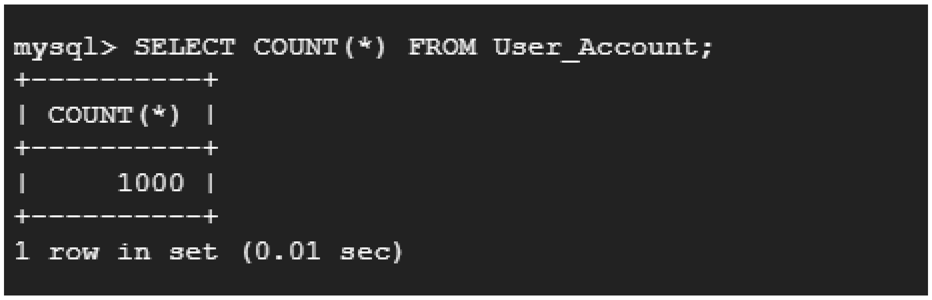
Assumptions (Noted by Lahari): The Budget entity will still have a one to one relationship with the Profile entity, but it will now be at most one from Budget to Profile, meaning that Budget will be its own entity and not reliant on Profiles to exist. A Profile can choose a Budget, but a Budget will not depend on a Profile. The same idea applies to Monthly_Plans, as Monthl_Plans will not be dependent on Budgets since there is now at most one relationship from Monthly_Plan to Budget.

DDL Commands:

```
CREATE TABLE User_Account (
    UserName VARCHAR(225) NOT NULL PRIMARY KEY,
    FirstName VARCHAR(225),
    Email VARCHAR(225),
    Password VARCHAR(225)
);


CREATE TABLE ProfileCreation (
    ProfileID INT NOT NULL PRIMARY KEY,
    Gender VARCHAR(225) NOT NULL,
    Age INT NOT NULL,
    Job BOOLEAN DEFAULT FALSE,
    Drinking BOOLEAN DEFAULT FALSE,
    Housing BOOLEAN DEFAULT FALSE,
    Major VARCHAR(255),
    Scholarship BOOLEAN DEFAULT FALSE,
    Smoking BOOLEAN DEFAULT FALSE,
    School_Year INT,
    PRIMARY KEY (UserName),
    FOREIGN KEY (UserName) REFERENCES User_Account(UserName)
);

CREATE TABLE Expenses (
    ExpenseID INT,
    ProfileID INT,
    CategoryName VARCHAR(255),
    ExpenseType VARCHAR(255),
    Amount INT,
    BudgetID INT,
    UserName VARCHAR(225),
    PRIMARY KEY (ExpenseID, ProfileID, UserName),
    FOREIGN KEY (ProfileID) REFERENCES ProfileCreation(ProfileID),
    FOREIGN KEY (BudgetID) REFERENCES Budget(BudgetID),
    FOREIGN KEY (UserName) REFERENCES ProfileCreation(UserName),
    CHECK (CategoryName IN ('Bills', 'Car', 'Other', 'Education',
'Entertainment', 'Friends', 'Tech', 'Rent', 'Dining Out/Take Out',
'Clothing', 'Trip', 'Grocery', 'Pet', 'Family')),
    CHECK (ExpenseType IN ('Want', 'Need'))
);

CREATE TABLE Monthly_Plans (
    PlanID INT NOT NULL PRIMARY KEY,
    Needs INT,
    Wants INT,
    Savings INT,
    BudgetID INT,
    FOREIGN KEY (BudgetID) REFERENCES Budget(BudgetID)
);

CREATE TABLE Budget (
    BudgetID INT NOT NULL PRIMARY KEY,
    ProfileID VARCHAR(30),
    Money REAL,
    FOREIGN KEY (ProfileID) REFERENCES ProfileCreation(ProfileID)
);
```

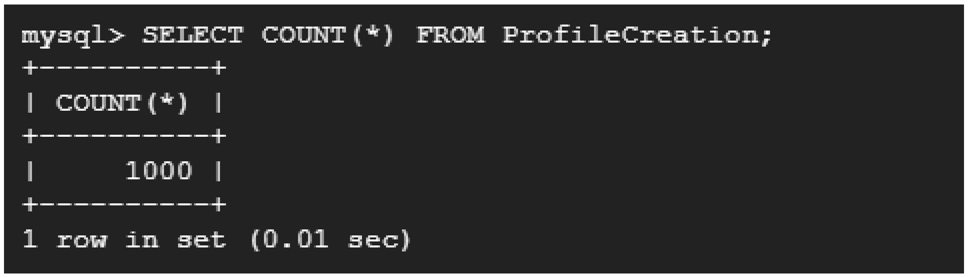
Table Counts Screenshot

User_Account Table Count:



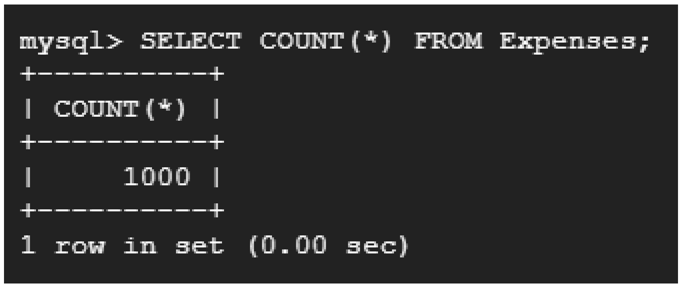
```
mysql> SELECT COUNT(*) FROM User_Account;
+----------+
| COUNT(*) |
+----------+
|     1000 |
+----------+
1 row in set (0.01 sec)
```

ProfileCreation Table Count:



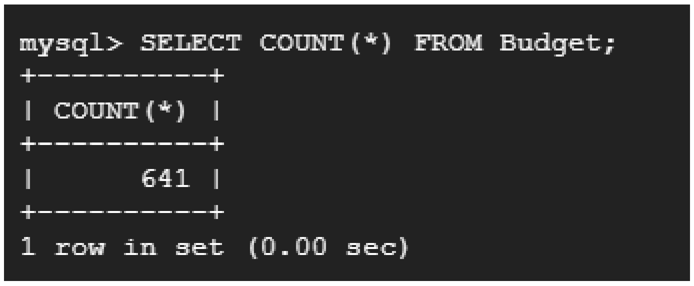
```
mysql> SELECT COUNT(*) FROM ProfileCreation;
+----------+
| COUNT(*) |
+----------+
|     1000 |
+----------+
1 row in set (0.01 sec)
```

Expenses Table Count:



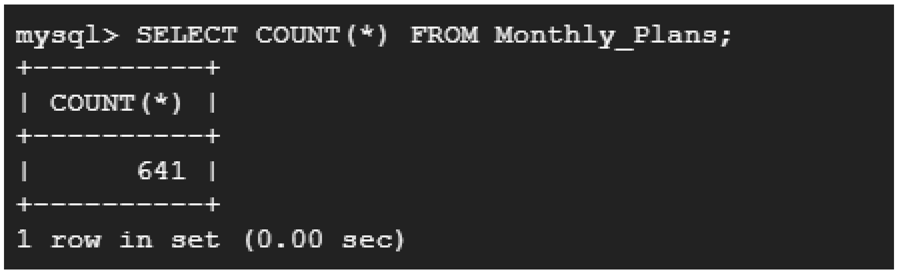
```
mysql> SELECT COUNT(*) FROM Expenses;
+----------+
| COUNT(*) |
+----------+
|     1000 |
+----------+
1 row in set (0.00 sec)
```

Budget Table Count:



```
mysql> SELECT COUNT(*) FROM Budget;
+----------+
| COUNT(*) |
+----------+
|      641 |
+----------+
1 row in set (0.00 sec)
```

Monthly_Plans:



```
mysql> SELECT COUNT(*) FROM Monthly_Plans;
+----------+
| COUNT(*) |
+----------+
|      641 |
+----------+
1 row in set (0.00 sec)
```

Part 1: Advanced SQL Queries

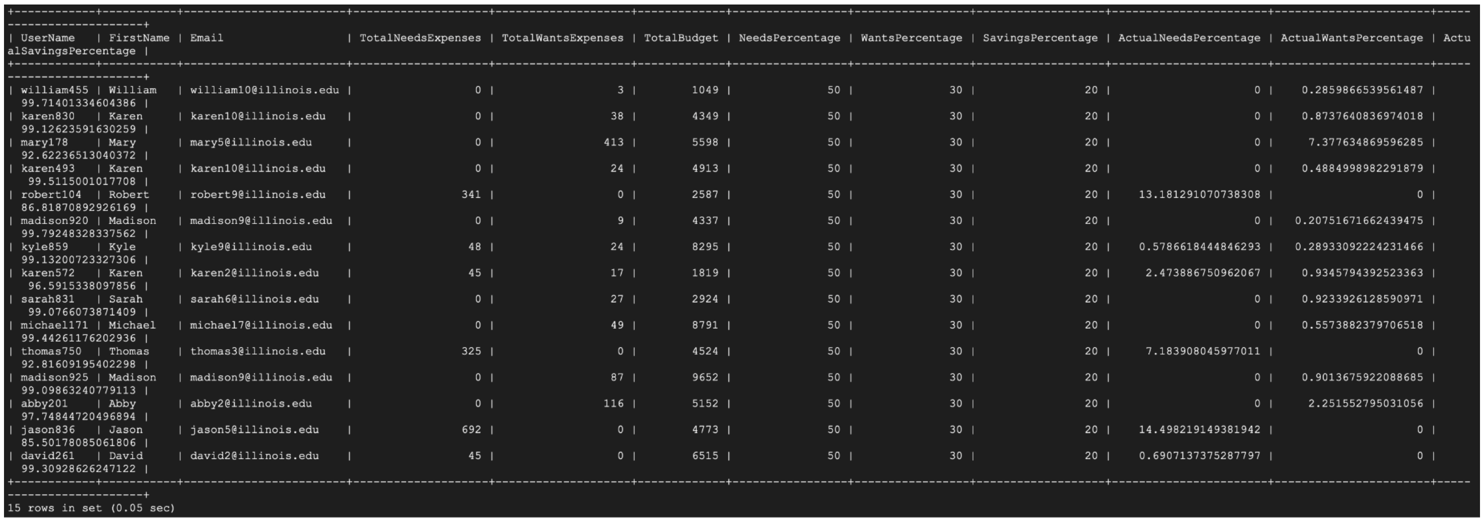
Query 1:

1.) Find users that are currently sticking to their monthly plan (GROUP BY, Join Relations(Expenses, Monthly Plan, Budgets, Profiles))

```
SELECT
    ua.UserName,
    ua.FirstName,
    ua.Email,
    SUM(CASE WHEN e.ExpenseType = 'Need' THEN e.Amount ELSE 0 END) AS
TotalNeedsExpenses,
    SUM(CASE WHEN e.ExpenseType = 'Want' THEN e.Amount ELSE 0 END) AS
TotalWantsExpenses,
    b.Money AS TotalBudget,
    mp.Needs AS NeedsPercentage,
    mp.Wants AS WantsPercentage,
    mp.Savings AS SavingsPercentage,
    (SUM(CASE WHEN e.ExpenseType = 'Need' THEN e.Amount ELSE 0 END) /
b.Money) * 100 AS ActualNeedsPercentage,
    (SUM(CASE WHEN e.ExpenseType = 'Want' THEN e.Amount ELSE 0 END) /
b.Money) * 100 AS ActualWantsPercentage,
    ((b.Money - SUM(CASE WHEN e.ExpenseType IN ('Need', 'Want') THEN
e.Amount ELSE 0 END)) / b.Money) * 100 AS ActualSavingsPercentage
FROM
    User_Account ua
JOIN
    Expenses e ON ua.UserName = e.UserName
JOIN
    Budget b ON e.BudgetID = b.BudgetID
JOIN
    Monthly_Plans mp ON b.BudgetID = mp.BudgetID
GROUP BY
    ua.UserName,
    ua.FirstName,
    ua.Email,
    b.Money,
    mp.Needs,
    mp.Wants,
    mp.Savings
HAVING
    (SUM(CASE WHEN e.ExpenseType = 'Need' THEN e.Amount ELSE 0 END) /
```

```sql
b.Money) * 100 <= mp.Needs
    AND (SUM(CASE WHEN e.ExpenseType = 'Want' THEN e.Amount ELSE 0 END) /
b.Money) * 100 <= mp.Wants
    AND ((b.Money - SUM(CASE WHEN e.ExpenseType IN ('Need', 'Want') THEN
e.Amount ELSE 0 END)) / b.Money) * 100 >= mp.Savings;
```

Top 15 Results (Query 1):



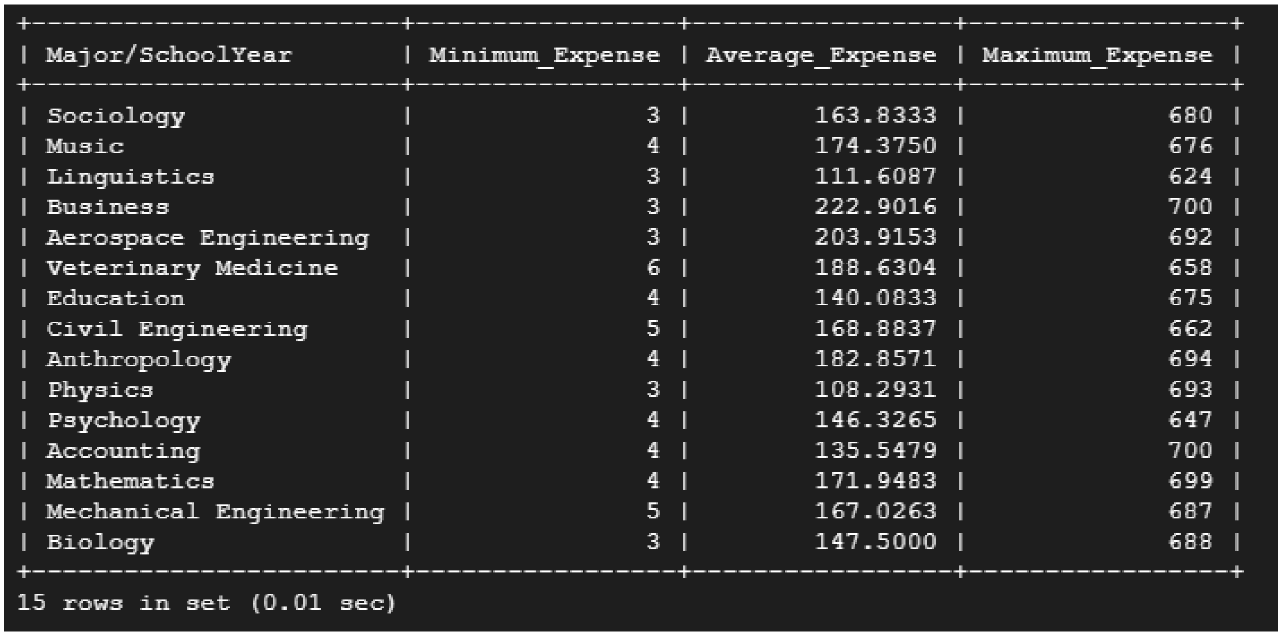
```
+------------+-----------+------------------------+--------------------+--------------------+-------------+-----------------+-----------------+-------------------+-----------------------+-----------------------+-----
--------------------+
| UserName   | FirstName | Email                  | TotalNeedsExpenses | TotalWantsExpenses | TotalBudget | NeedsPercentage | WantsPercentage | SavingsPercentage | ActualNeedsPercentage | ActualWantsPercentage | Actu
alSavingsPercentage |
+------------+-----------+------------------------+--------------------+--------------------+-------------+-----------------+-----------------+-------------------+-----------------------+-----------------------+-----
--------------------+
| william455 | William   | william10@illinois.edu |                  0 |                  3 |        1049 |              50 |              30 |                20 |                     0 |    0.2859866539561487 |
  99.71401334604386 |
| karen830   | Karen     | karen10@illinois.edu   |                  0 |                 38 |        4349 |              50 |              30 |                20 |                     0 |    0.8737640836974018 |
  99.12623591630259 |
| mary178    | Mary      | mary5@illinois.edu     |                  0 |                413 |        5598 |              50 |              30 |                20 |                     0 |     7.377634869596285 |
  92.62236513040372 |
| karen493   | Karen     | karen10@illinois.edu   |                  0 |                 24 |        4913 |              50 |              30 |                20 |                     0 |    0.4884998982291879 |
   99.5115001017708 |
| robert104  | Robert    | robert9@illinois.edu   |                341 |                  0 |        2587 |              50 |              30 |                20 |    13.181291070738308 |                     0 |
  86.81870892926169 |
| madison920 | Madison   | madison9@illinois.edu  |                  0 |                  9 |        4337 |              50 |              30 |                20 |                     0 |   0.20751671662439475 |
  99.79248328337562 |
| kyle859    | Kyle      | kyle9@illinois.edu     |                 48 |                 24 |        8295 |              50 |              30 |                20 |    0.5786618444846293 |   0.28933092224231466 |
  99.13200723327306 |
| karen572   | Karen     | karen2@illinois.edu    |                 45 |                 17 |        1819 |              50 |              30 |                20 |     2.473886750962067 |    0.9345794392523363 |
   96.5915338097856 |
| sarah831   | Sarah     | sarah6@illinois.edu    |                  0 |                 27 |        2924 |              50 |              30 |                20 |                     0 |    0.9233926128590971 |
   99.0766073871409 |
| michael171 | Michael   | michael7@illinois.edu  |                  0 |                 49 |        8791 |              50 |              30 |                20 |                     0 |    0.5573882379706518 |
  99.44261176202936 |
| thomas750  | Thomas    | thomas3@illinois.edu   |                325 |                  0 |        4524 |              50 |              30 |                20 |     7.183908045977011 |                     0 |
  92.81609195402298 |
| madison925 | Madison   | madison9@illinois.edu  |                  0 |                 87 |        9652 |              50 |              30 |                20 |                     0 |    0.9013675922088685 |
  99.09863240779113 |
| abby201    | Abby      | abby2@illinois.edu     |                  0 |                116 |        5152 |              50 |              30 |                20 |                     0 |     2.251552795031056 |
  97.74844720496894 |
| jason836   | Jason     | jason5@illinois.edu    |                692 |                  0 |        4773 |              50 |              30 |                20 |    14.498219149381942 |                     0 |
  85.50178085061806 |
| david261   | David     | david2@illinois.edu    |                 45 |                  0 |        6515 |              50 |              30 |                20 |    0.6907137375287797 |                     0 |
  99.30928626247122 |
+------------+-----------+------------------------+--------------------+--------------------+-------------+-----------------+-----------------+-------------------+-----------------------+-----------------------+-----
--------------------+
15 rows in set (0.05 sec)
```

Query 2:

2.) Get the MIN(), AVG(), MAX() expense of each major and each school year making sure that the maximum expense is greater than $500 for each group. (GROUP BY, SET Operators (UNION))

```sql
SELECT pc.Major AS 'Major/SchoolYear', MIN(e.Amount) AS Minimum_Expense,
AVG(e.Amount) AS Average_Expense, MAX(e.Amount) AS Maximum_Expense
FROM ProfileCreation pc NATURAL JOIN Expenses e JOIN Budget b ON
(e.BudgetID = b.BudgetID)
GROUP BY pc.Major
HAVING Maximum_Expense > 500

UNION

SELECT CAST(pc.School_Year AS CHAR) AS 'Major/SchoolYear', MIN(e.Amount) AS
Minimum_Expense, AVG(e.Amount) AS Average_Expense, MAX(e.Amount) AS
Maximum_Expense
FROM ProfileCreation pc NATURAL JOIN Expenses e JOIN Budget b ON
(e.BudgetID = b. BudgetID)
GROUP BY pc.School_Year
HAVING Maximum_Expense > 500;
```

Top 15 Results (Query 2):



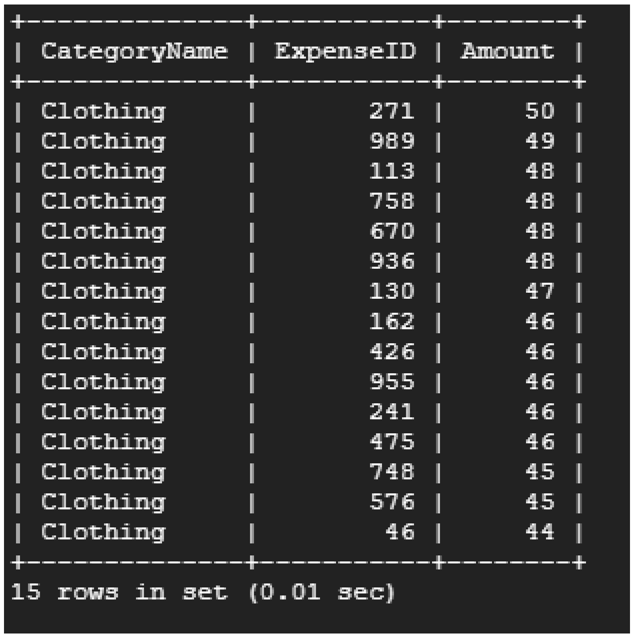
```
+------------------------+-----------------+-----------------+-----------------+
| Major/SchoolYear       | Minimum_Expense | Average_Expense | Maximum_Expense |
+------------------------+-----------------+-----------------+-----------------+
| Sociology              |               3 |        163.8333 |             680 |
| Music                  |               4 |        174.3750 |             676 |
| Linguistics            |               3 |        111.6087 |             624 |
| Business               |               3 |        222.9016 |             700 |
| Aerospace Engineering  |               3 |        203.9153 |             692 |
| Veterinary Medicine    |               6 |        188.6304 |             658 |
| Education              |               4 |        140.0833 |             675 |
| Civil Engineering      |               5 |        168.8837 |             662 |
| Anthropology           |               4 |        182.8571 |             694 |
| Physics                |               3 |        108.2931 |             693 |
| Psychology             |               4 |        146.3265 |             647 |
| Accounting             |               4 |        135.5479 |             700 |
| Mathematics            |               4 |        171.9483 |             699 |
| Mechanical Engineering |               5 |        167.0263 |             687 |
| Biology                |               3 |        147.5000 |             688 |
+------------------------+-----------------+-----------------+-----------------+
15 rows in set (0.01 sec)
```

Query 3:

3.) This SQL query is used to get the top 10 most expensive of each expense category that is a want and who has a monthly budget that is over $100.

```sql
SELECT ex.Amount, ex.CategoryName, ex.ExpenseID
FROM Expenses ex JOIN Budget b ON ex.BudgetId = b.BudgetId
WHERE b.Money > 100.00 AND ex.ExpenseType = 'Want' AND ex.Amount =
    (SELECT MAX(ex2.Amount)
     FROM Expenses ex2
     WHERE ex2.CategoryName = ex.CategoryName AND ex2.BudgetId =
ex.BudgetId
     GROUP BY ex2.CategoryName
     ORDER BY MAX(ex2.Amount) DESC
     LIMIT 10)
ORDER BY ex.CategoryName, ex.Amount DESC;
```

Top 15 Results (Query 3):



```
+--------------+-----------+--------+
| CategoryName | ExpenseID | Amount |
+--------------+-----------+--------+
| Clothing     |       271 |     50 |
| Clothing     |       989 |     49 |
| Clothing     |       113 |     48 |
| Clothing     |       758 |     48 |
| Clothing     |       670 |     48 |
| Clothing     |       936 |     48 |
| Clothing     |       130 |     47 |
| Clothing     |       162 |     46 |
| Clothing     |       426 |     46 |
| Clothing     |       955 |     46 |
| Clothing     |       241 |     46 |
| Clothing     |       475 |     46 |
| Clothing     |       748 |     45 |
| Clothing     |       576 |     45 |
| Clothing     |        46 |     44 |
+--------------+-----------+--------+
15 rows in set (0.01 sec)
```

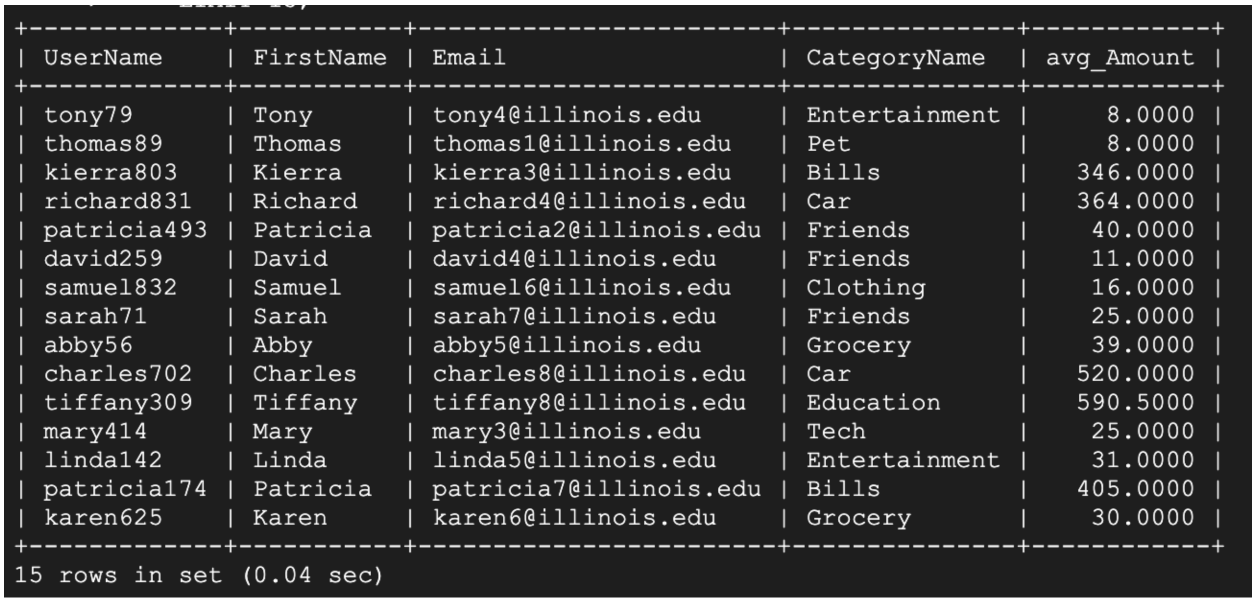
Query 4:

4.)

This query finds individuals who have spent money in more than one category. Then it checks what the average amount they have spent in that category. The purpose behind this is to “ignore” accounts with expenses in a single category and look at more holistic spending patterns to identify individual averages. (Subquery, Group By)

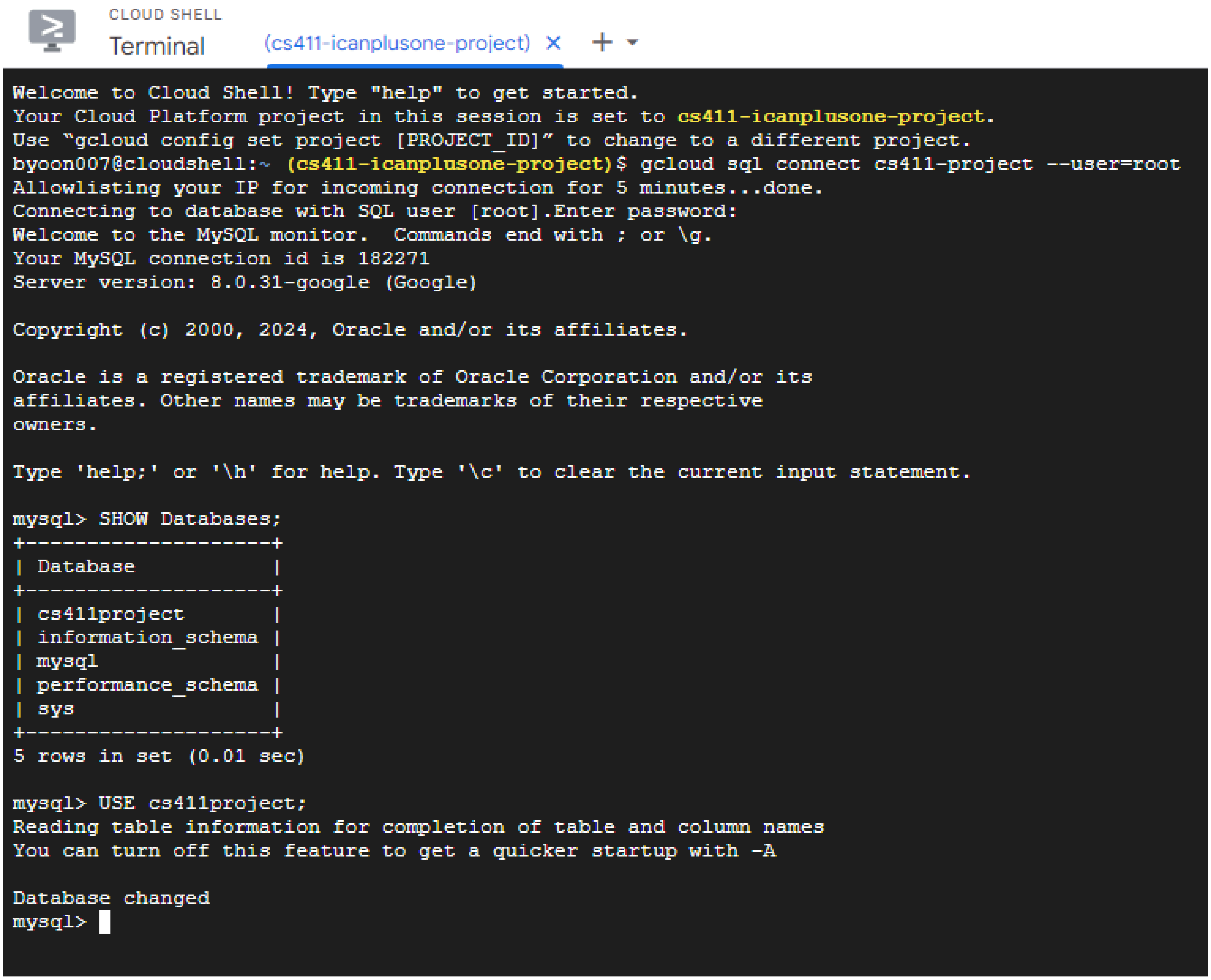
```sql
Select UserName, FirstName, Email, CategoryName, AVG(Amount) as avg_Amount
From User_Account ua NATURAL JOIN Expenses e
Where e.UserName IN (Select e1.UserName
                     From Expenses e1
                     Group by e1.UserName
                     Having Count(Distinct e1.CategoryName) > 1)
Group By UserName, e.CategoryName;
```

Top 15 Results (Query 4):



```
+-------------+-----------+------------------------+---------------+------------+
| UserName    | FirstName | Email                  | CategoryName  | avg_Amount |
+-------------+-----------+------------------------+---------------+------------+
| tony79      | Tony      | tony4@illinois.edu     | Entertainment |     8.0000 |
| thomas89    | Thomas    | thomas1@illinois.edu   | Pet           |     8.0000 |
| kierra803   | Kierra    | kierra3@illinois.edu   | Bills         |   346.0000 |
| richard831  | Richard   | richard4@illinois.edu  | Car           |   364.0000 |
| patricia493 | Patricia  | patricia2@illinois.edu | Friends       |    40.0000 |
| david259    | David     | david4@illinois.edu    | Friends       |    11.0000 |
| samuel832   | Samuel    | samuel6@illinois.edu   | Clothing      |    16.0000 |
| sarah71     | Sarah     | sarah7@illinois.edu    | Friends       |    25.0000 |
| abby56      | Abby      | abby5@illinois.edu     | Grocery       |    39.0000 |
| charles702  | Charles   | charles8@illinois.edu  | Car           |   520.0000 |
| tiffany309  | Tiffany   | tiffany8@illinois.edu  | Education     |   590.5000 |
| mary414     | Mary      | mary3@illinois.edu     | Tech          |    25.0000 |
| linda142    | Linda     | linda5@illinois.edu    | Entertainment |    31.0000 |
| patricia174 | Patricia  | patricia7@illinois.edu | Bills         |   405.0000 |
| karen625    | Karen     | karen6@illinois.edu    | Grocery       |    30.0000 |
+-------------+-----------+------------------------+---------------+------------+
15 rows in set (0.04 sec)
```

GCP SQL Connection:

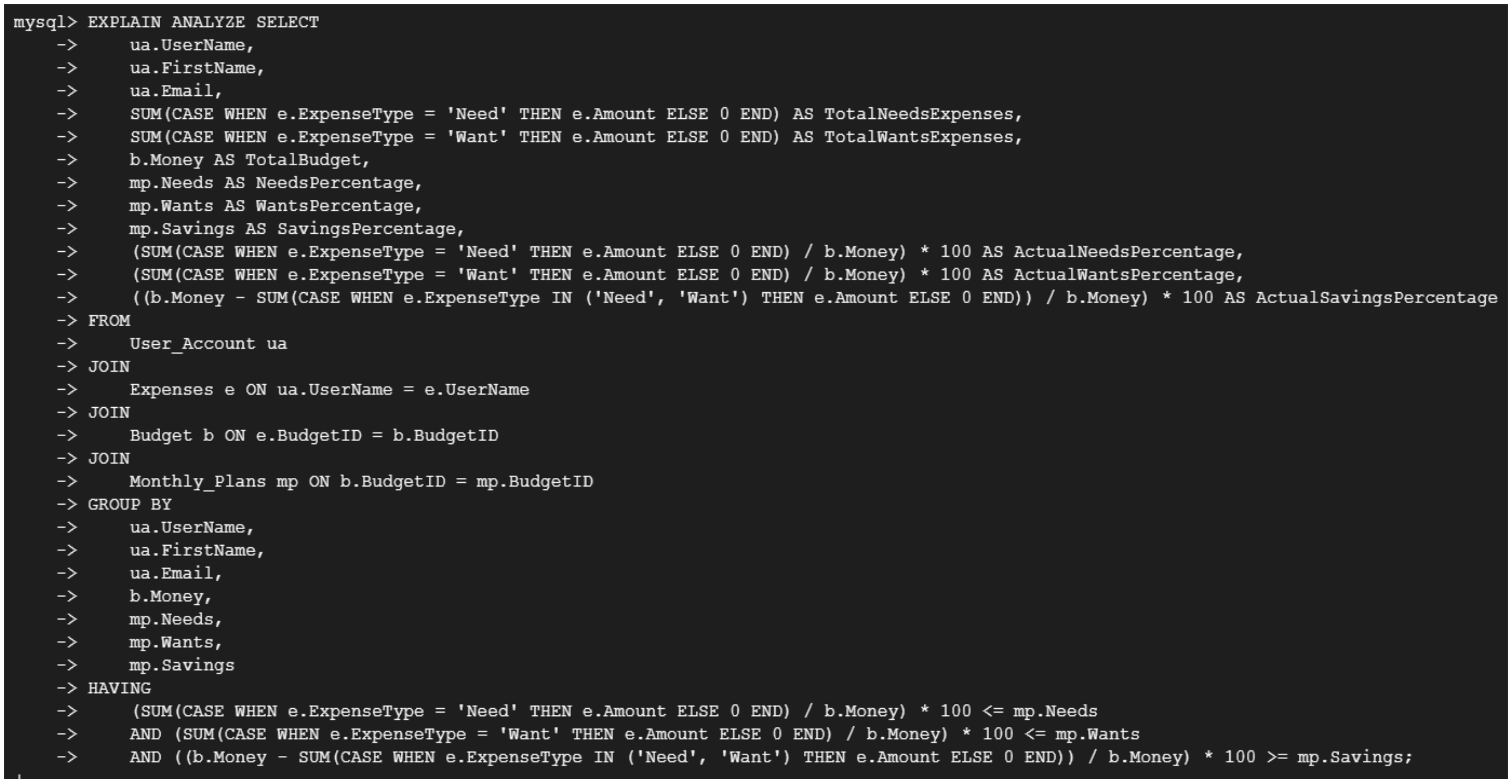


```
Welcome to Cloud Shell! Type "help" to get started.
Your Cloud Platform project in this session is set to cs411-icanplusone-project.
Use "gcloud config set project [PROJECT_ID]" to change to a different project.
byoon007@cloudshell:~ (cs411-icanplusone-project)$ gcloud sql connect cs411-project --user=root
Allowlisting your IP for incoming connection for 5 minutes...done.
Connecting to database with SQL user [root].Enter password:
Welcome to the MySQL monitor.  Commands end with ; or \g.
Your MySQL connection id is 182271
Server version: 8.0.31-google (Google)

Copyright (c) 2000, 2024, Oracle and/or its affiliates.

Oracle is a registered trademark of Oracle Corporation and/or its
affiliates. Other names may be trademarks of their respective
owners.

Type 'help;' or '\h' for help. Type '\c' to clear the current input statement.

mysql> SHOW Databases;
+--------------------+
| Database           |
+--------------------+
| cs411project       |
| information_schema |
| mysql              |
| performance_schema |
| sys                |
+--------------------+
5 rows in set (0.01 sec)

mysql> USE cs411project;
Reading table information for completion of table and column names
You can turn off this feature to get a quicker startup with -A

Database changed
mysql> 
```

<u>Part 2: Indexing:</u>

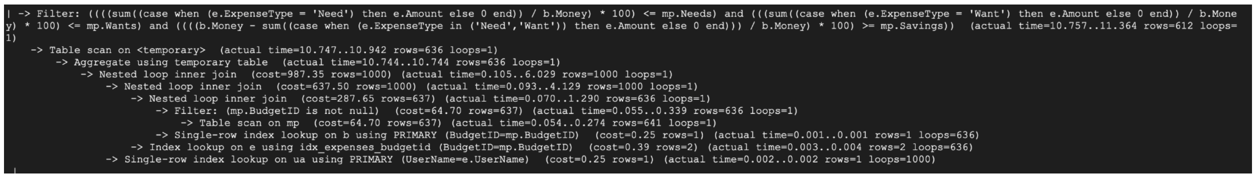
Query 1:

EXPLAIN ANALYZE Command:



```
mysql> EXPLAIN ANALYZE SELECT
    ->     ua.UserName,
    ->     ua.FirstName,
    ->     ua.Email,
    ->     SUM(CASE WHEN e.ExpenseType = 'Need' THEN e.Amount ELSE 0 END) AS TotalNeedsExpenses,
    ->     SUM(CASE WHEN e.ExpenseType = 'Want' THEN e.Amount ELSE 0 END) AS TotalWantsExpenses,
    ->     b.Money AS TotalBudget,
    ->     mp.Needs AS NeedsPercentage,
    ->     mp.Wants AS WantsPercentage,
    ->     mp.Savings AS SavingsPercentage,
    ->     (SUM(CASE WHEN e.ExpenseType = 'Need' THEN e.Amount ELSE 0 END) / b.Money) * 100 AS ActualNeedsPercentage,
    ->     (SUM(CASE WHEN e.ExpenseType = 'Want' THEN e.Amount ELSE 0 END) / b.Money) * 100 AS ActualWantsPercentage,
    ->     ((b.Money - SUM(CASE WHEN e.ExpenseType IN ('Need', 'Want') THEN e.Amount ELSE 0 END)) / b.Money) * 100 AS ActualSavingsPercentage
    -> FROM
    ->     User_Account ua
    -> JOIN
    ->     Expenses e ON ua.UserName = e.UserName
    -> JOIN
    ->     Budget b ON e.BudgetID = b.BudgetID
    -> JOIN
    ->     Monthly_Plans mp ON b.BudgetID = mp.BudgetID
    -> GROUP BY
    ->     ua.UserName,
    ->     ua.FirstName,
    ->     ua.Email,
    ->     b.Money,
    ->     mp.Needs,
    ->     mp.Wants,
    ->     mp.Savings
    -> HAVING
    ->     (SUM(CASE WHEN e.ExpenseType = 'Need' THEN e.Amount ELSE 0 END) / b.Money) * 100 <= mp.Needs
    ->     AND (SUM(CASE WHEN e.ExpenseType = 'Want' THEN e.Amount ELSE 0 END) / b.Money) * 100 <= mp.Wants
    ->     AND ((b.Money - SUM(CASE WHEN e.ExpenseType IN ('Need', 'Want') THEN e.Amount ELSE 0 END)) / b.Money) * 100 >= mp.Savings;
```

Without Indexing:

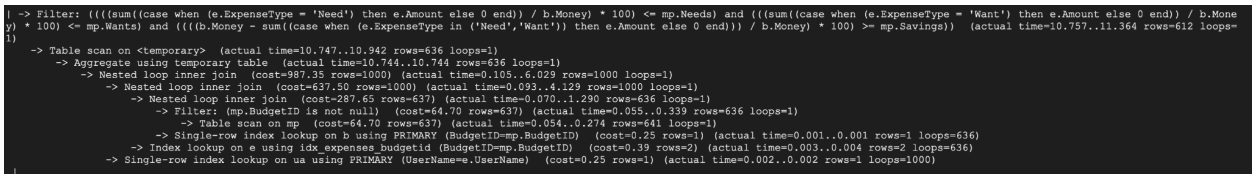


```
| -> Filter: ((((sum((case when (e.ExpenseType = 'Need') then e.Amount else 0 end)) / b.Money) * 100) <= mp.Needs) and (((sum((case when (e.ExpenseType = 'Want') then e.Amount else 0 end)) / b.Mone
y) * 100) <= mp.Wants) and ((((b.Money - sum((case when (e.ExpenseType in ('Need','Want')) then e.Amount else 0 end))) / b.Money) * 100) >= mp.Savings))  (actual time=10.757..11.364 rows=612 loops=
1)
    -> Table scan on <temporary>  (actual time=10.747..10.942 rows=636 loops=1)
        -> Aggregate using temporary table  (actual time=10.744..10.744 rows=636 loops=1)
            -> Nested loop inner join  (cost=987.35 rows=1000) (actual time=0.105..6.029 rows=1000 loops=1)
                -> Nested loop inner join  (cost=637.50 rows=1000) (actual time=0.093..4.129 rows=1000 loops=1)
                    -> Nested loop inner join  (cost=287.65 rows=637) (actual time=0.070..1.290 rows=636 loops=1)
                        -> Filter: (mp.BudgetID is not null)  (cost=64.70 rows=637) (actual time=0.055..0.339 rows=636 loops=1)
                            -> Table scan on mp  (cost=64.70 rows=637) (actual time=0.054..0.274 rows=641 loops=1)
                        -> Single-row index lookup on b using PRIMARY (BudgetID=mp.BudgetID)  (cost=0.25 rows=1) (actual time=0.001..0.001 rows=1 loops=636)
                    -> Index lookup on e using idx_expenses_budgetid (BudgetID=mp.BudgetID)  (cost=0.39 rows=2) (actual time=0.003..0.004 rows=2 loops=636)
                -> Single-row index lookup on ua using PRIMARY (UserName=e.UserName)  (cost=0.25 rows=1) (actual time=0.002..0.002 rows=1 loops=1000)
```

Without indexing, the major cost consists of the three nested loop inner joins occurring with costs of 987.35, 637.5, and 287.65. Additionally, the table scan on Monthly_Plans has a cost of 64.7.

Indexing on Expenses.ExpenseType

CREATE INDEX q1_EXPexptype ON Expenses(ExpenseType);

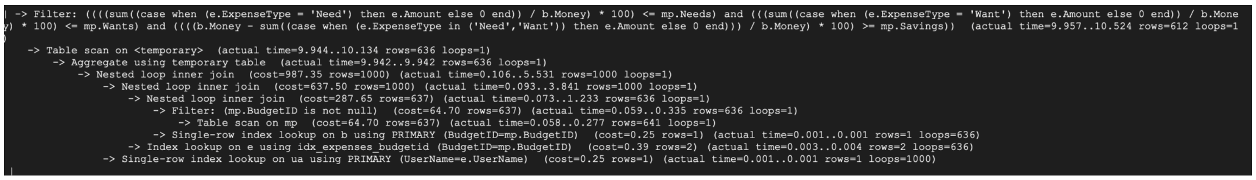


```
| -> Filter: ((((sum((case when (e.ExpenseType = 'Need') then e.Amount else 0 end)) / b.Money) * 100) <= mp.Needs) and (((sum((case when (e.ExpenseType = 'Want') then e.Amount else 0 end)) / b.Mone
y) * 100) <= mp.Wants) and ((((b.Money - sum((case when (e.ExpenseType in ('Need','Want')) then e.Amount else 0 end))) / b.Money) * 100) >= mp.Savings))  (actual time=10.757..11.364 rows=612 loops=
1)
    -> Table scan on <temporary>  (actual time=10.747..10.942 rows=636 loops=1)
        -> Aggregate using temporary table  (actual time=10.744..10.744 rows=636 loops=1)
            -> Nested loop inner join  (cost=987.35 rows=1000) (actual time=0.105..6.029 rows=1000 loops=1)
                -> Nested loop inner join  (cost=637.50 rows=1000) (actual time=0.093..4.129 rows=1000 loops=1)
                    -> Nested loop inner join  (cost=287.65 rows=637) (actual time=0.070..1.290 rows=636 loops=1)
                        -> Filter: (mp.BudgetID is not null)  (cost=64.70 rows=637) (actual time=0.055..0.339 rows=636 loops=1)
                            -> Table scan on mp  (cost=64.70 rows=637) (actual time=0.054..0.274 rows=641 loops=1)
                        -> Single-row index lookup on b using PRIMARY (BudgetID=mp.BudgetID)  (cost=0.25 rows=1) (actual time=0.001..0.001 rows=1 loops=636)
                    -> Index lookup on e using idx_expenses_budgetid (BudgetID=mp.BudgetID)  (cost=0.39 rows=2) (actual time=0.003..0.004 rows=2 loops=636)
                -> Single-row index lookup on ua using PRIMARY (UserName=e.UserName)  (cost=0.25 rows=1) (actual time=0.002..0.002 rows=1 loops=1000)
 |
```

We created an index on Expenses.ExpenseType, but it did not improve the cost of the query, which is most likely caused by the fact that the advanced query has a lot of conditions in it to filter out records that do not meet the criteria. This most likely causes the indexing to not have a large impact on the cost.

Indexing on Monthly_Plans.Needs, Monthly_Plans.Wants, Monthly_Plans.Savings

CREATE INDEX q1_MPn ON Monthly_Plans(Needs);
CREATE INDEX q1_MPw ON Monthly_Plans(Wants);
CREATE INDEX q1_MPs ON Monthly_Plans(Savings);

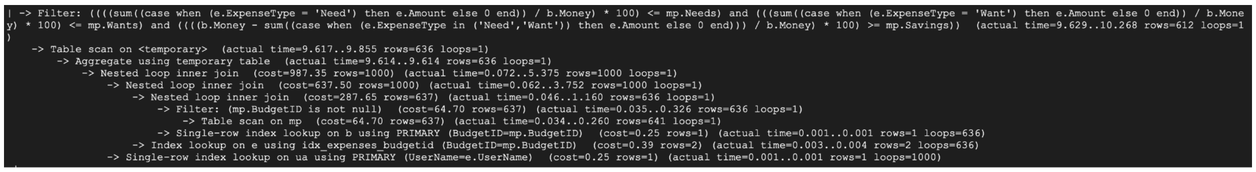


```
| -> Filter: ((((sum((case when (e.ExpenseType = 'Need') then e.Amount else 0 end)) / b.Money) * 100) <= mp.Needs) and (((sum((case when (e.ExpenseType = 'Want') then e.Amount else 0 end)) / b.Mone
y) * 100) <= mp.Wants) and ((((b.Money - sum((case when (e.ExpenseType in ('Need','Want')) then e.Amount else 0 end))) / b.Money) * 100) >= mp.Savings))  (actual time=9.957..10.524 rows=612 loops=1
)
    -> Table scan on <temporary>  (actual time=9.944..10.134 rows=636 loops=1)
        -> Aggregate using temporary table  (actual time=9.942..9.942 rows=636 loops=1)
            -> Nested loop inner join  (cost=987.35 rows=1000) (actual time=0.106..5.531 rows=1000 loops=1)
                -> Nested loop inner join  (cost=637.50 rows=1000) (actual time=0.093..3.841 rows=1000 loops=1)
                    -> Nested loop inner join  (cost=287.65 rows=637) (actual time=0.073..1.233 rows=636 loops=1)
                        -> Filter: (mp.BudgetID is not null)  (cost=64.70 rows=637) (actual time=0.059..0.335 rows=636 loops=1)
                            -> Table scan on mp  (cost=64.70 rows=637) (actual time=0.058..0.277 rows=641 loops=1)
                        -> Single-row index lookup on b using PRIMARY (BudgetID=mp.BudgetID)  (cost=0.25 rows=1) (actual time=0.001..0.001 rows=1 loops=636)
                    -> Index lookup on e using idx_expenses_budgetid (BudgetID=mp.BudgetID)  (cost=0.39 rows=2) (actual time=0.003..0.004 rows=2 loops=636)
                -> Single-row index lookup on ua using PRIMARY (UserName=e.UserName)  (cost=0.25 rows=1) (actual time=0.001..0.001 rows=1 loops=1000)
 |
```

We created an index on the Needs, Wants, and Savings attributes in the Monthly_Plans table, but it did not improve the cost of the query, which is most likely caused by the fact that the advanced query has a lot of conditions in it to filter out records that do not meet the criteria. This most likely causes the indexing to not have a large impact on the cost.

Indexing on Budget.Money

CREATE INDEX q1_BUDmon ON Budget(Money);

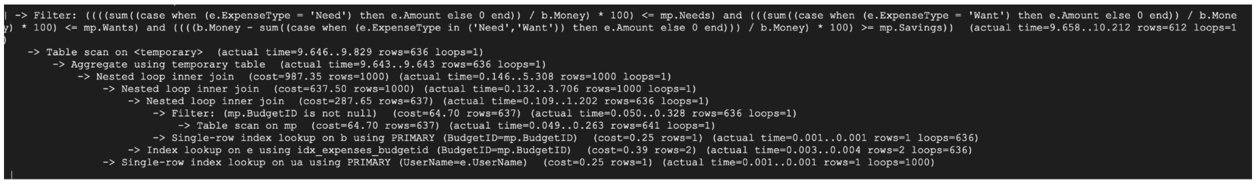


```
| -> Filter: ((((sum((case when (e.ExpenseType = 'Need') then e.Amount else 0 end)) / b.Money) * 100) <= mp.Needs) and (((sum((case when (e.ExpenseType = 'Want') then e.Amount else 0 end)) / b.Mone
y) * 100) <= mp.Wants) and ((((b.Money - sum((case when (e.ExpenseType in ('Need','Want')) then e.Amount else 0 end))) / b.Money) * 100) >= mp.Savings))  (actual time=9.629..10.268 rows=612 loops=1
)
    -> Table scan on <temporary>  (actual time=9.617..9.855 rows=636 loops=1)
        -> Aggregate using temporary table  (actual time=9.614..9.614 rows=636 loops=1)
            -> Nested loop inner join  (cost=987.35 rows=1000) (actual time=0.072..5.375 rows=1000 loops=1)
                -> Nested loop inner join  (cost=637.50 rows=1000) (actual time=0.062..3.752 rows=1000 loops=1)
                    -> Nested loop inner join  (cost=287.65 rows=637) (actual time=0.046..1.160 rows=636 loops=1)
                        -> Filter: (mp.BudgetID is not null)  (cost=64.70 rows=637) (actual time=0.035..0.326 rows=636 loops=1)
                            -> Table scan on mp  (cost=64.70 rows=637) (actual time=0.034..0.260 rows=641 loops=1)
                        -> Single-row index lookup on b using PRIMARY (BudgetID=mp.BudgetID)  (cost=0.25 rows=1) (actual time=0.001..0.001 rows=1 loops=636)
                    -> Index lookup on e using idx_expenses_budgetid (BudgetID=mp.BudgetID)  (cost=0.39 rows=2) (actual time=0.003..0.004 rows=2 loops=636)
                -> Single-row index lookup on ua using PRIMARY (UserName=e.UserName)  (cost=0.25 rows=1) (actual time=0.001..0.001 rows=1 loops=1000)
```

We created an index on Budget.Money, but it did not improve the cost of the query, which is most likely caused by the fact that the advanced query has a lot of conditions in it to filter out records that do not meet the criteria. This most likely causes the indexing to not have a large impact on the cost.

Indexing on User_Account.FirstName, User_Account.Email

CREATE INDEX q1_UAfn ON User_Account(FirstName);
CREATE INDEX q1_UAe ON User_Account(Email);



```
| -> Filter: ((((sum((case when (e.ExpenseType = 'Need') then e.Amount else 0 end)) / b.Money) * 100) <= mp.Needs) and (((sum((case when (e.ExpenseType = 'Want') then e.Amount else 0 end)) / b.Mone
y) * 100) <= mp.Wants) and ((((b.Money - sum((case when (e.ExpenseType in ('Need','Want')) then e.Amount else 0 end))) / b.Money) * 100) >= mp.Savings))  (actual time=9.658..10.212 rows=612 loops=1
)
    -> Table scan on <temporary>  (actual time=9.646..9.829 rows=636 loops=1)
        -> Aggregate using temporary table  (actual time=9.643..9.643 rows=636 loops=1)
            -> Nested loop inner join  (cost=987.35 rows=1000) (actual time=0.146..5.308 rows=1000 loops=1)
                -> Nested loop inner join  (cost=637.50 rows=1000) (actual time=0.132..3.706 rows=1000 loops=1)
                    -> Nested loop inner join  (cost=287.65 rows=637) (actual time=0.109..1.202 rows=636 loops=1)
                        -> Filter: (mp.BudgetID is not null)  (cost=64.70 rows=637) (actual time=0.050..0.328 rows=636 loops=1)
                            -> Table scan on mp  (cost=64.70 rows=637) (actual time=0.049..0.263 rows=641 loops=1)
                        -> Single-row index lookup on b using PRIMARY (BudgetID=mp.BudgetID)  (cost=0.25 rows=1) (actual time=0.001..0.001 rows=1 loops=636)
                    -> Index lookup on e using idx_expenses_budgetid (BudgetID=mp.BudgetID)  (cost=0.39 rows=2) (actual time=0.003..0.004 rows=2 loops=636)
                -> Single-row index lookup on ua using PRIMARY (UserName=e.UserName)  (cost=0.25 rows=1) (actual time=0.001..0.001 rows=1 loops=1000)
 |
```

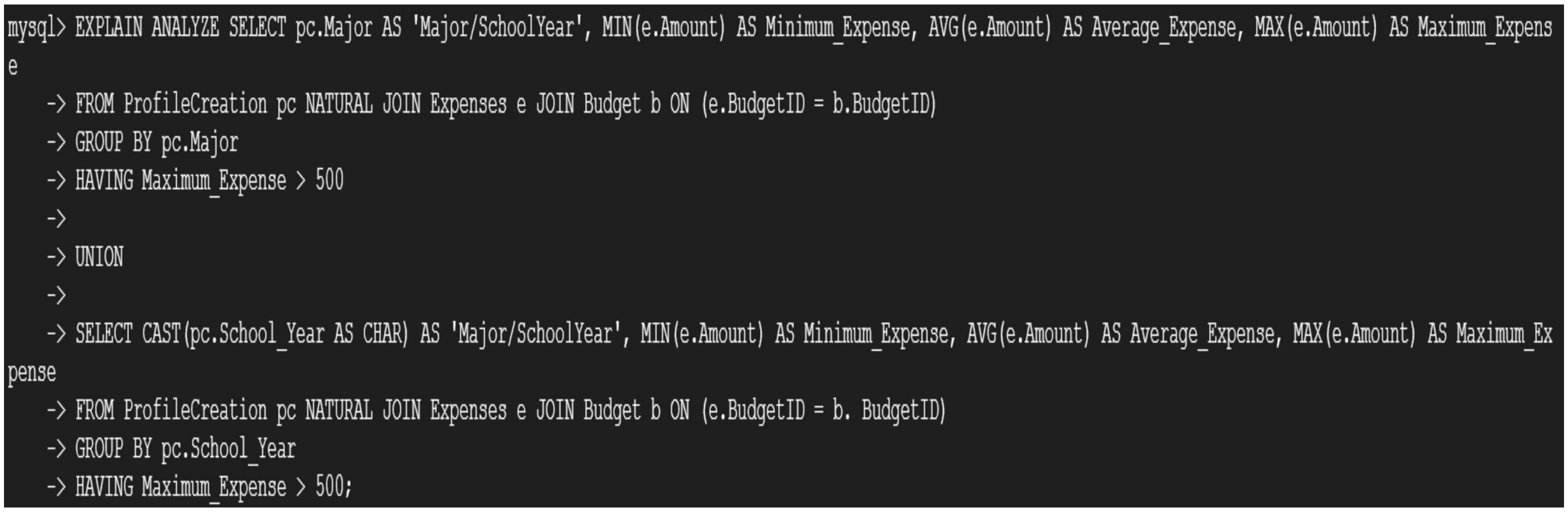
We created an index on User_Account.FirstName and User_Account.Email, but it did not improve the cost of the query, which is most likely caused by the fact that the advanced query has a lot of conditions in it to filter out records that do not meet the criteria. This most likely causes the indexing to not have a large impact on the cost.

Final Design (Query 1):

After creating and testing multiple indexes on the four tables used in this query (Expenses, Monthly_Plans, Budget, and User_Account) based on attributes used in the GROUP BY that were not primary keys, the explain analyze results show that there was no change in cost for this query. Due to this, the design remains the same as it was without the attempted indexing.

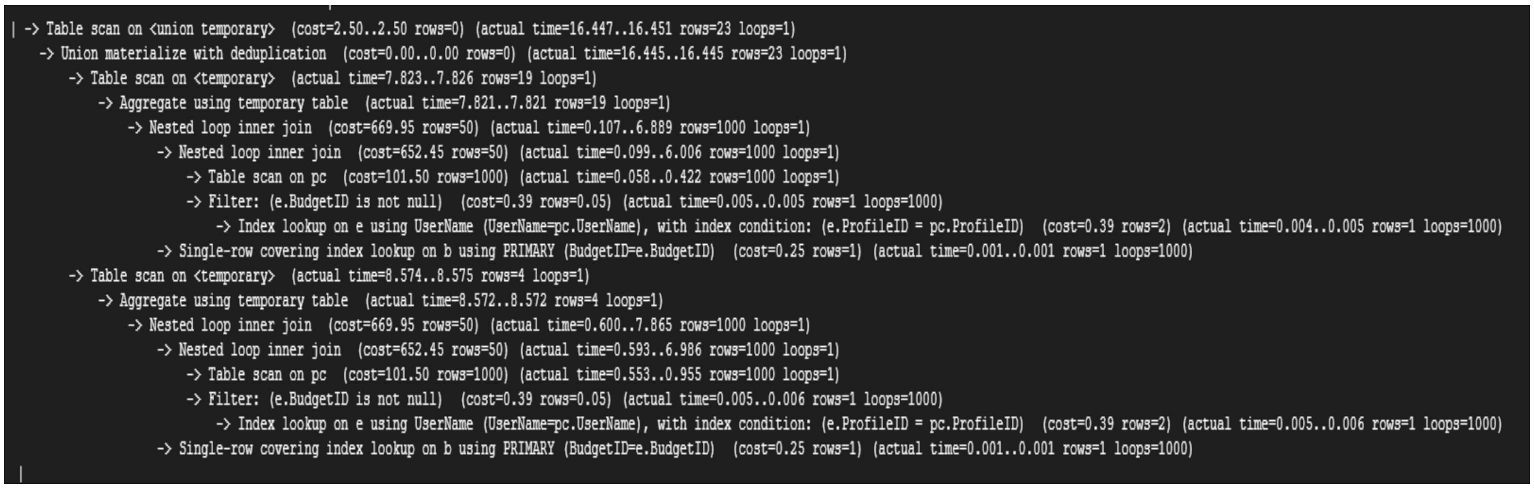
Query 2:

EXPLAIN ANALYZE Command:



```
mysql> EXPLAIN ANALYZE SELECT pc.Major AS 'Major/SchoolYear', MIN(e.Amount) AS Minimum_Expense, AVG(e.Amount) AS Average_Expense, MAX(e.Amount) AS Maximum_Expens
e
    -> FROM ProfileCreation pc NATURAL JOIN Expenses e JOIN Budget b ON (e.BudgetID = b.BudgetID)
    -> GROUP BY pc.Major
    -> HAVING Maximum_Expense > 500
    ->
    -> UNION
    ->
    -> SELECT CAST(pc.School_Year AS CHAR) AS 'Major/SchoolYear', MIN(e.Amount) AS Minimum_Expense, AVG(e.Amount) AS Average_Expense, MAX(e.Amount) AS Maximum_Ex
pense
    -> FROM ProfileCreation pc NATURAL JOIN Expenses e JOIN Budget b ON (e.BudgetID = b. BudgetID)
    -> GROUP BY pc.School_Year
    -> HAVING Maximum_Expense > 500;
```

Without Indexing:

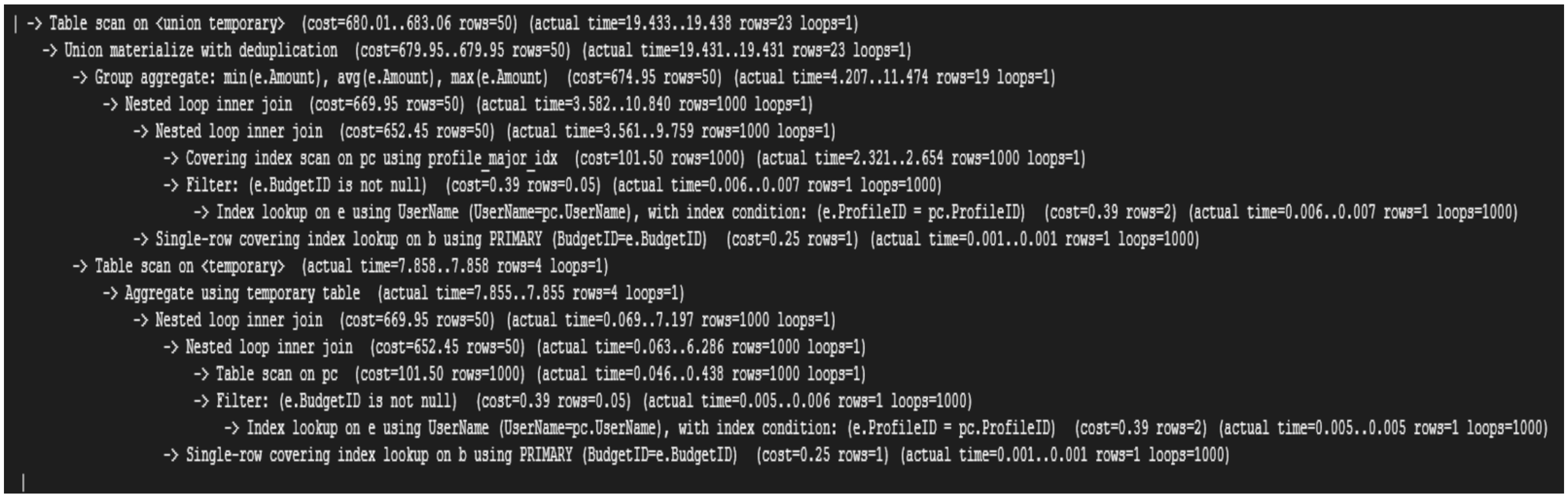


```
| -> Table scan on <union temporary>  (cost=2.50..2.50 rows=0) (actual time=16.447..16.451 rows=23 loops=1)
    -> Union materialize with deduplication  (cost=0.00..0.00 rows=0) (actual time=16.445..16.445 rows=23 loops=1)
        -> Table scan on <temporary>  (actual time=7.823..7.826 rows=19 loops=1)
            -> Aggregate using temporary table  (actual time=7.821..7.821 rows=19 loops=1)
                -> Nested loop inner join  (cost=669.95 rows=50) (actual time=0.107..6.889 rows=1000 loops=1)
                    -> Nested loop inner join  (cost=652.45 rows=50) (actual time=0.099..6.006 rows=1000 loops=1)
                        -> Table scan on pc  (cost=101.50 rows=1000) (actual time=0.058..0.422 rows=1000 loops=1)
                        -> Filter: (e.BudgetID is not null)  (cost=0.39 rows=0.05) (actual time=0.005..0.005 rows=1 loops=1000)
                            -> Index lookup on e using UserName (UserName=pc.UserName), with index condition: (e.ProfileID = pc.ProfileID)  (cost=0.39 rows=2) (actual time=0.004..0.005 rows=1 loops=1000)
                    -> Single-row covering index lookup on b using PRIMARY (BudgetID=e.BudgetID)  (cost=0.25 rows=1) (actual time=0.001..0.001 rows=1 loops=1000)
        -> Table scan on <temporary>  (actual time=8.574..8.575 rows=4 loops=1)
            -> Aggregate using temporary table  (actual time=8.572..8.572 rows=4 loops=1)
                -> Nested loop inner join  (cost=669.95 rows=50) (actual time=0.600..7.865 rows=1000 loops=1)
                    -> Nested loop inner join  (cost=652.45 rows=50) (actual time=0.593..6.986 rows=1000 loops=1)
                        -> Table scan on pc  (cost=101.50 rows=1000) (actual time=0.553..0.955 rows=1000 loops=1)
                        -> Filter: (e.BudgetID is not null)  (cost=0.39 rows=0.05) (actual time=0.005..0.006 rows=1 loops=1000)
                            -> Index lookup on e using UserName (UserName=pc.UserName), with index condition: (e.ProfileID = pc.ProfileID)  (cost=0.39 rows=2) (actual time=0.005..0.006 rows=1 loops=1000)
                    -> Single-row covering index lookup on b using PRIMARY (BudgetID=e.BudgetID)  (cost=0.25 rows=1) (actual time=0.001..0.001 rows=1 loops=1000)
 |
```

Without indexing, there is a nested loop inner join that occurs with a cost of 671.05 and a table scan on union temporary that has a cost of 2.50.

Indexing On ProfileCreation.Major:

CREATE INDEX query2_major_idx ON ProfileCreation(Major);

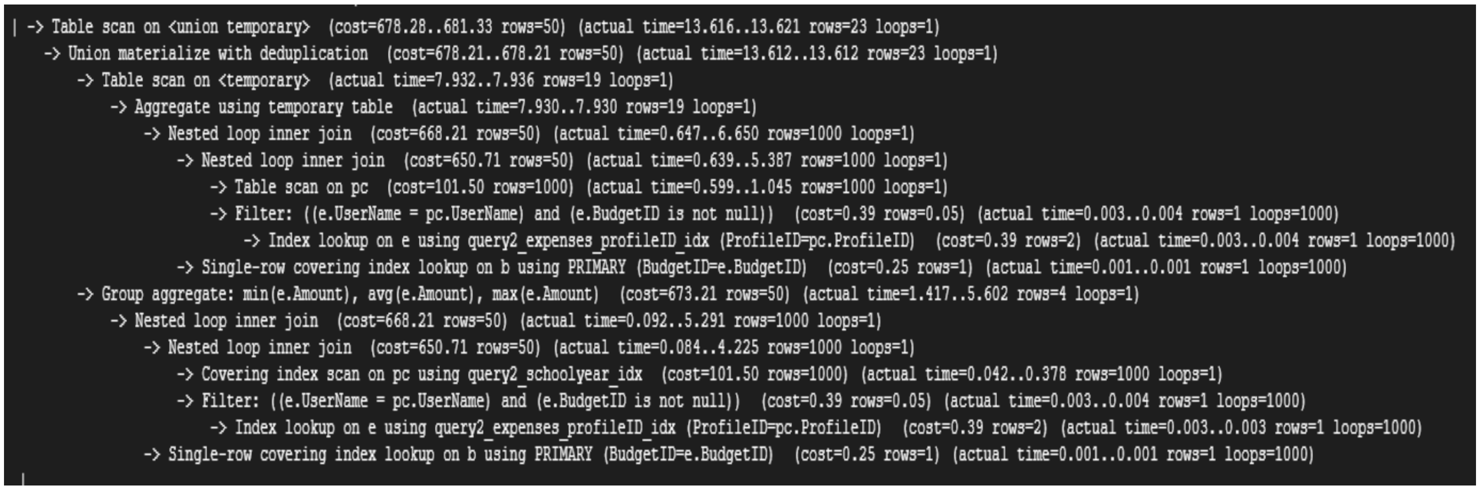


```
| -> Table scan on <union temporary>  (cost=680.01..683.06 rows=50) (actual time=19.433..19.438 rows=23 loops=1)
    -> Union materialize with deduplication  (cost=679.95..679.95 rows=50) (actual time=19.431..19.431 rows=23 loops=1)
        -> Group aggregate: min(e.Amount), avg(e.Amount), max(e.Amount)  (cost=674.95 rows=50) (actual time=4.207..11.474 rows=19 loops=1)
            -> Nested loop inner join  (cost=669.95 rows=50) (actual time=3.582..10.840 rows=1000 loops=1)
                -> Nested loop inner join  (cost=652.45 rows=50) (actual time=3.561..9.759 rows=1000 loops=1)
                    -> Covering index scan on pc using profile_major_idx  (cost=101.50 rows=1000) (actual time=2.321..2.654 rows=1000 loops=1)
                    -> Filter: (e.BudgetID is not null)  (cost=0.39 rows=0.05) (actual time=0.006..0.007 rows=1 loops=1000)
                        -> Index lookup on e using UserName (UserName=pc.UserName), with index condition: (e.ProfileID = pc.ProfileID)  (cost=0.39 rows=2) (actual time=0.006..0.007 rows=1 loops=1000)
                -> Single-row covering index lookup on b using PRIMARY (BudgetID=e.BudgetID)  (cost=0.25 rows=1) (actual time=0.001..0.001 rows=1 loops=1000)
        -> Table scan on <temporary>  (actual time=7.858..7.858 rows=4 loops=1)
            -> Aggregate using temporary table  (actual time=7.855..7.855 rows=4 loops=1)
                -> Nested loop inner join  (cost=669.95 rows=50) (actual time=0.069..7.197 rows=1000 loops=1)
                    -> Nested loop inner join  (cost=652.45 rows=50) (actual time=0.063..6.286 rows=1000 loops=1)
                        -> Table scan on pc  (cost=101.50 rows=1000) (actual time=0.046..0.438 rows=1000 loops=1)
                        -> Filter: (e.BudgetID is not null)  (cost=0.39 rows=0.05) (actual time=0.005..0.006 rows=1 loops=1000)
                            -> Index lookup on e using UserName (UserName=pc.UserName), with index condition: (e.ProfileID = pc.ProfileID)  (cost=0.39 rows=2) (actual time=0.005..0.005 rows=1 loops=1000)
                    -> Single-row covering index lookup on b using PRIMARY (BudgetID=e.BudgetID)  (cost=0.25 rows=1) (actual time=0.001..0.001 rows=1 loops=1000)
 |
```

With indexing on ProfileCreation.Major, there seems to be an overall increase in cost compared to having no index. The table scan on union temporary cost rose all the way to 680.01 while the other costs such as the nested loop inner join costs seem to have stayed the same. Additionally, a new cost of union materialized deduplicates appears with the index that did not exist when there was no index. Due to the use of the UNION operator, creating an index that affects only the first half of the query may cause the table scan on union temporary cost to increase compared to the cost of having no index.

Indexing On ProfileCreation.SchoolYear:

CREATE INDEX query2_schoolyear_idx ON ProfileCreation(School_Year);



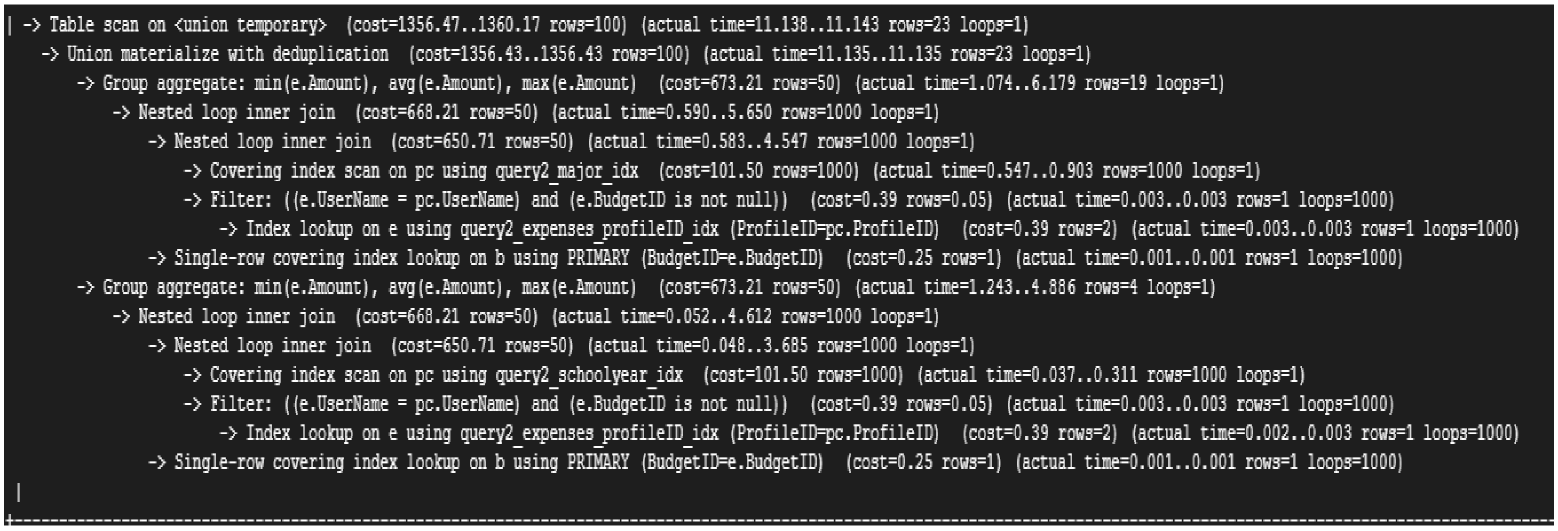
```
| -> Table scan on <union temporary>  (cost=678.28..681.33 rows=50) (actual time=13.616..13.621 rows=23 loops=1)
    -> Union materialize with deduplication  (cost=678.21..678.21 rows=50) (actual time=13.612..13.612 rows=23 loops=1)
        -> Table scan on <temporary>  (actual time=7.932..7.936 rows=19 loops=1)
            -> Aggregate using temporary table  (actual time=7.930..7.930 rows=19 loops=1)
                -> Nested loop inner join  (cost=668.21 rows=50) (actual time=0.647..6.650 rows=1000 loops=1)
                    -> Nested loop inner join  (cost=650.71 rows=50) (actual time=0.639..5.387 rows=1000 loops=1)
                        -> Table scan on pc  (cost=101.50 rows=1000) (actual time=0.599..1.045 rows=1000 loops=1)
                        -> Filter: ((e.UserName = pc.UserName) and (e.BudgetID is not null))  (cost=0.39 rows=0.05) (actual time=0.003..0.004 rows=1 loops=1000)
                            -> Index lookup on e using query2_expenses_profileID_idx (ProfileID=pc.ProfileID)  (cost=0.39 rows=2) (actual time=0.003..0.004 rows=1 loops=1000)
                    -> Single-row covering index lookup on b using PRIMARY (BudgetID=e.BudgetID)  (cost=0.25 rows=1) (actual time=0.001..0.001 rows=1 loops=1000)
        -> Group aggregate: min(e.Amount), avg(e.Amount), max(e.Amount)  (cost=673.21 rows=50) (actual time=1.417..5.602 rows=4 loops=1)
            -> Nested loop inner join  (cost=668.21 rows=50) (actual time=0.092..5.291 rows=1000 loops=1)
                -> Nested loop inner join  (cost=650.71 rows=50) (actual time=0.084..4.225 rows=1000 loops=1)
                    -> Covering index scan on pc using query2_schoolyear_idx  (cost=101.50 rows=1000) (actual time=0.042..0.378 rows=1000 loops=1)
                    -> Filter: ((e.UserName = pc.UserName) and (e.BudgetID is not null))  (cost=0.39 rows=0.05) (actual time=0.003..0.004 rows=1 loops=1000)
                        -> Index lookup on e using query2_expenses_profileID_idx (ProfileID=pc.ProfileID)  (cost=0.39 rows=2) (actual time=0.003..0.003 rows=1 loops=1000)
                -> Single-row covering index lookup on b using PRIMARY (BudgetID=e.BudgetID)  (cost=0.25 rows=1) (actual time=0.001..0.001 rows=1 loops=1000)
 |
```

Indexing on the attribute School_Year from the ProfileCreation table seems to have slightly better performance than indexing on Major from the ProfileCreation table, but the overall cost is still much higher than without any indexing in terms of the table scan on union temporary cost. The union materialized with deduplication also increased significantly compared to the cost when no indexing was done.The cost in other parts such as the nested loop inner joins seems to be relatively similar to the cost without indexing. Like stated above, this may be due to the fact that there is a UNION operator, but only the bottom half of the query is affected by the indexing.

Indexing On ProfileCreation.SchoolYear And ProfileCreation.Major:

CREATE INDEX query2_major_idx ON ProfileCreation(Major);

CREATE INDEX query2_schoolyear_idx ON ProfileCreation(School_Year);

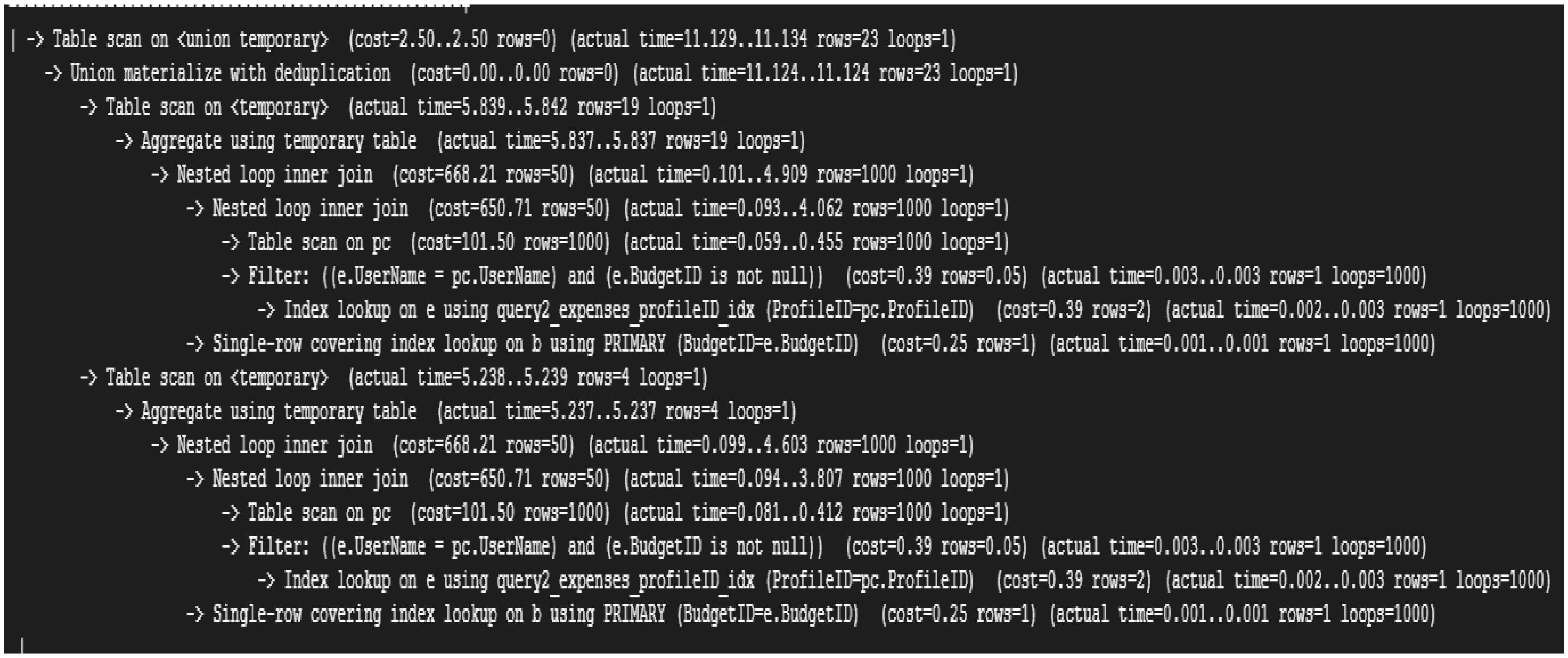


```
| -> Table scan on <union temporary>  (cost=1356.47..1360.17 rows=100) (actual time=11.138..11.143 rows=23 loops=1)
    -> Union materialize with deduplication  (cost=1356.43..1356.43 rows=100) (actual time=11.135..11.135 rows=23 loops=1)
        -> Group aggregate: min(e.Amount), avg(e.Amount), max(e.Amount)  (cost=673.21 rows=50) (actual time=1.074..6.179 rows=19 loops=1)
            -> Nested loop inner join  (cost=668.21 rows=50) (actual time=0.590..5.650 rows=1000 loops=1)
                -> Nested loop inner join  (cost=650.71 rows=50) (actual time=0.583..4.547 rows=1000 loops=1)
                    -> Covering index scan on pc using query2_major_idx  (cost=101.50 rows=1000) (actual time=0.547..0.903 rows=1000 loops=1)
                    -> Filter: ((e.UserName = pc.UserName) and (e.BudgetID is not null))  (cost=0.39 rows=0.05) (actual time=0.003..0.003 rows=1 loops=1000)
                        -> Index lookup on e using query2_expenses_profileID_idx (ProfileID=pc.ProfileID)  (cost=0.39 rows=2) (actual time=0.003..0.003 rows=1 loops=1000)
                -> Single-row covering index lookup on b using PRIMARY (BudgetID=e.BudgetID)  (cost=0.25 rows=1) (actual time=0.001..0.001 rows=1 loops=1000)
        -> Group aggregate: min(e.Amount), avg(e.Amount), max(e.Amount)  (cost=673.21 rows=50) (actual time=1.243..4.886 rows=4 loops=1)
            -> Nested loop inner join  (cost=668.21 rows=50) (actual time=0.052..4.612 rows=1000 loops=1)
                -> Nested loop inner join  (cost=650.71 rows=50) (actual time=0.048..3.685 rows=1000 loops=1)
                    -> Covering index scan on pc using query2_schoolyear_idx  (cost=101.50 rows=1000) (actual time=0.037..0.311 rows=1000 loops=1)
                    -> Filter: ((e.UserName = pc.UserName) and (e.BudgetID is not null))  (cost=0.39 rows=0.05) (actual time=0.003..0.003 rows=1 loops=1000)
                        -> Index lookup on e using query2_expenses_profileID_idx (ProfileID=pc.ProfileID)  (cost=0.39 rows=2) (actual time=0.002..0.003 rows=1 loops=1000)
                -> Single-row covering index lookup on b using PRIMARY (BudgetID=e.BudgetID)  (cost=0.25 rows=1) (actual time=0.001..0.001 rows=1 loops=1000)
 |
+----------------------------------------------------------------------------------------------------------------------------------------------------------------------------
```

Having an index on both Major and School_Year in the ProfileCreation table actually led to a significant increase in costs for the table scan on union temporary and the union materialize with deduplication, with a cost of 1356.47 and 1356.43 respectively. This is most likely due to the fact that creating an index that affects both halves of the query and then using the UNION operator causes the cost to increase by a significant amount. This makes sense since creating an index that only affected one half of the query but not the other as shown above increased the cost by a large amount, but not as large as creating an index on both attributes that are part of the GROUP BY statement in both subqueries before the UNION occurs.

With Indexing On Expenses.Amount:

CREATE INDEX query2_amount_idx ON Expenses(Amount);



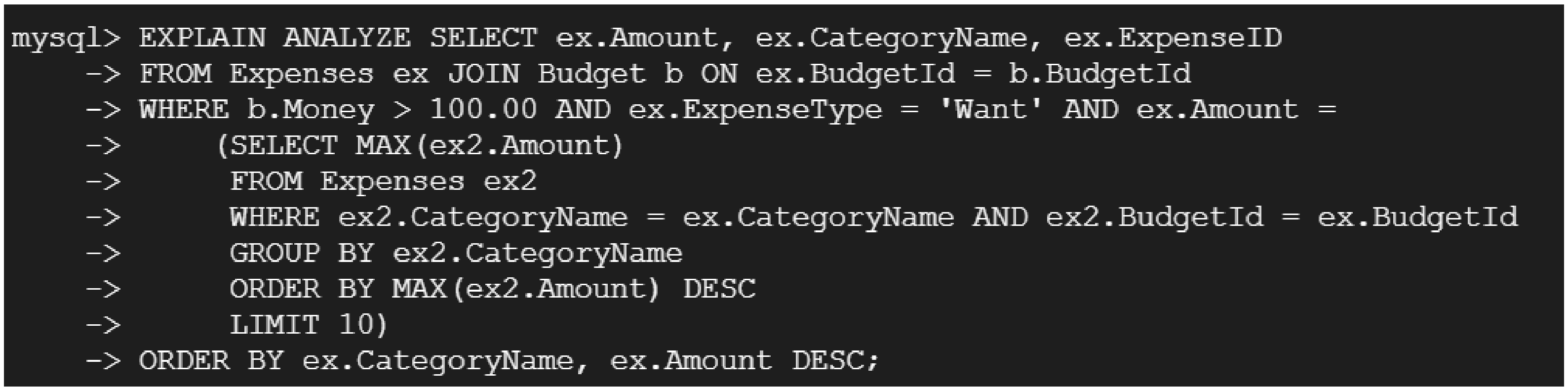
```
-----------------------------------------------------+
| -> Table scan on <union temporary>  (cost=2.50..2.50 rows=0) (actual time=11.129..11.134 rows=23 loops=1)
    -> Union materialize with deduplication  (cost=0.00..0.00 rows=0) (actual time=11.124..11.124 rows=23 loops=1)
        -> Table scan on <temporary>  (actual time=5.839..5.842 rows=19 loops=1)
            -> Aggregate using temporary table  (actual time=5.837..5.837 rows=19 loops=1)
                -> Nested loop inner join  (cost=668.21 rows=50) (actual time=0.101..4.909 rows=1000 loops=1)
                    -> Nested loop inner join  (cost=650.71 rows=50) (actual time=0.093..4.062 rows=1000 loops=1)
                        -> Table scan on pc  (cost=101.50 rows=1000) (actual time=0.059..0.455 rows=1000 loops=1)
                        -> Filter: ((e.UserName = pc.UserName) and (e.BudgetID is not null))  (cost=0.39 rows=0.05) (actual time=0.003..0.003 rows=1 loops=1000)
                            -> Index lookup on e using query2_expenses_profileID_idx (ProfileID=pc.ProfileID)  (cost=0.39 rows=2) (actual time=0.002..0.003 rows=1 loops=1000)
                    -> Single-row covering index lookup on b using PRIMARY (BudgetID=e.BudgetID)  (cost=0.25 rows=1) (actual time=0.001..0.001 rows=1 loops=1000)
        -> Table scan on <temporary>  (actual time=5.238..5.239 rows=4 loops=1)
            -> Aggregate using temporary table  (actual time=5.237..5.237 rows=4 loops=1)
                -> Nested loop inner join  (cost=668.21 rows=50) (actual time=0.099..4.603 rows=1000 loops=1)
                    -> Nested loop inner join  (cost=650.71 rows=50) (actual time=0.094..3.807 rows=1000 loops=1)
                        -> Table scan on pc  (cost=101.50 rows=1000) (actual time=0.081..0.412 rows=1000 loops=1)
                        -> Filter: ((e.UserName = pc.UserName) and (e.BudgetID is not null))  (cost=0.39 rows=0.05) (actual time=0.003..0.003 rows=1 loops=1000)
                            -> Index lookup on e using query2_expenses_profileID_idx (ProfileID=pc.ProfileID)  (cost=0.39 rows=2) (actual time=0.002..0.003 rows=1 loops=1000)
                    -> Single-row covering index lookup on b using PRIMARY (BudgetID=e.BudgetID)  (cost=0.25 rows=1) (actual time=0.001..0.001 rows=1 loops=1000)
 |
```

Creating an index for the Amount attribute from the Expenses table seemed to have lowered the cost by a small amount compared to the costs when there was no index. The table scan on union temporary cost is the exact same as the cost without indexing, but the nested loop inner join costs of 668.21 and 650.71 is a slight improvement compared to the costs for the same categories for running the query without an index. This is most likely due to the fact that the Amount attribute has a large range of possible values within it, leading to an overall lowering of the cost in some parts of the query as a result of the indexing.

Final Index Design (Query 2):
The final index design chosen for Query 2 will be to create an index on the Amount attribute on the Expenses table. This is due to the fact that Amount is in the HAVING clause within the advanced query and indexing on the other attributes tested above led to higher costs compared to the costs without any indexing. Only when indexing the Amount attribute the overall cost of running Query 2 decreased, even if the lowering of the cost was not that large.

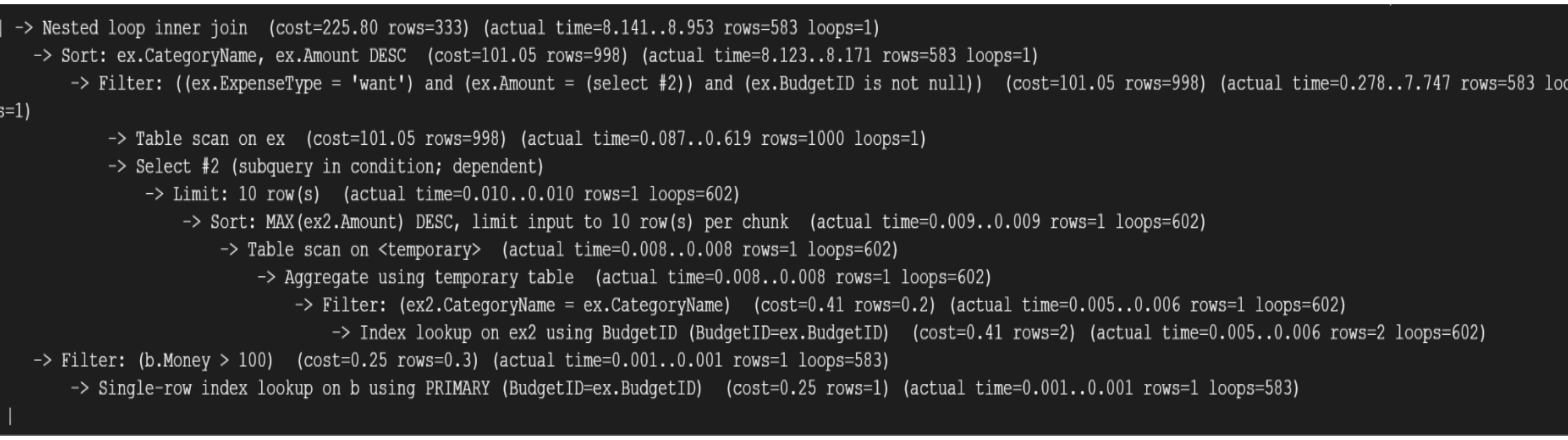
Query 3:

EXPLAIN ANALYZE Command:



```
mysql> EXPLAIN ANALYZE SELECT ex.Amount, ex.CategoryName, ex.ExpenseID
    -> FROM Expenses ex JOIN Budget b ON ex.BudgetId = b.BudgetId
    -> WHERE b.Money > 100.00 AND ex.ExpenseType = 'Want' AND ex.Amount =
    ->     (SELECT MAX(ex2.Amount)
    ->      FROM Expenses ex2
    ->      WHERE ex2.CategoryName = ex.CategoryName AND ex2.BudgetId = ex.BudgetId
    ->      GROUP BY ex2.CategoryName
    ->      ORDER BY MAX(ex2.Amount) DESC
    ->      LIMIT 10)
    -> ORDER BY ex.CategoryName, ex.Amount DESC;
```

Without Indexing:

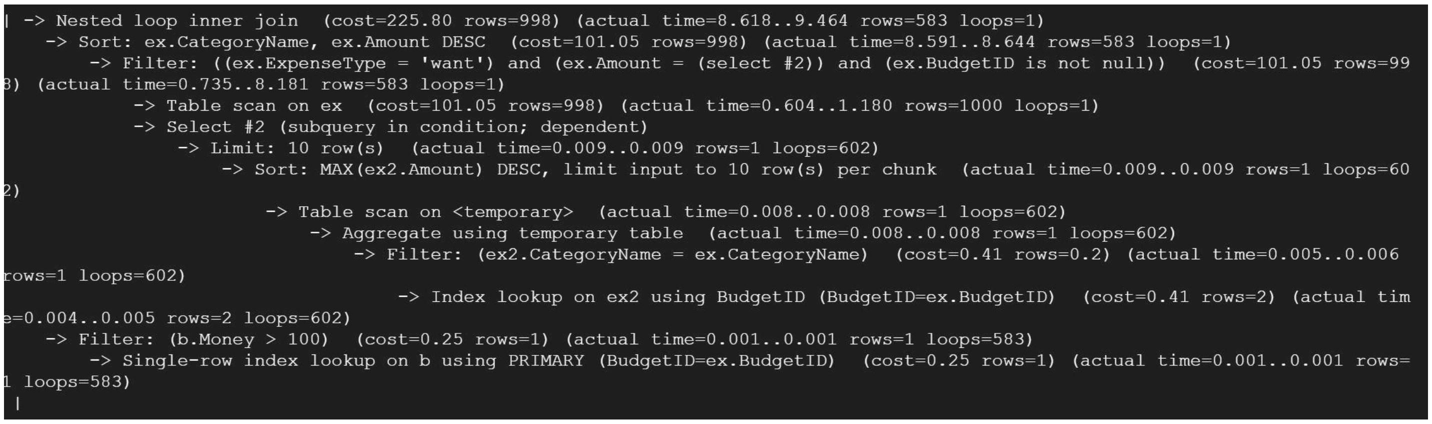


```
| -> Nested loop inner join  (cost=225.80 rows=333) (actual time=8.141..8.953 rows=583 loops=1)
    -> Sort: ex.CategoryName, ex.Amount DESC  (cost=101.05 rows=998) (actual time=8.123..8.171 rows=583 loops=1)
        -> Filter: ((ex.ExpenseType = 'want') and (ex.Amount = (select #2)) and (ex.BudgetID is not null))  (cost=101.05 rows=998) (actual time=0.278..7.747 rows=583 loo
s=1)
            -> Table scan on ex  (cost=101.05 rows=998) (actual time=0.087..0.619 rows=1000 loops=1)
            -> Select #2 (subquery in condition; dependent)
                -> Limit: 10 row(s)  (actual time=0.010..0.010 rows=1 loops=602)
                    -> Sort: MAX(ex2.Amount) DESC, limit input to 10 row(s) per chunk  (actual time=0.009..0.009 rows=1 loops=602)
                        -> Table scan on <temporary>  (actual time=0.008..0.008 rows=1 loops=602)
                            -> Aggregate using temporary table  (actual time=0.008..0.008 rows=1 loops=602)
                                -> Filter: (ex2.CategoryName = ex.CategoryName)  (cost=0.41 rows=0.2) (actual time=0.005..0.006 rows=1 loops=602)
                                    -> Index lookup on ex2 using BudgetID (BudgetID=ex.BudgetID)  (cost=0.41 rows=2) (actual time=0.005..0.006 rows=2 loops=602)
    -> Filter: (b.Money > 100)  (cost=0.25 rows=0.3) (actual time=0.001..0.001 rows=1 loops=583)
        -> Single-row index lookup on b using PRIMARY (BudgetID=ex.BudgetID)  (cost=0.25 rows=1) (actual time=0.001..0.001 rows=1 loops=583)
 |
```

Without indexing the nested loop for the inner join costs 225.80 with 333 rows. It then sorts the table through CategoryName, and Amount in descending order which costs 101.05 with 998 rows. It then filters by ExpenseType = ‘want’ and Amount which costs 101.05 with 998 rows. It proceeds by scanning the table and limiting the rows to 10, to get the top 10 expenses. It sorts again to find the max amount of expenses and scans the table again using a temporary table and finally, filters by CategoryName and does an index loop on the temporary ex2 table using BudgetID.

Index on CategoryName

- CREATE INDEX ex_categoryName_idx on Expenses(CategoryName);

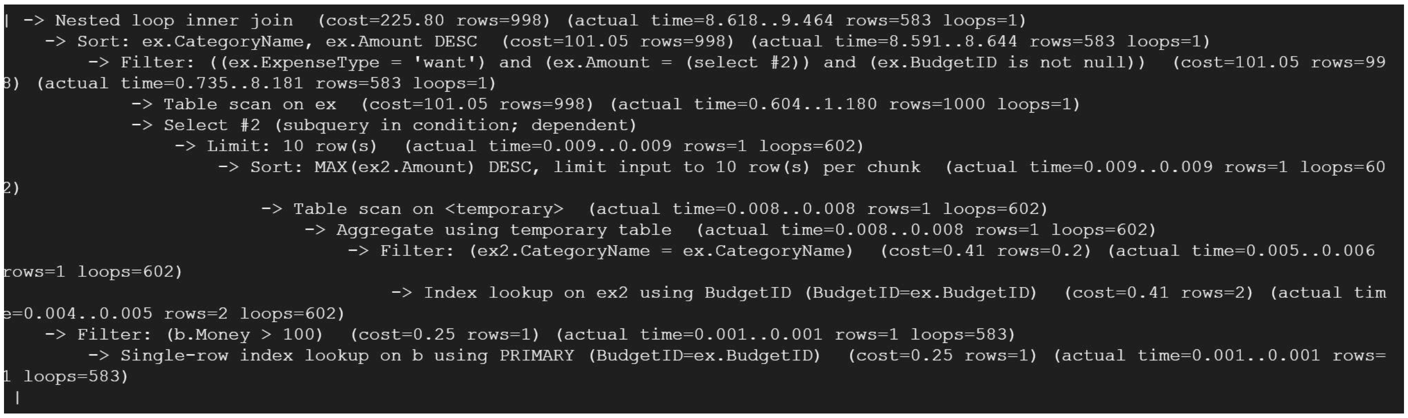


```
| -> Nested loop inner join  (cost=225.80 rows=998) (actual time=8.618..9.464 rows=583 loops=1)
    -> Sort: ex.CategoryName, ex.Amount DESC  (cost=101.05 rows=998) (actual time=8.591..8.644 rows=583 loops=1)
        -> Filter: ((ex.ExpenseType = 'want') and (ex.Amount = (select #2)) and (ex.BudgetID is not null))  (cost=101.05 rows=99
8) (actual time=0.735..8.181 rows=583 loops=1)
            -> Table scan on ex  (cost=101.05 rows=998) (actual time=0.604..1.180 rows=1000 loops=1)
            -> Select #2 (subquery in condition; dependent)
                -> Limit: 10 row(s)  (actual time=0.009..0.009 rows=1 loops=602)
                    -> Sort: MAX(ex2.Amount) DESC, limit input to 10 row(s) per chunk  (actual time=0.009..0.009 rows=1 loops=60
2)
                        -> Table scan on <temporary>  (actual time=0.008..0.008 rows=1 loops=602)
                            -> Aggregate using temporary table  (actual time=0.008..0.008 rows=1 loops=602)
                                -> Filter: (ex2.CategoryName = ex.CategoryName)  (cost=0.41 rows=0.2) (actual time=0.005..0.006
rows=1 loops=602)
                                    -> Index lookup on ex2 using BudgetID (BudgetID=ex.BudgetID)  (cost=0.41 rows=2) (actual tim
e=0.004..0.005 rows=2 loops=602)
    -> Filter: (b.Money > 100)  (cost=0.25 rows=1) (actual time=0.001..0.001 rows=1 loops=583)
        -> Single-row index lookup on b using PRIMARY (BudgetID=ex.BudgetID)  (cost=0.25 rows=1) (actual time=0.001..0.001 rows=
1 loops=583)
 |
```

We created an Index on CategoryName, but it did not improve the cost of the initial query, most likely due to the amount of records that are being returned by this query, making the index not have a large enough impact on the overall cost of using this advanced query.

Index on Amount

- CREATE INDEX ex_amount_idx on Expenses(Amount);

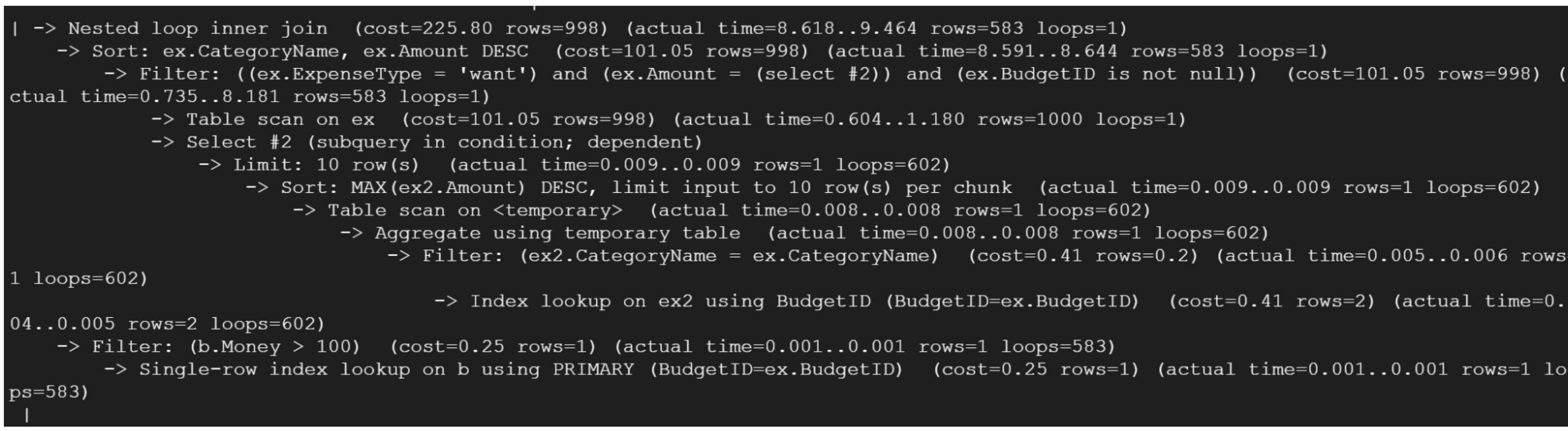


```
| -> Nested loop inner join  (cost=225.80 rows=998) (actual time=8.618..9.464 rows=583 loops=1)
    -> Sort: ex.CategoryName, ex.Amount DESC  (cost=101.05 rows=998) (actual time=8.591..8.644 rows=583 loops=1)
        -> Filter: ((ex.ExpenseType = 'want') and (ex.Amount = (select #2)) and (ex.BudgetID is not null))  (cost=101.05 rows=99
8) (actual time=0.735..8.181 rows=583 loops=1)
            -> Table scan on ex  (cost=101.05 rows=998) (actual time=0.604..1.180 rows=1000 loops=1)
            -> Select #2 (subquery in condition; dependent)
                -> Limit: 10 row(s)  (actual time=0.009..0.009 rows=1 loops=602)
                    -> Sort: MAX(ex2.Amount) DESC, limit input to 10 row(s) per chunk  (actual time=0.009..0.009 rows=1 loops=60
2)
                        -> Table scan on <temporary>  (actual time=0.008..0.008 rows=1 loops=602)
                            -> Aggregate using temporary table  (actual time=0.008..0.008 rows=1 loops=602)
                                -> Filter: (ex2.CategoryName = ex.CategoryName)  (cost=0.41 rows=0.2) (actual time=0.005..0.006
rows=1 loops=602)
                                    -> Index lookup on ex2 using BudgetID (BudgetID=ex.BudgetID)  (cost=0.41 rows=2) (actual tim
e=0.004..0.005 rows=2 loops=602)
    -> Filter: (b.Money > 100)  (cost=0.25 rows=1) (actual time=0.001..0.001 rows=1 loops=583)
        -> Single-row index lookup on b using PRIMARY (BudgetID=ex.BudgetID)  (cost=0.25 rows=1) (actual time=0.001..0.001 rows=
1 loops=583)
 |
```

We created another index on Amount, but it did not improve the cost of the initial query, most likely due to the amount of records that are being returned by this query, making the index not have a large enough impact on the overall cost of using this advanced query.

Index on Money

- CREATE INDEX ex_money_idx on Budget(Money);



```
| -> Nested loop inner join  (cost=225.80 rows=998) (actual time=8.618..9.464 rows=583 loops=1)
    -> Sort: ex.CategoryName, ex.Amount DESC  (cost=101.05 rows=998) (actual time=8.591..8.644 rows=583 loops=1)
        -> Filter: ((ex.ExpenseType = 'want') and (ex.Amount = (select #2)) and (ex.BudgetID is not null))  (cost=101.05 rows=998) (
ctual time=0.735..8.181 rows=583 loops=1)
            -> Table scan on ex  (cost=101.05 rows=998) (actual time=0.604..1.180 rows=1000 loops=1)
            -> Select #2 (subquery in condition; dependent)
                -> Limit: 10 row(s)  (actual time=0.009..0.009 rows=1 loops=602)
                    -> Sort: MAX(ex2.Amount) DESC, limit input to 10 row(s) per chunk  (actual time=0.009..0.009 rows=1 loops=602)
                        -> Table scan on <temporary>  (actual time=0.008..0.008 rows=1 loops=602)
                            -> Aggregate using temporary table  (actual time=0.008..0.008 rows=1 loops=602)
                                -> Filter: (ex2.CategoryName = ex.CategoryName)  (cost=0.41 rows=0.2) (actual time=0.005..0.006 rows
1 loops=602)
                                    -> Index lookup on ex2 using BudgetID (BudgetID=ex.BudgetID)  (cost=0.41 rows=2) (actual time=0.
04..0.005 rows=2 loops=602)
    -> Filter: (b.Money > 100)  (cost=0.25 rows=1) (actual time=0.001..0.001 rows=1 loops=583)
        -> Single-row index lookup on b using PRIMARY (BudgetID=ex.BudgetID)  (cost=0.25 rows=1) (actual time=0.001..0.001 rows=1 lo
ps=583)
 |
```

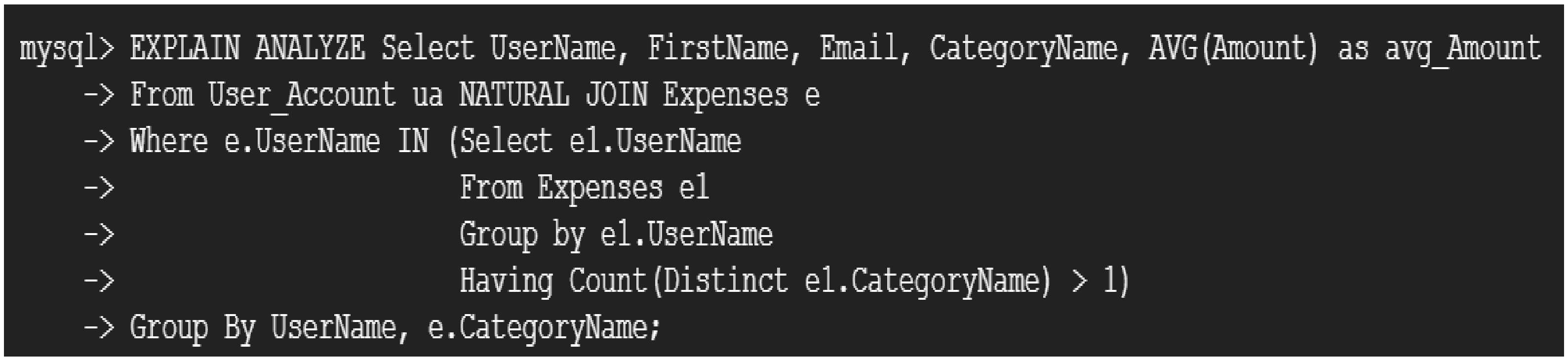
We created another index on Money, but it did not improve the cost of the initial query, most likely due to the amount of records that are being returned by this query, making the index not have a large enough impact on the overall cost of using this advanced query.

Final Design (Query 3)

After creating three indexes from attributes that are not primary keys s(CategoryName, Amount, and Money) and analyzing the advanced query, we realized that because we want to display the top 10 expenses that are classified as “wants” and are from students with a budget more than or equal to $100.00, our query cost is already low and after running each query again after creating indexes, our analysis doesn’t show a change in cost.

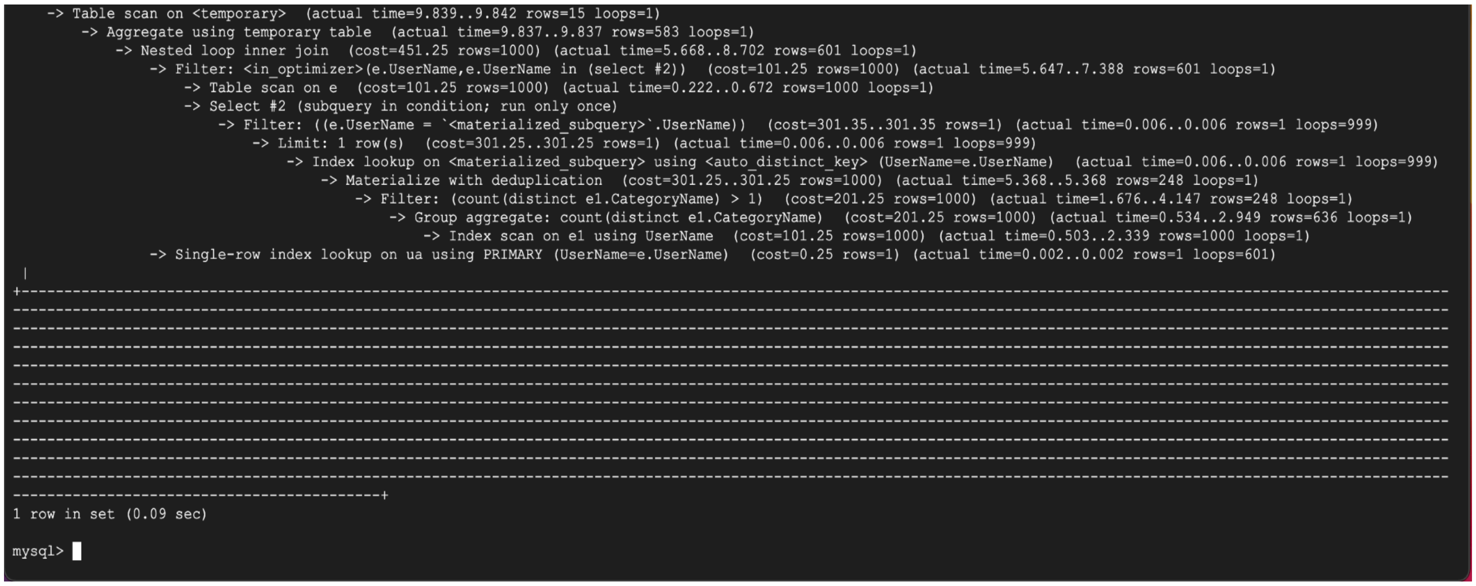
Query 4:

EXPLAIN ANALYZE Command:



```
mysql> EXPLAIN ANALYZE Select UserName, FirstName, Email, CategoryName, AVG(Amount) as avg_Amount
    -> From User_Account ua NATURAL JOIN Expenses e
    -> Where e.UserName IN (Select e1.UserName
    ->                      From Expenses e1
    ->                      Group by e1.UserName
    ->                      Having Count(Distinct e1.CategoryName) > 1)
    -> Group By UserName, e.CategoryName;
```

Without Indexing:

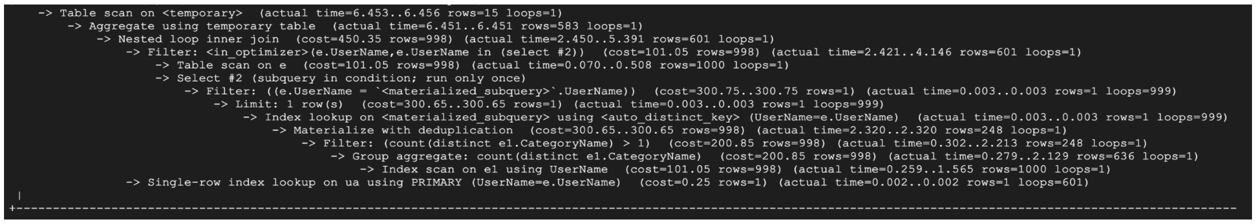


```
    -> Table scan on <temporary>  (actual time=9.839..9.842 rows=15 loops=1)
        -> Aggregate using temporary table  (actual time=9.837..9.837 rows=583 loops=1)
            -> Nested loop inner join  (cost=451.25 rows=1000) (actual time=5.668..8.702 rows=601 loops=1)
                -> Filter: <in_optimizer>(e.UserName,e.UserName in (select #2))  (cost=101.25 rows=1000) (actual time=5.647..7.388 rows=601 loops=1)
                    -> Table scan on e  (cost=101.25 rows=1000) (actual time=0.222..0.672 rows=1000 loops=1)
                    -> Select #2 (subquery in condition; run only once)
                        -> Filter: ((e.UserName = `<materialized_subquery>`.UserName))  (cost=301.35..301.35 rows=1) (actual time=0.006..0.006 rows=1 loops=999)
                            -> Limit: 1 row(s)  (cost=301.25..301.25 rows=1) (actual time=0.006..0.006 rows=1 loops=999)
                                -> Index lookup on <materialized_subquery> using <auto_distinct_key> (UserName=e.UserName)  (actual time=0.006..0.006 rows=1 loops=999)
                                    -> Materialize with deduplication  (cost=301.25..301.25 rows=1000) (actual time=5.368..5.368 rows=248 loops=1)
                                        -> Filter: (count(distinct e1.CategoryName) > 1)  (cost=201.25 rows=1000) (actual time=1.676..4.147 rows=248 loops=1)
                                            -> Group aggregate: count(distinct e1.CategoryName)  (cost=201.25 rows=1000) (actual time=0.534..2.949 rows=636 loops=1)
                                                -> Index scan on e1 using UserName  (cost=101.25 rows=1000) (actual time=0.503..2.339 rows=1000 loops=1)
                -> Single-row index lookup on ua using PRIMARY (UserName=e.UserName)  (cost=0.25 rows=1) (actual time=0.002..0.002 rows=1 loops=601)
 |
+------------------------------------------------------------------------------------------------------------------------------------------------------------------
-------------------------------------------------------------------------------------------------------------------------------------------------------------------
-------------------------------------------------------------------------------------------------------------------------------------------------------------------
-------------------------------------------------------------------------------------------------------------------------------------------------------------------
-------------------------------------------------------------------------------------------------------------------------------------------------------------------
-------------------------------------------------------------------------------------------------------------------------------------------------------------------
-------------------------------------------------------------------------------------------------------------------------------------------------------------------
-------------------------------------------------------------------------------------------------------------------------------------------------------------------
-------------------------------------------------------------------------------------------------------------------------------------------------------------------
-------------------------------------------------------------------------------------------------------------------------------------------------------------------
-------------------------------------------------------------------------------------------------------------------------------------------------------------------
-------------------------------------------+
1 row in set (0.09 sec)

mysql> 
```

- Without any indexing the cost was 451.25 in the inner join. We are not expecting the index or multiple indexes to make a great difference on the cost simply because we have written the advanced query in an efficient way that utilizes aggregation and subqueries.

Indexing On User_Account.Email:

- CREATE INDEX user_email_idx on User_Account(Email);

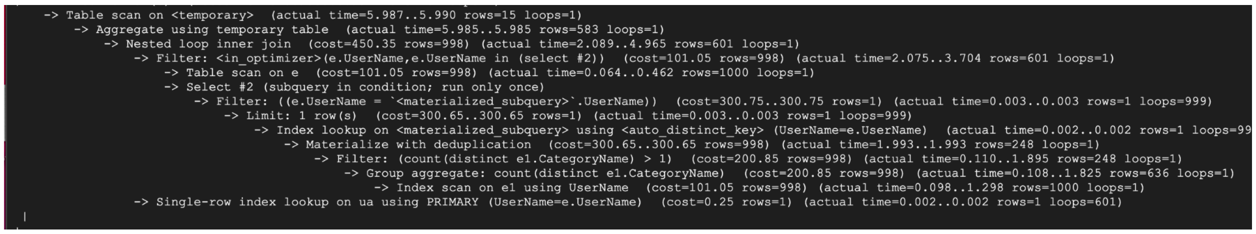


```
    -> Table scan on <temporary>  (actual time=6.453..6.456 rows=15 loops=1)
        -> Aggregate using temporary table  (actual time=6.451..6.451 rows=583 loops=1)
            -> Nested loop inner join  (cost=450.35 rows=998) (actual time=2.450..5.391 rows=601 loops=1)
                -> Filter: <in_optimizer>(e.UserName,e.UserName in (select #2))  (cost=101.05 rows=998) (actual time=2.421..4.146 rows=601 loops=1)
                    -> Table scan on e  (cost=101.05 rows=998) (actual time=0.070..0.508 rows=1000 loops=1)
                    -> Select #2 (subquery in condition; run only once)
                        -> Filter: ((e.UserName = `<materialized_subquery>`.UserName))  (cost=300.75..300.75 rows=1) (actual time=0.003..0.003 rows=1 loops=999)
                            -> Limit: 1 row(s)  (cost=300.65..300.65 rows=1) (actual time=0.003..0.003 rows=1 loops=999)
                                -> Index lookup on <materialized_subquery> using <auto_distinct_key> (UserName=e.UserName)  (actual time=0.003..0.003 rows=1 loops=999)
                                    -> Materialize with deduplication  (cost=300.65..300.65 rows=998) (actual time=2.320..2.320 rows=248 loops=1)
                                        -> Filter: (count(distinct e1.CategoryName) > 1)  (cost=200.85 rows=998) (actual time=0.302..2.213 rows=248 loops=1)
                                            -> Group aggregate: count(distinct e1.CategoryName)  (cost=200.85 rows=998) (actual time=0.279..2.129 rows=636 loops=1)
                                                -> Index scan on e1 using UserName  (cost=101.05 rows=998) (actual time=0.259..1.565 rows=1000 loops=1)
                -> Single-row index lookup on ua using PRIMARY (UserName=e.UserName)  (cost=0.25 rows=1) (actual time=0.002..0.002 rows=1 loops=601)
 |
+---------------------------------------------------------------------------------------------------------------------------------------------------------------------------
```

- When using the index on email I was able to decrease the cost by a small amount on the winner join loop. The cost decreased by slightly less than 1. From 451.25 to 450.35. This was most likely due to the fact that we are not returning many rows, we are grouping by category name, and we also have a subquery in the advanced query. All of these factors combined allude to the fact that the query itself is extremely efficient and adding an index would only make it fractionally better.

Indexing On Expenses.CategoryName:

- CREATE INDEX exp_catname_idx on Expenses(CategoryName);

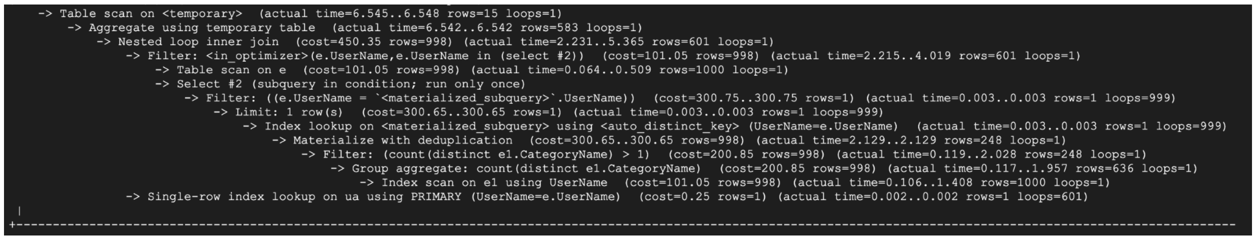


```
    -> Table scan on <temporary>  (actual time=5.987..5.990 rows=15 loops=1)
        -> Aggregate using temporary table  (actual time=5.985..5.985 rows=583 loops=1)
            -> Nested loop inner join  (cost=450.35 rows=998) (actual time=2.089..4.965 rows=601 loops=1)
                -> Filter: <in_optimizer>(e.UserName,e.UserName in (select #2))  (cost=101.05 rows=998) (actual time=2.075..3.704 rows=601 loops=1)
                    -> Table scan on e  (cost=101.05 rows=998) (actual time=0.064..0.462 rows=1000 loops=1)
                    -> Select #2 (subquery in condition; run only once)
                        -> Filter: ((e.UserName = `<materialized_subquery>`.UserName))  (cost=300.75..300.75 rows=1) (actual time=0.003..0.003 rows=1 loops=999)
                            -> Limit: 1 row(s)  (cost=300.65..300.65 rows=1) (actual time=0.003..0.003 rows=1 loops=999)
                                -> Index lookup on <materialized_subquery> using <auto_distinct_key> (UserName=e.UserName)  (actual time=0.002..0.002 rows=1 loops=99
                                    -> Materialize with deduplication  (cost=300.65..300.65 rows=998) (actual time=1.993..1.993 rows=248 loops=1)
                                        -> Filter: (count(distinct e1.CategoryName) > 1)  (cost=200.85 rows=998) (actual time=0.110..1.895 rows=248 loops=1)
                                            -> Group aggregate: count(distinct e1.CategoryName)  (cost=200.85 rows=998) (actual time=0.108..1.825 rows=636 loops=1)
                                                -> Index scan on e1 using UserName  (cost=101.05 rows=998) (actual time=0.098..1.298 rows=1000 loops=1)
                -> Single-row index lookup on ua using PRIMARY (UserName=e.UserName)  (cost=0.25 rows=1) (actual time=0.002..0.002 rows=1 loops=601)
 |
```

- Decreased slightly from the original cost, went from 451.25 to 450.35. We decided to index based on the CategoryName because it is in the group by clause. As we know the group by clause when indexing an attribute in the clause will typically result in cost decrease. In this case we noticed a small decrease but nothing significant, actually the cost decrease was the same as the double index that we used in the last example. We believe this is because of the query being a subquery combined with the group by clause that only returns data for individuals with expenses in multiple categories. Based on our created data we noticed there were individuals who only reported spending in a single category so we wanted a more holistic approach.

Indexing On User_Account.FirstName:

- CREATE INDEX user_fname_idx on User_Account(FirstName);

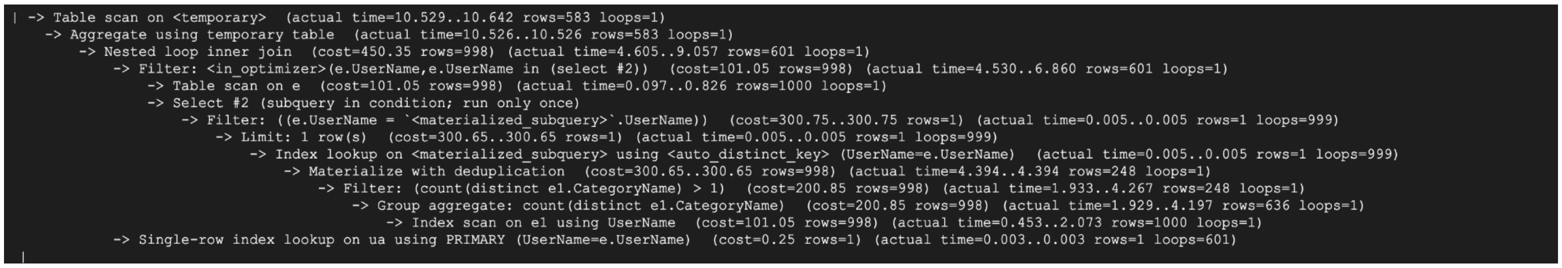


```
    -> Table scan on <temporary>  (actual time=6.545..6.548 rows=15 loops=1)
        -> Aggregate using temporary table  (actual time=6.542..6.542 rows=583 loops=1)
            -> Nested loop inner join  (cost=450.35 rows=998) (actual time=2.231..5.365 rows=601 loops=1)
                -> Filter: <in_optimizer>(e.UserName,e.UserName in (select #2))  (cost=101.05 rows=998) (actual time=2.215..4.019 rows=601 loops=1)
                    -> Table scan on e  (cost=101.05 rows=998) (actual time=0.064..0.509 rows=1000 loops=1)
                    -> Select #2 (subquery in condition; run only once)
                        -> Filter: ((e.UserName = `<materialized_subquery>`.UserName))  (cost=300.75..300.75 rows=1) (actual time=0.003..0.003 rows=1 loops=999)
                            -> Limit: 1 row(s)  (cost=300.65..300.65 rows=1) (actual time=0.003..0.003 rows=1 loops=999)
                                -> Index lookup on <materialized_subquery> using <auto_distinct_key> (UserName=e.UserName)  (actual time=0.003..0.003 rows=1 loops=999)
                                    -> Materialize with deduplication  (cost=300.65..300.65 rows=998) (actual time=2.129..2.129 rows=248 loops=1)
                                        -> Filter: (count(distinct e1.CategoryName) > 1)  (cost=200.85 rows=998) (actual time=0.119..2.028 rows=248 loops=1)
                                            -> Group aggregate: count(distinct e1.CategoryName)  (cost=200.85 rows=998) (actual time=0.117..1.957 rows=636 loops=1)
                                                -> Index scan on e1 using UserName  (cost=101.05 rows=998) (actual time=0.106..1.408 rows=1000 loops=1)
                -> Single-row index lookup on ua using PRIMARY (UserName=e.UserName)  (cost=0.25 rows=1) (actual time=0.002..0.002 rows=1 loops=601)
 |
+-------------------------------------------------------------------------------------------------------------------------------------------------------------------------
```

Decreased slightly from the original cost, went from 451.25 to 450.35. For this index we decided to create it on the attribute FirstName in the User_Account table. We decided to do this because it is not a primary key or a foreign key and it is one of the attributes which we are returning. We saw a small decrease in the cost. It is an attribute in the user account attribute and it was also in the select clause. We understand that indexes work better with attributes in the join GROUP BY or having but we wanted to test all of them including what if we index something in the select clause.

Indexing On User_Account.FirstName And User_Account.Email:

- CREATE INDEX user_fname_idx on User_Account(FirstName)
- CREATE INDEX user_email_idx on User_Account(Email)



```
| -> Table scan on <temporary>  (actual time=10.529..10.642 rows=583 loops=1)
    -> Aggregate using temporary table  (actual time=10.526..10.526 rows=583 loops=1)
        -> Nested loop inner join  (cost=450.35 rows=998) (actual time=4.605..9.057 rows=601 loops=1)
            -> Filter: <in_optimizer>(e.UserName,e.UserName in (select #2))  (cost=101.05 rows=998) (actual time=4.530..6.860 rows=601 loops=1)
                -> Table scan on e  (cost=101.05 rows=998) (actual time=0.097..0.826 rows=1000 loops=1)
                -> Select #2 (subquery in condition; run only once)
                    -> Filter: ((e.UserName = `<materialized_subquery>`.UserName))  (cost=300.75..300.75 rows=1) (actual time=0.005..0.005 rows=1 loops=999)
                        -> Limit: 1 row(s)  (cost=300.65..300.65 rows=1) (actual time=0.005..0.005 rows=1 loops=999)
                            -> Index lookup on <materialized_subquery> using <auto_distinct_key> (UserName=e.UserName)  (actual time=0.005..0.005 rows=1 loops=999)
                                -> Materialize with deduplication  (cost=300.65..300.65 rows=998) (actual time=4.394..4.394 rows=248 loops=1)
                                    -> Filter: (count(distinct e1.CategoryName) > 1)  (cost=200.85 rows=998) (actual time=1.933..4.267 rows=248 loops=1)
                                        -> Group aggregate: count(distinct e1.CategoryName)  (cost=200.85 rows=998) (actual time=1.929..4.197 rows=636 loops=1)
                                            -> Index scan on e1 using UserName  (cost=101.05 rows=998) (actual time=0.453..2.073 rows=1000 loops=1)
            -> Single-row index lookup on ua using PRIMARY (UserName=e.UserName)  (cost=0.25 rows=1) (actual time=0.003..0.003 rows=1 loops=601)
 |
```

For this index we decided to index both the FirstName and the Email the reason for this was because we did not see much of a difference in cost when going from no index to one index so we wanted to test if multiple active indexes would make a difference. Based on our results we discovered that adding multiple indexes to this advanced query did not make much of a difference when compared to the cost with an index and when compared to the single indexes there was zero difference in cost whatsoever. This is most likely due to the fact that the advanced query chooses a small amount of records that indexing will not have much of an impact for a returning dataset that is relatively small. Another reason why this is the case is because we are using a subquery in the advanced query which is the most efficient way to sort data which is what Dr.Alawini in the lecture video.

Final Index Design (Query 4):

Since all of the indexes returned the same price decrease, we could argue to use any of them but it is our decision to use the 3rd index which is the CategoryName index from the expenses table. We decided on this index because if we were to increase the amount of categories in our dataset in the Expenses table we would definitely want to index the categoryname for more efficient results. Another reason we chose the third index is because we are grouping by categoryName and in Dr.Alawani’s video lecture he emphasized the fact that indexing the attribute in the group by is the most effective way to reduce costs.