

Final Index Design:

Based on this analysis, the best index configuration is:

1. **Index on** `Plan_Contains(Plan_ID)`.

Reasoning:

- The `Plan_ID` index is sufficient to optimize the joins and aggregation in the subquery.
- The **composite index** or the **index on** `Session_ID` did not result in a significant performance gain for this specific query and, in fact, caused performance degradation when both were used.

Thus, `Plan_Contains(Plan_ID)` is the key index for optimizing this query, and adding extra indexes beyond this does not seem to improve performance significantly.

— Query 2 —

```
mysql> EXPLAIN ANALYZE
    -> WITH RankedExercises AS (
    ->     SELECT e.Exercise_Name, e.Muscle_Group, e.Difficulty,
    ->            ROW_NUMBER() OVER (PARTITION BY e.Muscle_Group ORDER BY e.Difficulty DESC) AS rn
    ->     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;
```

```
| -> Limit: 15 row(s)  (cost=0.10..0.10 rows=0) (actual time=678.550..678.552 rows=15 loops=1)
    -> Sort: RankedExercises.Difficulty DESC, limit input to 15 row(s) per chunk  (cost=0.10..0.10 rows=0) (actual time=678.539..678.540 rows=15 loops=1)
        -> Index lookup on RankedExercises using <auto_key0> (rn=1)  (actual time=678.448..678.455 rows=17 loops=1)
            -> Materialize CTE RankedExercises  (cost=0.00..0.00 rows=0) (actual time=678.443..678.443 rows=1529 loops=1)
                -> Window aggregate: row_number() OVER (PARTITION BY e.Muscle_Group ORDER BY e.Difficulty desc )   (actual time=667.446..668.696 rows=1529 loops=1)
                    -> Sort: e.Muscle_Group, e.Difficulty DESC  (actual time=667.433..667.692 rows=1529 loops=1)
                        -> Stream results  (cost=4478.82 rows=2917) (actual time=174.278..656.844 rows=1529 loops=1)
                            -> Nested loop inner join  (cost=4478.82 rows=2917) (actual time=174.267..655.937 rows=1529 loops=1)
                                -> Nested loop inner join  (cost=3457.87 rows=2917) (actual time=173.973..650.407 rows=1529 loops=1)
                                    -> Filter: (e.Difficulty > (select #3))  (cost=300.05 rows=2917) (actual time=103.199..548.516 rows=1529 loops=1)
                                        -> Table scan on e  (cost=300.05 rows=2917) (actual time=53.672..107.773 rows=2917 loops=1)
                                        -> Select #3 (subquery in condition; dependent)
                                            -> Aggregate: avg(e2.Difficulty)  (cost=44.79 rows=1) (actual time=0.150..0.150 rows=1 loops=2917)
                                                -> Covering index lookup on e2 using idx_muscle_difficulty (Muscle_Group=e.Muscle_Group)  (cost=27.64 rows=172) (actual time=0.035..0.125 rows=396 loops=2
917)

                                    -> Filter: (s.User_ID is not null)  (cost=0.98 rows=1) (actual time=0.066..0.066 rows=1 loops=1529)
                                        -> Single-row index lookup on s using PRIMARY (Set_ID=e.Exercise_ID)  (cost=0.98 rows=1) (actual time=0.066..0.066 rows=1 loops=1529)
                                -> Single-row covering index lookup on u using PRIMARY (User_ID=s.User_ID)  (cost=0.25 rows=1) (actual time=0.003..0.003 rows=1 loops=1529)
 |
```

```
mysql> CREATE INDEX idx_exercises_muscle_group ON Exercises(Muscle_Group);
Query OK, 0 rows affected, 1 warning (0.37 sec)
Records: 0  Duplicates: 0  Warnings: 1
```

```
| -> Limit: 15 row(s)  (cost=0.10..0.10 rows=0) (actual time=381.375..381.377 rows=15 loops=1)
    -> Sort: RankedExercises.Difficulty DESC, limit input to 15 row(s) per chunk  (cost=0.10..0.10 rows=0) (actual time=381.372..381.373 rows=15 loops=1)
        -> Index lookup on RankedExercises using <auto_key0> (rn=1)  (actual time=381.343..381.347 rows=17 loops=1)
            -> Materialize CTE RankedExercises  (cost=0.00..0.00 rows=0) (actual time=381.341..381.341 rows=1529 loops=1)
                -> Window aggregate: row_number() OVER (PARTITION BY e.Muscle_Group ORDER BY e.Difficulty desc )   (actual time=379.828..380.478 rows=1529 loops=1)
                    -> Sort: e.Muscle_Group, e.Difficulty DESC  (actual time=379.817..379.923 rows=1529 loops=1)
                        -> Stream results  (cost=4016.07 rows=2917) (actual time=6.716..376.293 rows=1529 loops=1)
                            -> Nested loop inner join  (cost=4016.07 rows=2917) (actual time=6.711..375.628 rows=1529 loops=1)
                                -> Nested loop inner join  (cost=2995.12 rows=2917) (actual time=6.703..374.226 rows=1529 loops=1)
                                    -> Filter: (e.Difficulty > (select #3))  (cost=295.20 rows=2917) (actual time=6.685..371.638 rows=1529 loops=1)
                                        -> Table scan on e  (cost=295.20 rows=2917) (actual time=4.223..5.640 rows=2917 loops=1)
                                        -> Select #3 (subquery in condition; dependent)
                                            -> Aggregate: avg(e2.Difficulty)  (cost=43.22 rows=1) (actual time=0.125..0.125 rows=1 loops=2917)
                                                -> Covering index lookup on e2 using idx_muscle_difficulty (Muscle_Group=e.Muscle_Group)  (cost=26.06 rows=172) (actual time=0.017..0.100 rows=396 loops=2
917)
                                    -> Filter: (s.User_ID is not null)  (cost=0.83 rows=1) (actual time=0.001..0.002 rows=1 loops=1529)
                                        -> Single-row index lookup on s using PRIMARY (Set_ID=e.Exercise_ID)  (cost=0.83 rows=1) (actual time=0.001..0.001 rows=1 loops=1529)
                                -> Single-row covering index lookup on u using PRIMARY (User_ID=s.User_ID)  (cost=0.25 rows=1) (actual time=0.001..0.001 rows=1 loops=1529)
 |
```

```
mysql> CREATE INDEX idx_sets_user_id ON Sets(User_ID);
Query OK, 0 rows affected (0.59 sec)
Records: 0  Duplicates: 0  Warnings: 0
```

```
| -> Limit: 15 row(s)  (cost=0.10..0.10 rows=0) (actual time=389.189..389.191 rows=15 loops=1)
    -> Sort: RankedExercises.Difficulty DESC, limit input to 15 row(s) per chunk  (cost=0.10..0.10 rows=0) (actual time=389.178..389.179 rows=15 loops=1)
        -> Index lookup on RankedExercises using <auto_key0> (rn=1)  (actual time=389.139..389.145 rows=17 loops=1)
            -> Materialize CTE RankedExercises  (cost=0.00..0.00 rows=0) (actual time=389.135..389.135 rows=1529 loops=1)
                -> Window aggregate: row_number() OVER (PARTITION BY e.Muscle_Group ORDER BY e.Difficulty desc )   (actual time=382.975..383.961 rows=1529 loops=1)
                    -> Sort: e.Muscle_Group, e.Difficulty DESC  (actual time=382.965..383.191 rows=1529 loops=1)
                        -> Stream results  (cost=2337.10 rows=2917) (actual time=2.038..380.376 rows=1529 loops=1)
                            -> Nested loop inner join  (cost=2337.10 rows=2917) (actual time=2.037..379.660 rows=1529 loops=1)
                                -> Nested loop inner join  (cost=1316.15 rows=2917) (actual time=1.991..375.524 rows=1529 loops=1)
                                    -> Filter: (e.Difficulty > (select #3))  (cost=295.20 rows=2917) (actual time=1.944..371.608 rows=1529 loops=1)
                                        -> Table scan on e  (cost=295.20 rows=2917) (actual time=0.607..3.578 rows=2917 loops=1)
                                        -> Select #3 (subquery in condition; dependent)
                                            -> Aggregate: avg(e2.Difficulty)  (cost=43.22 rows=1) (actual time=0.125..0.125 rows=1 loops=2917)
                                                -> Covering index lookup on e2 using idx_muscle_difficulty (Muscle_Group=e.Muscle_Group)  (cost=26.06 rows=172) (actual time=0.016..0.101 rows=396 loops=2
917)
                                    -> Filter: (s.User_ID is not null)  (cost=0.25 rows=1) (actual time=0.002..0.002 rows=1 loops=1529)
                                        -> Single-row index lookup on s using PRIMARY (Set_ID=e.Exercise_ID)  (cost=0.25 rows=1) (actual time=0.002..0.002 rows=1 loops=1529)
                                -> Single-row covering index lookup on u using PRIMARY (User_ID=s.User_ID)  (cost=0.25 rows=1) (actual time=0.002..0.003 rows=1 loops=1529)
 |
```

```
mysql> CREATE INDEX idx_exercises_difficulty ON Exercises(Difficulty);
Query OK, 0 rows affected (0.18 sec)
Records: 0  Duplicates: 0  Warnings: 0
```

```
| -> Limit: 15 row(s)  (cost=0.10..0.10 rows=0) (actual time=425.389..425.391 rows=15 loops=1)
    -> Sort: RankedExercises.Difficulty DESC, limit input to 15 row(s) per chunk  (cost=0.10..0.10 rows=0) (actual time=425.388..425.389 rows=15 loops=1)
        -> Index lookup on RankedExercises using <auto_key0> (rn=1)  (actual time=425.363..425.367 rows=17 loops=1)
            -> Materialize CTE RankedExercises  (cost=0.00..0.00 rows=0) (actual time=425.360..425.360 rows=1529 loops=1)
                -> Window aggregate: row_number() OVER (PARTITION BY e.Muscle_Group ORDER BY e.Difficulty desc )   (actual time=423.907..424.517 rows=1529 loops=1)
                    -> Sort: e.Muscle_Group, e.Difficulty DESC  (actual time=423.889..424.002 rows=1529 loops=1)
                        -> Stream results  (cost=2337.10 rows=2917) (actual time=0.949..422.796 rows=1529 loops=1)
                            -> Nested loop inner join  (cost=2337.10 rows=2917) (actual time=0.948..421.886 rows=1529 loops=1)
                                -> Nested loop inner join  (cost=1316.15 rows=2917) (actual time=0.940..419.858 rows=1529 loops=1)
                                    -> Filter: (e.Difficulty > (select #3))  (cost=295.20 rows=2917) (actual time=0.927..415.842 rows=1529 loops=1)
                                        -> Table scan on e  (cost=295.20 rows=2917) (actual time=0.094..2.405 rows=2917 loops=1)
                                        -> Select #3 (subquery in condition; dependent)
                                            -> Aggregate: avg(e2.Difficulty)  (cost=43.22 rows=1) (actual time=0.141..0.141 rows=1 loops=2917)
                                                -> Covering index lookup on e2 using idx_muscle_difficulty (Muscle_Group=e.Muscle_Group)  (cost=26.06 rows=172) (actual time=0.018..0.114 rows=396 loops=2
917)
                                    -> Filter: (s.User_ID is not null)  (cost=0.25 rows=1) (actual time=0.002..0.002 rows=1 loops=1529)
                                        -> Single-row index lookup on s using PRIMARY (Set_ID=e.Exercise_ID)  (cost=0.25 rows=1) (actual time=0.002..0.002 rows=1 loops=1529)
                                -> Single-row covering index lookup on u using PRIMARY (User_ID=s.User_ID)  (cost=0.25 rows=1) (actual time=0.001..0.001 rows=1 loops=1529)
 |
```

Original Query Performance (No Indexes)

The query execution without any indexes initially revealed:

- **Cost**: The total cost for the query execution was high due to the table scan on `Exercises`. The cost for filtering on `e.Difficulty` was 295.20, indicating a large number of rows were being scanned.
- **Time**: The execution time was **678 ms**, primarily due to the table scan on `Exercises`, nested loop joins, and the window function (`ROW_NUMBER()`).

Indexing Configuration 1: Index on `Exercises(Muscle_Group)`

The first index created was on the `Muscle_Group` column in the `Exercises` table. This index optimizes both the filtering in the subquery (`WHERE e2.Muscle_Group = e.Muscle_Group`) and the partitioning in the window function (`ROW_NUMBER() OVER (PARTITION BY e.Muscle_Group ORDER BY e.Difficulty DESC)`).

Performance After Indexing `Exercises(Muscle_Group)`:

- **Cost**: The cost dropped from **4478.82** to **4016.07**.
- **Time**: The query execution time significantly improved to **381 ms**, a reduction of nearly 40% compared to the initial **678 ms**.

Key Observations:

- **Window Function Optimization**: The `ROW_NUMBER()` function benefits from the index because it partitions by `Muscle_Group`, which allows for quicker processing of the results.
- **Subquery Optimization**: The subquery calculating the average `Difficulty` also benefits from the `Muscle_Group` index, improving lookup times.
- **Join Performance**: The overall join operations, particularly between `Exercises` and `Sets`, become more efficient due to better filtering on `Muscle_Group`.

Indexing Configuration 2: Index on `Sets(User_ID)`

Next, an index was created on `User_ID` in the `Sets` table, which is used for the join condition between `Sets` and `Users` (`JOIN Users u ON s.User_ID = u.User_ID`).

Performance After Indexing `Sets(User_ID)`:

- **Cost**: The cost further dropped to **2337.10**.
- **Time**: The execution time improved slightly to **389 ms**.

Key Observations:

- The indexing on `Sets(User_ID)` optimizes the join operation between `Sets` and `Users`. However, the improvement was relatively small because the query's main bottleneck was elsewhere (in the `Exercises` table).

- While the performance improved, it's clear that the larger benefit came from optimizing the filtering and partitioning operations, particularly related to `Muscle_Group`.

Indexing Configuration 3: Index on `Exercises(Difficulty)`

Finally, an index was added on the `Difficulty` column in the `Exercises` table. This index could benefit the filtering condition `WHERE e.Difficulty > (...)` and the sorting by `Difficulty DESC` in the final result.

Performance After Indexing `Exercises(Difficulty)`:

- **Cost**: The total cost remained at **2337.10**, similar to the previous configuration.

Key Observations:

- The added index on `Difficulty` didn’t provide significant performance improvement. In fact, the execution time slightly worsened.
- This suggests that while the sorting operation could benefit from the `Difficulty` index, the query's filtering and sorting operations were already efficient without this index. The overhead introduced by maintaining the index likely outweighed any performance gains from sorting.

Final Index Design

After evaluating the performance impacts, the **index on `Exercises(Muscle_Group)`** (`idx_exercises_muscle_group`) is the most effective.

Reasoning

1. **Partitioning in Window Function**: The `Muscle_Group` index significantly optimizes the window function's partitioning (`ROW_NUMBER() OVER (PARTITION BY e.Muscle_Group ORDER BY e.Difficulty DESC)`), which is one of the main computational steps in the query.
2. **Subquery Optimization**: The `Muscle_Group` index speeds up the subquery that calculates the average difficulty for each muscle group.
3. **Improved Join Efficiency**: The query involves a join between `Exercises` and `Sets` based on `Exercise_ID`, and filtering on `Muscle_Group` helps optimize this join.

— Query 3 —

```
EXPLAIN ANALYZE
SELECT p.Plan_ID, p.Plan_Name, COUNT(DISTINCT e.Muscle_Group) AS
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=891.021..891.024 rows=15 loo
    -> Sort: Muscle_Groups_Targeted DESC, p.Plan_ID DESC, limit
        -> Stream results  (cost=66795.46 rows=53023) (actual ti
            -> Group aggregate: count(distinct e.Muscle_Group)
                -> Nested loop inner join  (cost=61493.13 rows=5
                    -> Nested loop inner join  (cost=42935.00 ro
                        -> Nested loop inner join  (cost=20509.0
                            -> Nested loop inner join  (cost=144
                                -> Nested loop inner join  (cost
                                    -> Nested loop inner join  (
                                        -> Sort: p.Plan_ID, p.Pl
                                            -> Table scan on p
                                        -> Covering index lookup
                                    -> Single-row covering index
                                -> Covering index lookup on sc u
                            -> Single-row covering index lookup
                        -> Covering index lookup on sec using PR
```

```
                      -> Single-row index lookup on e using PRIMAR
 |
```

Analysis

- **Total Cost**: `66795.46`
- **Execution Time**: `891.021 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

1. **Plan_Contains.Plan_ID** (Plan)

```sql
CREATE INDEX idx_plan_contains_plan_id ON Plan_Contains(Plan_
```

```
| -> Limit: 15 row(s)  (actual time=190.493..190.495 rows=15
    -> Sort: Muscle_Groups_Targeted DESC, p.Plan_ID DESC, lim
        -> Stream results  (cost=46341.41 rows=53023) (actual
            -> Group aggregate: count(distinct e.Muscle_Group
                -> Nested loop inner join  (cost=41039.08 row
                    -> Nested loop inner join  (cost=22480.95
                        -> Nested loop inner join  (cost=1286
                            -> Nested loop inner join  (cost=
                                -> Nested loop inner join  (c
                                    -> Nested loop inner join
                                        -> Sort: p.Plan_ID, p
                                            -> Table scan on
                                        -> Covering index loo
                                    -> Single-row covering in
```

```
                                          -> Covering index lookup on s
                                      -> Single-row covering index look
                                  -> Covering index lookup on sec using
                              -> Single-row index lookup on e using PRI
 |
```

New Time: 190.493

New Cost: 46341.41

This first added index results in approximately a 4x improvement in time spent on the query even though the cost is only decreased by 30%. This is because the GROUP BY operation is able to achieve significant speedup with the decreased input cost. The reduction is due to a significant decrease in the second-innermost nested loop inner join.

2. **Session_Contains.Session_ID** (Session and Plan_Contains)

```
CREATE INDEX idx_session_contains_session_id ON Session_Conta
```

```
| -> Limit: 15 row(s)  (actual time=245.797..245.799 rows=15
    -> Sort: Muscle_Groups_Targeted DESC, p.Plan_ID DESC, lim
        -> Stream results  (cost=48440.01 rows=53023) (actual
            -> Group aggregate: count(distinct e.Muscle_Group
                -> Nested loop inner join  (cost=43137.69 row
                    -> Nested loop inner join  (cost=24579.55
                        -> Nested loop inner join  (cost=1496
                            -> Nested loop inner join  (cost=
                                -> Nested loop inner join  (c
                                    -> Nested loop inner join
                                        -> Sort: p.Plan_ID, p
```

```
                                                      -> Table scan on
                                                  -> Covering index loo
                                              -> Single-row covering in
                                          -> Covering index lookup on s
                                      -> Single-row covering index look
                                  -> Covering index lookup on sec using
                              -> Single-row index lookup on e using PRI
|
```

Cost: 48440.01

Time: 245.797

We next add the index `idx_session_contains_session_id` which should decrease the lookup time for rows. However, we actually see a (mostly negligible) increase in cost and time. This is because while the time in the innermost loops is decreased, the overall cost is increased in a way that undoes the advantages earned.

3. **Set_Contains.Set_ID** (Sets and Session_Contains)

```
CREATE INDEX idx_set_contains_set_id ON Set_Contains(Set_I
```

```
| -> Limit: 15 row(s)  (actual time=145.172..145.174 rows=
    -> Sort: Muscle_Groups_Targeted DESC, p.Plan_ID DESC,
        -> Stream results  (cost=46341.41 rows=53023) (act
            -> Group aggregate: count(distinct e.Muscle_Gr
                -> Nested loop inner join  (cost=41039.08
                    -> Nested loop inner join  (cost=22480
                        -> Nested loop inner join  (cost=1
                            -> Nested loop inner join  (co
```

```
                                      -> Nested loop inner join
                                          -> Nested loop inner j
                                              -> Sort: p.Plan_ID
                                                  -> Table scan
                                              -> Covering index
                                          -> Single-row covering
                                      -> Covering index lookup o
                                  -> Single-row covering index l
                              -> Covering index lookup on sec us
                          -> Single-row index lookup on e using
 |
```

Cost: 46341.41

Time: 145.172

This addition results in an improvement in both cost and time relative to the previous iteration, which is a result of better lookups for all levels of indexing.

— 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=219
    -> Sort: RankedExercises.Muscle_Group, limit input to 15 row
        -> Index lookup on RankedExercises using <auto_key0> (rn
            -> Materialize CTE RankedExercises  (cost=0.00..0.00
                -> Window aggregate: row_number() OVER (PARTITIO
                    -> Sort: ExerciseFrequency.Muscle_Group, Exe
                        -> Table scan on ExerciseFrequency  (cos
                            -> Materialize CTE ExerciseFrequency
                                -> Table scan on <temporary>  (a
                                    -> Aggregate using temporary
                                        -> Nested loop inner joi
                                            -> Table scan on e
                                            -> Covering index lo
 |
```

Analysis

- **Total Cost**: 7902.88
- **Execution Time**: 219.908 for the limit operation
- Joins on Set_Contains.Exercise_ID and Exercises.Exercise_ID are costly

1. Muscle Groups — Excercise ID

```
CREATE INDEX idx_exercises_muscle_group_exercise_id ON Exercises
```

```
| -> Limit: 15 row(s)  (cost=0.10..0.10 rows=0) (actual time=55
    -> Sort: RankedExercises.Muscle_Group, limit input to 15 row
        -> Index lookup on RankedExercises using <auto_key0> (r
            -> Materialize CTE RankedExercises  (cost=0.00..0.00
                -> Window aggregate: row_number() OVER (PARTITIO
                    -> Sort: ExerciseFrequency.Muscle_Group, Exe
                        -> Table scan on ExerciseFrequency  (cos
                            -> Materialize CTE ExerciseFrequency
                                -> Table scan on <temporary>  (a
                                    -> Aggregate using temporary
                                        -> Nested loop inner joi
                                            -> Table scan on e 
                                            -> Covering index lo
 |
```

Time: 55.473

Despite no change in cost (at any phase), each lookup at the lowest level is now able to be completed in *far* less time (roughly 10x improvement), which results in an overall time improvement of about 75% overall.

2. SetContains—Exercise (only)

```
CREATE INDEX idx_set_exercise ON Set_Contains(Exercise_ID);
```

```
| -> Limit: 15 row(s)  (cost=0.10..0.10 rows=0) (actual time=123
    -> Sort: RankedExercises.Muscle_Group, limit input to 15 row
        -> Index lookup on RankedExercises using <auto_key0> (r
            -> Materialize CTE RankedExercises  (cost=0.00..0.00
                -> Window aggregate: row_number() OVER (PARTITIO
                    -> Sort: ExerciseFrequency.Muscle_Group, Exe
                        -> Table scan on ExerciseFrequency  (cos
                            -> Materialize CTE ExerciseFrequency
                                -> Table scan on <temporary>  (a
```

```
                                              -> Aggregate using temporary
                                                  -> Nested loop inner joi
                                                      -> Table scan on e
                                                      -> Covering index lo
 |
```

Time: 123.135

Similarly to #1, despite no change in cost, the time is greatly decreased due to much faster lookups at the lowest level. The overall decrease in time in this case is about half.

3. Both prior indexes at once:

```
| -> Limit: 15 row(s)  (cost=0.10..0.10 rows=0) (actual time=58
    -> Sort: RankedExercises.Muscle_Group, limit input to 15 row
        -> Index lookup on RankedExercises using <auto_key0> (rr
            -> Materialize CTE RankedExercises  (cost=0.00..0.00
                -> Window aggregate: row_number() OVER (PARTITIO
                    -> Sort: ExerciseFrequency.Muscle_Group, Exe
                        -> Table scan on ExerciseFrequency  (cos
                            -> Materialize CTE ExerciseFrequency
                                -> Table scan on <temporary>  (a
                                    -> Aggregate using temporary
                                        -> Nested loop inner joi
                                            -> Table scan on e
                                            -> Covering index lo
 |
```

Time: 58.242

Again, there is no change in cost. However, the noteworthy change here is that the benefits from the two indexes did not stack with each other, instead resulting in roughly the same performance as with just case #1, because the effect of #2 is subsumed within it.