DDL Commands for creating the tables

```
Create Table Accounts(
        UserID INT,
        Username VARCHAR(255),
        Password VARCHAR(255),
        Income FLOAT,
        MinIncome FLOAT,
        MaxIncome FLOAT,
        PRIMARY KEY(UserID),
        FOREIGN KEY(MinIncome, MaxIncome)
                REFERENCES AverageExpense(MinIncome, MaxIncome)
                ON DELETE SET NULL
)

Create Table Receipts (
        ReceiptID INT,
        UserID INT,
        PurchaseDate DATE,
        Seller VARCHAR(255),
        PRIMARY KEY (ReceiptID),
        FOREIGN KEY (UserID)
                REFERENCES Accounts(UserID)
                ON DELETE CASCADE
)

Create Table Items (
        ItemID INT,
        Category VARCHAR(255),
        ReceiptID INT,
        ItemName VARCHAR(255),
        Price FLOAT,
        PRIMARY KEY (ItemID),
        FOREIGN KEY (ReceiptId)
                REFERENCES Receipts(ReceiptID)
                ON DELETE CASCADE,
        FOREIGN KEY (Category)
                REFERENCES Budget(Category)
                ON DELETE SET NULL
)
```

```
Create Table AverageExpense (
        MinIncome FLOAT,
        MaxIncome FLOAT,
        AvgExpense FLOAT,
        PRIMARY KEY (MinIncome, MaxIncome)
)


Create Table Budget (
        Category VARCHAR(255) UNIQUE,
        UserID INT,
        Budget FLOAT,
        Spent FLOAT,
        PRIMARY KEY (Category, UserId),
        FOREIGN KEY (UserID)
                REFERENCES Accounts(UserID)
                ON DELETE CASCADE
)

Create Table Contributes (
        UserID INT,
        ItemID INT,
        Percentage FLOAT,
        PRIMARY KEY (UserID, ItemID),
        FOREIGN KEY (UserID)
                REFERENCES Accounts(UserID)
                ON DELETE CASCADE,
        FOREIGN KEY (ItemID)
                REFERENCES Items(ItemID)
                ON DELETE CASCADE
)
```

Code to connect to database

gcloud sql connect wynaut --user=root

use Wynaut-Database

Screenshot of Implementation

```
mysql> show tables;
+---------------------------+
| Tables_in_Wynaut-Database |
+---------------------------+
| Accounts                  |
| AverageExpense            |
| Budget                    |
| Contributes               |
| Items                     |
| Receipts                  |
+---------------------------+
6 rows in set (0.00 sec)
```

3 databases of 1000 entries

```
mysql> Select Count(UserID) From Accounts;
+---------------+
| Count(UserID) |
+---------------+
|          1000 |
+---------------+
1 row in set (0.00 sec)
```

```
mysql> Select Count(ReceiptID) From Receipts;
+------------------+
| Count(ReceiptID) |
+------------------+
|             1000 |
+------------------+
1 row in set (0.00 sec)
```

```
mysql> Select Count(ItemID) From Items;
+---------------+
| Count(ItemID) |
+---------------+
|          1001 |
+---------------+
1 row in set (0.01 sec)
```

Advanced query ideas

Amount each user pays for a given receipt (ReceiptId = 0 as an example)

Code

```
SELECT Contributes.UserId as UserId, SUM(Items.Price * Contributes.Percentage) as Paid
FROM Contributes NATURAL JOIN Items
Where Items.ReceiptID = 0
GROUP BY Contributes.UserId;
```

Image

```
mysql> SELECT Contributes.UserId as UserId, SUM(Items.Price * Contributes.Percentage) as Paid
    -> FROM Contributes NATURAL JOIN Items
    -> Where Items.ReceiptID = 0
    -> GROUP BY Contributes.UserId
    -> LIMIT 15
    -> ;
+--------+----------------------+
| UserId | Paid                 |
+--------+----------------------+
|     15 |  0.09989999547824269 |
|     87 |    1.798200030255316 |
|    165 |    5.794199700522427 |
|    314 |   1.9979999839961522 |
|    690 |  0.29969998643472806 |
|    166 |    3.970199973392482 |
|    703 |   0.6616999955654137 |
|    777 | 0.050900000388175215 |
|    903 |   0.4072000031054017 |
|      4 |    0.899100015127658 |
|    210 |   0.7991999638259415 |
|    264 |    6.893099818253518 |
|    366 |   0.6992999869555234 |
|    455 |  0.19979999095648537 |
|    955 |  0.49949999599903805 |
+--------+----------------------+
15 rows in set (0.01 sec)
```

Indexing

Before:

```
| -> Limit: 15 row(s)  (actual time=0.186..0.189 rows=15 loops=1)
    -> Table scan on <temporary>  (actual time=0.185..0.186 rows=15 loops=1)
        -> Aggregate using temporary table  (actual time=0.183..0.183 rows=35 loops=1)
            -> Nested loop inner join  (cost=14.6 rows=34.8) (actual time=0.0636..0.155 rows=35 loops=1)
                -> Index lookup on Items using Items_ibfk_1 (ReceiptID=0)  (cost=2.45 rows=7) (actual time=0.0343..0.0357 rows=7 loops=1)
                -> Index lookup on Contributes using Contributes_ibfk_2 (ItemID=Items.ItemID)  (cost=1.32 rows=4.98) (actual time=0.0153..0.0163 rows=5 loops=7)
 |
```

Index by Items.Price (No Change):

```
| -> Table scan on <temporary>  (actual time=1.11..1.12 rows=35 loops=1)
    -> Aggregate using temporary table  (actual time=1.11..1.11 rows=35 loops=1)
        -> Nested loop inner join  (cost=14.6 rows=34.8) (actual time=0.939..1.07 rows=35 loops=1)
            -> Index lookup on Items using Items_ibfk_1 (ReceiptID=0)  (cost=2.45 rows=7) (actual time=0.875..0.877 rows=7 loops=1)
            -> Index lookup on Contributes using Contributes_ibfk_2 (ItemID=Items.ItemID)  (cost=1.32 rows=4.98) (actual time=0.0253..0.0263 rows=5 loops=7)
 |
```

Index by Contributes.Percentage (No Change):

```
| -> Table scan on <temporary>  (actual time=0.425..0.428 rows=35 loops=1)
    -> Aggregate using temporary table  (actual time=0.423..0.423 rows=35 loops=1)
        -> Nested loop inner join  (cost=14.6 rows=34.8) (actual time=0.297..0.391 rows=35 loops=1)
            -> Index lookup on Items using Items_ibfk_1 (ReceiptID=0)  (cost=2.45 rows=7) (actual time=0.265..0.266 rows=7 loops=1)
            -> Index lookup on Contributes using Contributes_ibfk_2 (ItemID=Items.ItemID)  (cost=1.32 rows=4.98) (actual time=0.0158..0.0168 rows=5 loops=7)
 |
```

Index by Items.ReceiptID (No Change):

```
| -> Table scan on <temporary>  (actual time=0.476..0.564 rows=35 loops=1)
    -> Aggregate using temporary table  (actual time=0.475..0.475 rows=35 loops=1)
        -> Nested loop inner join  (cost=14.6 rows=34.8) (actual time=0.219..0.431 rows=35 loops=1)
            -> Index lookup on Items using index_3 (ReceiptID=0)  (cost=2.45 rows=7) (actual time=0.11..0.113 rows=7 loops=1)
            -> Index lookup on Contributes using Contributes_ibfk_2 (ItemID=Items.ItemID)  (cost=1.32 rows=4.98) (actual time=0.0429..0.0445 rows=5 loops=7)
 |
```

Justification

Indexing by Items.Price and Contributes.Percentage didn’t make a difference probably due to the fact that these two attributes were used for calculations and not for querying the data. Calculations don't impact the cost of the query since only searching matters. Indexing by ReceiptID also didn’t make a difference even though there was filtering involved using the WHERE clause. This is probably because ReceiptID was a foreign key that referenced the ReceiptID in the Receipts Table which is a primary key. This may have already created an index in which Items.ReceiptID used when filtering the data.

Average item cost for each categories for a certain user (UserId = 5 as an example)

Code

EXPLAIN ANALYZE
Select Category, Avg(newPrice) as AverageItemCost
FROM (Select Category, (Price * Percentage) as newPrice
    FROM Items NATURAL JOIN Contributes
    WHERE UserID = 5) as price_subquery
GROUP BY Category
Order by AverageItemCost desc;

Image

```
mysql> Select Category, Avg(newPrice) as AverageItemCost
    -> FROM (Select Category, (Price * Percentage) as newPrice
    -> FROM Items NATURAL JOIN Contributes
    -> WHERE UserID = 5) as price_subquery
    -> GROUP BY Category
    -> Order by AverageItemCost desc;
+--------------------+---------------------+
| Category           | AverageItemCost     |
+--------------------+---------------------+
| Healthcare         |   14.77227409773545 |
| Housing            |  13.660849665673084 |
| Miscellaneous      |   6.812516826049364 |
| Transportation     |   5.058749877807375 |
| Taxes              |   3.387833255360524 |
| Food               |   3.060899914009337 |
| Utilities          |  1.9933499495401978 |
| Education          |   1.599199945944548 |
| Household Supplies |   0.515266654895246 |
| Insurance          | 0.27989999145492916 |
+--------------------+---------------------+
10 rows in set (0.01 sec)
```

Our output is less than 15 rows because we divided items into 10 possible categories.

Before any indexing:

```
mysql> EXPLAIN ANALYZE
    -> Select Category, Avg(newPrice) as AverageItemCost
    -> FROM (Select Category, (Price * Percentage) as newPrice
    -> FROM Items NATURAL JOIN Contributes
    -> WHERE UserID = 5) as price_subquery
    -> GROUP BY Category
    -> Order by AverageItemCost desc;
+--------------------------------------------------------------------------------
--------------------------------------------------------------------------------
--------------------------------------------------------------------------------
----+
| EXPLAIN


    |
+--------------------------------------------------------------------------------
--------------------------------------------------------------------------------
--------------------------------------------------------------------------------
----+
| -> Sort: AverageItemCost DESC  (actual time=0.332..0.333 rows=10 loops=1)
    -> Table scan on <temporary>  (actual time=0.308..0.31 rows=10 loops=1)
        -> Aggregate using temporary table  (actual time=0.306..0.306 rows=10 loops=1)
            -> Nested loop inner join  (cost=16.9 rows=37) (actual time=0.0822..0.237 rows=37 loops=1)
                -> Index lookup on Contributes using PRIMARY (UserID=5)  (cost=3.96 rows=37) (actual time=0.0534..0.0599 rows=37 loops=1)
                -> Single-row index lookup on Items using PRIMARY (ItemID=Contributes.ItemID)  (cost=0.253 rows=1) (actual time=0.00429..0.00432 rows=1 loops=37)
 |
+--------------------------------------------------------------------------------
```

Indexing on Items.Category (No change):

```
mysql> create index item_category on Items(Category)
    -> ;
Query OK, 0 rows affected (0.79 sec)
Records: 0  Duplicates: 0  Warnings: 0

mysql> EXPLAIN ANALYZE
    -> Select Category, Avg(newPrice) as AverageItemCost
    -> FROM (Select Category, (Price * Percentage) as newPrice
    -> FROM Items NATURAL JOIN Contributes
    -> WHERE UserID = 5) as price_subquery
    -> GROUP BY Category
    -> Order by AverageItemCost desc;
+----------------------------------------------------------------------------------------------------
----------------------------------------------------------------------------------------------------
----------------------------------------------------------------------------------------------------
--+
| EXPLAIN


  |
+----------------------------------------------------------------------------------------------------
----------------------------------------------------------------------------------------------------
----------------------------------------------------------------------------------------------------
--+
| -> Sort: AverageItemCost DESC  (actual time=0.507..0.508 rows=10 loops=1)
    -> Table scan on <temporary>  (actual time=0.486..0.487 rows=10 loops=1)
        -> Aggregate using temporary table  (actual time=0.484..0.484 rows=10 loops=1)
            -> Nested loop inner join  (cost=16.9 rows=37) (actual time=0.137..0.396 rows=37 loops=1)
                -> Index lookup on Contributes using PRIMARY (UserID=5)  (cost=3.96 rows=37) (actual time=0.117..0.125 rows=37 loops=1)
                -> Single-row index lookup on Items using PRIMARY (ItemID=Contributes.ItemID)  (cost=0.253 rows=1) (actual time=0.00693..0.00696 rows=1 loops=37)
 |
+----------------------------------------------------------------------------------------------------
----------------------------------------------------------------------------------------------------
----------------------------------------------------------------------------------------------------
--+
1 row in set (0.01 sec)
```

Indexing on Items.Price (No change):

```
  KEY `index_1` (`Price`),
  CONSTRAINT `Items_ibfk_1` FOREIGN KEY (`ReceiptID`) REFERENCES `Receipts` (`ReceiptID`) ON DELETE CASCADE,
  CONSTRAINT `Items_ibfk_2` FOREIGN KEY (`Category`) REFERENCES `Budget` (`Category`) ON DELETE SET NULL
) ENGINE=InnoDB DEFAULT CHARSET=utf8mb4 COLLATE=utf8mb4_0900_ai_ci |
+-------+--------------------------------------------------------------------------------------------
----------------------------------------------------------------------------------------------------
----------------------------------------------------------------------------------------------------
1 row in set (0.01 sec)

mysql> EXPLAIN ANALYZE
    -> Select Category, Avg(newPrice) as AverageItemCost
    -> FROM (Select Category, (Price * Percentage) as newPrice
    -> FROM Items NATURAL JOIN Contributes
    -> WHERE UserID = 5) as price_subquery
    -> GROUP BY Category
    -> Order by AverageItemCost desc;
+----------------------------------------------------------------------------------------------------
----------------------------------------------------------------------------------------------------
----------------------------------------------------------------------------------------------------
----+
| EXPLAIN


    |
+----------------------------------------------------------------------------------------------------
----------------------------------------------------------------------------------------------------
----------------------------------------------------------------------------------------------------
----+
| -> Sort: AverageItemCost DESC  (actual time=0.331..0.332 rows=10 loops=1)
    -> Table scan on <temporary>  (actual time=0.314..0.316 rows=10 loops=1)
        -> Aggregate using temporary table  (actual time=0.313..0.313 rows=10 loops=1)
            -> Nested loop inner join  (cost=16.9 rows=37) (actual time=0.0591..0.234 rows=37 loops=1)
                -> Index lookup on Contributes using PRIMARY (UserID=5)  (cost=3.96 rows=37) (actual time=0.0385..0.0509 rows=37 loops=1)
                -> Single-row index lookup on Items using PRIMARY (ItemID=Contributes.ItemID)  (cost=0.253 rows=1) (actual time=0.00467..0.0047 rows=1 loops=37)
 |
```

Indexing on Contributes.Percentage (No Change):

```
  KEY `index_2` (`Percentage`),
  CONSTRAINT `Contributes_ibfk_1` FOREIGN KEY (`UserID`) REFERENCES `Accounts` (`UserID`) ON DELETE CASCADE,
  CONSTRAINT `Contributes_ibfk_2` FOREIGN KEY (`ItemID`) REFERENCES `Items` (`ItemID`) ON DELETE CASCADE
) ENGINE=InnoDB DEFAULT CHARSET=utf8mb4 COLLATE=utf8mb4_0900_ai_ci |
+-------------+------------------------------------------------------------------------------------------------------------------
---------------------------------------------------------------------------------------------------------------------------------
---------------------------------------------------------------------------------------------+
1 row in set (0.00 sec)

mysql> EXPLAIN ANALYZE
    -> Select Category, Avg(newPrice) as AverageItemCost
    -> FROM (Select Category, (Price * Percentage) as newPrice
    -> FROM Items NATURAL JOIN Contributes
    -> WHERE UserID = 5) as price_subquery
    -> GROUP BY Category
    -> Order by AverageItemCost desc;
+--------------------------------------------------------------------------------------------------------------------------------
---------------------------------------------------------------------------------------------------------------------------------
---------------------------------------------------------------------------------------------------------------------------------
-----+
| EXPLAIN


     |
+--------------------------------------------------------------------------------------------------------------------------------
---------------------------------------------------------------------------------------------------------------------------------
---------------------------------------------------------------------------------------------------------------------------------
-----+
| -> Sort: AverageItemCost DESC  (actual time=0.465..0.466 rows=10 loops=1)
    -> Table scan on <temporary>  (actual time=0.447..0.449 rows=10 loops=1)
        -> Aggregate using temporary table  (actual time=0.445..0.445 rows=10 loops=1)
            -> Nested loop inner join  (cost=16.9 rows=37) (actual time=0.0447..0.309 rows=37 loops=1)
                -> Index lookup on Contributes using PRIMARY (UserID=5)  (cost=3.96 rows=37) (actual time=0.0279..0.0369 rows=37 loops=1)
                -> Single-row index lookup on Items using PRIMARY (ItemID=Contributes.ItemID)  (cost=0.253 rows=1) (actual time=0.00704..0.00707 rows=1 loops=37)
 |
```

Justification

Indexes queried against Items.Category, Items.Price, and Contributes.Percentage did not show performance improvements since the query optimizer continued to maintain sequential scans with the same cost estimates of 16.9. I think that this lack of benefit is the consequence of three reasons: (1) The WHERE UserID = 5 limitation operated on Contributes.UserID, which was not indexed, which forced full table scans; (2) Natural join actually makes effective use of primary key indexes (ItemID), and no optimization slack remained for secondary indexes on non-key fields; (3) With only 10 categories, group operations were insignificantly overhead-independent with or without an index. While math functions like Price*Percentage in the subquery cannot be indexed directly, the index on Contributes.UserID would have theoretically improved filtering. Because the dataset scale is small and there already are primary key optimizations, the final recommendation for index design is to use the indexes that are actually required, as additional indices provided no noticeable enhancements to this specific query profile and dataset size.

List Users who have total spending under the avgExpense of their income bracket

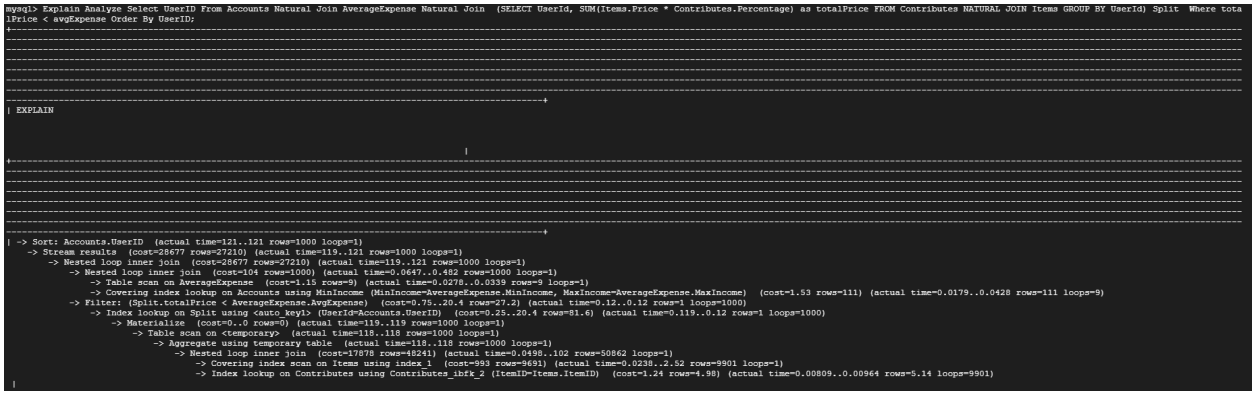
Explain Analyze
Select UserID
From Accounts Natural Join AverageExpense Natural Join
(SELECT UserId, SUM(Items.Price * Contributes.Percentage) as totalPrice FROM Contributes NATURAL JOIN Items GROUP BY UserId) Split
Where Split.totalPrice < avgExpense
Order By UserID
Limit 15;

```
mysql> Select UserID
    -> From Accounts Natural Join AverageExpense Natural Join
    -> (SELECT UserId, SUM(Items.Price * Contributes.Percentage) as totalPrice FROM Contributes NATURAL JOIN Items GROUP BY UserId) Split
    -> Where Split.totalPrice < avgExpense
    -> Order By UserID
    -> Limit 15;
+--------+
| UserID |
+--------+
|      0 |
|      1 |
|      2 |
|      3 |
|      4 |
|      5 |
|      6 |
|      7 |
|      8 |
|      9 |
|     10 |
|     11 |
|     12 |
|     13 |
|     14 |
+--------+
```

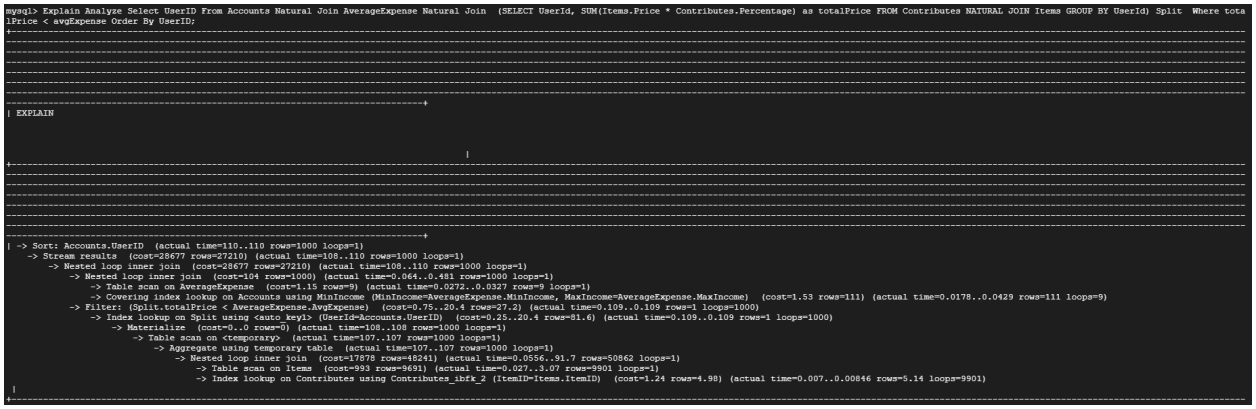
Due to the data we have used, none of the users have spent more than the averageExpense of their tax bracket.

```
mysql> Explain Analyze Select UserID From Accounts Natural Join AverageExpense Natural Join  (SELECT UserId, SUM(Items.Price * Contributes.Percentage) as totalPrice FROM Contributes NATURAL JOIN Items GROUP BY UserId) Split  Where tota
lPrice < avgExpense Order By UserID;
+------------------------------------------------------------------------------------------+
| EXPLAIN                                                                                  |
+------------------------------------------------------------------------------------------+
| -> Sort: Accounts.UserID  (actual time=114..114 rows=1000 loops=1)
    -> Stream results  (cost=29311 rows=27814) (actual time=112..114 rows=1000 loops=1)
        -> Nested loop inner join  (cost=29311 rows=27814) (actual time=112..114 rows=1000 loops=1)
            -> Nested loop inner join  (cost=104 rows=1000) (actual time=0.0679..0.524 rows=1000 loops=1)
                -> Table scan on AverageExpense  (cost=1.15 rows=9) (actual time=0.0276..0.0325 rows=9 loops=1)
                -> Covering index lookup on Accounts using MinIncome (MinIncome=AverageExpense.MinIncome, MaxIncome=AverageExpense.MaxIncome)  (cost=1.53 rows=111) (actual time=0.0201..0.0476 rows=111 loops=9)
            -> Filter: (Split.totalPrice < AverageExpense.AvgExpense)  (cost=0.75..20.9 rows=27.8) (actual time=0.113..0.113 rows=1 loops=1000)
                -> Index lookup on Split using <auto_key1> (UserId=Accounts.UserID)  (cost=0.25..20.9 rows=83.4) (actual time=0.113..0.113 rows=1 loops=1000)
                    -> Materialize  (cost=0..0 rows=0) (actual time=112..112 rows=1000 loops=1)
                        -> Table scan on <temporary>  (actual time=111..111 rows=1000 loops=1)
                            -> Aggregate using temporary table  (actual time=111..111 rows=1000 loops=1)
                                -> Nested loop inner join  (cost=18274 rows=49311) (actual time=0.0609..94.8 rows=50862 loops=1)
                                    -> Table scan on Items  (cost=1015 rows=9906) (actual time=0.0295..3.04 rows=9901 loops=1)
                                    -> Index lookup on Contributes using Contributes_ibfk_2 (ItemID=Items.ItemID)  (cost=1.24 rows=4.98) (actual time=0.00732..0.00883 rows=5.14 loops=9901)
 |
```

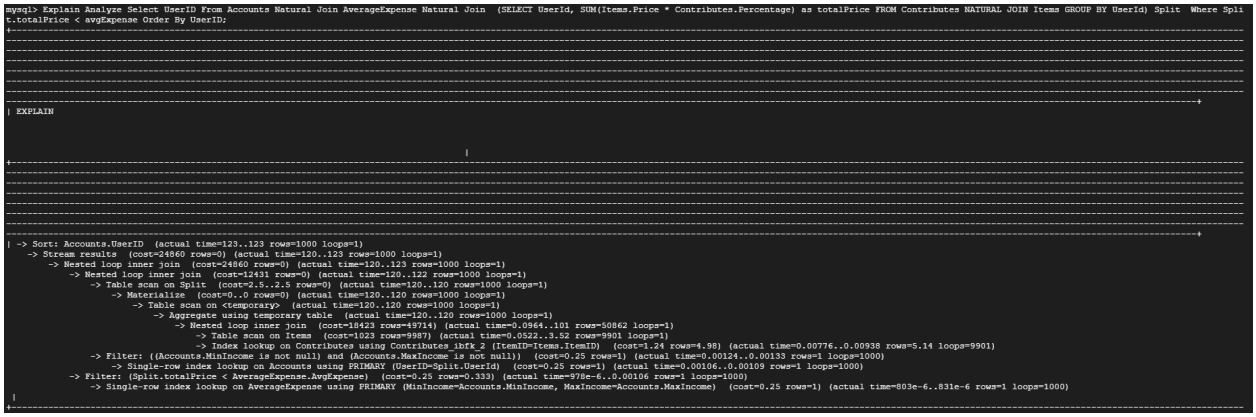
Index Items.Price



Index Items.Price and Contributes.Percentage



Indexing avgExpense



For the query that lists users who have total spending under the avgExpense of their income bracket, we decided to Index avgExpense. After indexing avgExpense, the cost went down a good amount. This is likely due to the fact that Split.totalPrice is being compared to avgExpense in the Where clause, making indexing avgExpense speed up the query. Indexing Items.Price did decrease the cost by a little but weirdly enough, it seems like indexing Contributes.Percentage didn't change the cost from just indexing Items.Price. Also, removing the indices does not revert the cost to the previous cost.

Receipts in a given year that contain only items bought by a group of users in which all users contribute:

Note: Should be given a temporary table of userIds. (38,50,296) used as userIDs and 2023 used as the year.

```
mysql> Select ReceiptID
    -> From
    ->     (Select ReceiptID, Count(UserID) as totalUsers
    ->     From Items Natural Join Contributes
    ->     Where UserId in
    ->         (Select UserID
    ->         From Accounts
    ->         Where UserID in (38,50,296))
    ->
    ->         and
    ->
    ->         ReceiptID not in
    ->         (Select ReceiptID
    ->         From Items Natural Join Contributes
    ->         Where not UserId in
    ->             (Select UserID
    ->             From Accounts
    ->             Where UserID in (38,50,296)))
    ->     Group by ReceiptID) as A
    ->
    ->     Natural Join
    ->
    ->     Receipts
    -> Where totalUsers =
    ->     (Select Count(UserID)
    ->     From Accounts
    ->     Where UserID in (38,50,296))
    ->
    ->     and
    ->
    ->     PurchaseDate like "2023%"
    -> ;
+-----------+
| ReceiptID |
+-----------+
|       227 |
+-----------+
1 row in set (0.01 sec)
```

Output is less than 15 because there only exists one receipt that only contains the users in the example where each user buys at least one item. Our data just makes it rare for this situation to happen but for actual use cases, it would be more common.

```
| -> Nested loop inner join  (cost=30.1 rows=1.66) (actual time=3.13..3.13 rows=1 loops=1)
    -> Filter: ((A.totalUsers = (select #6)) and (A.ReceiptID is not null))  (cost=1.29..19.3 rows=14.9) (actual time=3.1..3.11 rows=1 loops=1)
        -> Table scan on A  (cost=2.5..2.5 rows=0) (actual time=3.08..3.08 rows=1 loops=1)
            -> Materialize  (cost=0..0 rows=0) (actual time=3.08..3.08 rows=1 loops=1)
                -> Table scan on <temporary>  (actual time=3.06..3.06 rows=1 loops=1)
                    -> Aggregate using temporary table  (actual time=3.06..3.06 rows=1 loops=1)
                        -> Nested loop inner join  (cost=70.3 rows=149) (actual time=0.282..3.05 rows=3 loops=1)
                            -> Nested loop inner join  (cost=18 rows=149) (actual time=0.0404..0.114 rows=170 loops=1)
                                -> Filter: (Accounts.UserID in (38,50,296))  (cost=2.26 rows=3) (actual time=0.0229..0.0284 rows=3 loops=1)
                                    -> Covering index range scan on Accounts using PRIMARY over (UserID = 38) OR (UserID = 50) OR (UserID = 296)  (cost=2.26 rows=3) (actual time=0.0216..0.0264 rows=3 loops=1)
                                -> Covering index lookup on Contributes using PRIMARY (UserID=Accounts.UserID)  (cost=1.93 rows=49.8) (actual time=0.0138..0.0241 rows=56.7 loops=3)
                            -> Filter: <in_optimizer>(Items.ReceiptID,<exists>(select #4) is false)  (cost=0.251 rows=1) (actual time=0.0172..0.0172 rows=0.0176 loops=170)
                                -> Single-row index lookup on Items using PRIMARY (ItemID=Contributes.ItemID)  (cost=0.251 rows=1) (actual time=0.0019..0.00193 rows=1 loops=170)
                                -> Select #4 (subquery in condition; dependent)
                                    -> Limit: 1 row(s)  (cost=56.5 rows=1) (actual time=0.0138..0.0138 rows=0.982 loops=170)
                                        -> Filter: <if>(outer_field_is_not_null, <is_not_null_test>(Items.ReceiptID), true)  (cost=56.5 rows=296) (actual time=0.0136..0.0136 rows=0.982 loops=170)
                                            -> Nested loop antijoin  (cost=56.5 rows=296) (actual time=0.0135..0.0135 rows=0.982 loops=170)
                                                -> Nested loop inner join  (cost=17.1 rows=98.6) (actual time=0.008..0.00811 rows=1.11 loops=170)
                                                    -> Filter: <if>(outer_field_is_not_null, ((<cache>(Items.ReceiptID) = Items.ReceiptID) or (Items.ReceiptID is null)), true)  (cost=2.24 rows=19.8) (actual time=0.00444..0.00451 rows=1 loops=170)
                                                        -> Alternative plans for IN subquery: Index lookup unless ReceiptID IS NULL  (cost=2.24 rows=19.8) (actual time=0.00422..0.00429 rows=1 loops=170)
                                                            -> Covering index lookup on Items using Items_ibfk_1 (ReceiptID=<cache>(Items.ReceiptID) or NULL)  (cost=2.24 rows=19.8) (actual time=0.00406..0.00413 rows=1 loops=170)
                                                            -> Table scan on Items  (cost=24.2 rows=9906) (never executed)
                                                    -> Covering index lookup on Contributes using Contributes_ibfk_2 (ItemID=Items.ItemID)  (cost=0.277 rows=4.98) (actual time=0.00335..0.00338 rows=1.11 loops=170)
                                                -> Single-row index lookup on <subquery5> using <auto_distinct_key> (UserID=Contributes.UserID)  (cost=5.88..5.88 rows=1) (actual time=0.0046..0.0046 rows=0.116 loops=189)
                                                    -> Materialize with deduplication  (cost=2.56..2.56 rows=3) (actual time=0.00475..0.00475 rows=3 loops=170)
                                                        -> Filter: (Accounts.UserID is not null)  (cost=2.26 rows=3) (actual time=0.00166..0.00399 rows=3 loops=170)
                                                            -> Filter: (Accounts.UserID in (38,50,296))  (cost=2.26 rows=3) (actual time=0.00152..0.00365 rows=3 loops=170)
                                                                -> Covering index range scan on Accounts using PRIMARY over (UserID = 38) OR (UserID = 50) OR (UserID = 296)  (cost=2.26 rows=3) (actual time=0.00131..0.00314 rows=3 loops=170)
    -> Select #6 (subquery in condition; run only once)
        -> Aggregate: count(Accounts.UserID)  (cost=2.56 rows=1) (actual time=0.0102..0.0103 rows=1 loops=1)
            -> Filter: (Accounts.UserID in (38,50,296))  (cost=2.26 rows=3) (actual time=0.00505..0.00713 rows=3 loops=1)
                -> Covering index range scan on Accounts using PRIMARY over (UserID = 38) OR (UserID = 50) OR (UserID = 296)  (cost=2.26 rows=3) (actual time=0.00424..0.00608 rows=3 loops=1)
  -> Filter: (Receipts.PurchaseDate like '2023%')  (cost=0.626 rows=0.111) (actual time=0.0231..0.0232 rows=1 loops=1)
      -> Single-row index lookup on Receipts using PRIMARY (ReceiptID=A.ReceiptID)  (cost=0.626 rows=1) (actual time=0.0165..0.0166 rows=1 loops=1)
|
```

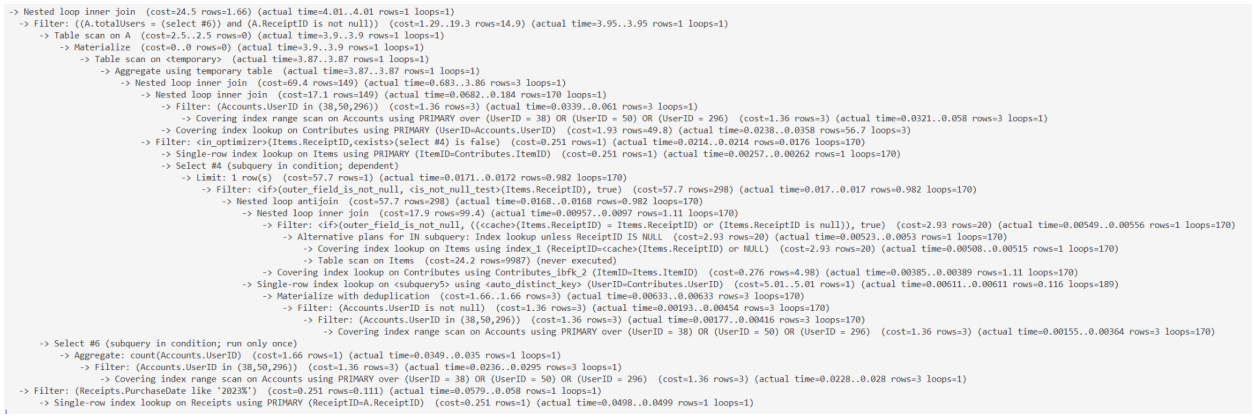
Items.ReceiptID (Cost up)

```
-> Nested loop inner join  (cost=30.1 rows=1.66) (actual time=6.04..6.04 rows=1 loops=1)
  -> Filter: ((A.totalUsers = (select #6)) and (A.ReceiptID is not null))  (cost=1.29..19.3 rows=14.9) (actual time=5.91..5.91 rows=1 loops=1)
      -> Table scan on A  (cost=2.5..2.5 rows=0) (actual time=5.84..5.84 rows=1 loops=1)
          -> Materialize  (cost=0..0 rows=0) (actual time=5.84..5.84 rows=1 loops=1)
              -> Table scan on <temporary>  (actual time=5.82..5.82 rows=1 loops=1)
                  -> Aggregate using temporary table  (actual time=5.82..5.82 rows=1 loops=1)
                      -> Nested loop inner join  (cost=70.3 rows=149) (actual time=2.38..5.75 rows=3 loops=1)
                          -> Nested loop inner join  (cost=18 rows=149) (actual time=1.01..1.08 rows=170 loops=1)
                              -> Filter: (Accounts.UserID in (38,50,296))  (cost=2.26 rows=3) (actual time=0.974..0.981 rows=3 loops=1)
                                  -> Covering index range scan on Accounts using PRIMARY over (UserID = 38) OR (UserID = 50) OR (UserID = 296)  (cost=2.26 rows=3) (actual time=0.0453..0.0514 rows=3 loops=1)
                              -> Covering index lookup on Contributes using PRIMARY (UserID=Accounts.UserID)  (cost=1.93 rows=49.8) (actual time=0.0192..0.0295 rows=56.7 loops=3)
                          -> Filter: <in_optimizer>(Items.ReceiptID,<exists>(select #4) is false)  (cost=0.251 rows=1) (actual time=0.0273..0.0273 rows=0.0176 loops=170)
                              -> Single-row index lookup on Items using PRIMARY (ItemID=Contributes.ItemID)  (cost=0.251 rows=1) (actual time=0.00453..0.00456 rows=1 loops=170)
                              -> Select #4 (subquery in condition; dependent)
                                  -> Limit: 1 row(s)  (cost=56 rows=1) (actual time=0.0168..0.0169 rows=0.982 loops=170)
                                      -> Filter: <if>(outer_field_is_not_null, <is_not_null_test>(Items.ReceiptID), true)  (cost=56 rows=289) (actual time=0.0167..0.0167 rows=0.982 loops=170)
                                          -> Nested loop antijoin  (cost=56 rows=289) (actual time=0.0165..0.0165 rows=0.982 loops=170)
                                              -> Nested loop inner join  (cost=17.4 rows=96.5) (actual time=0.00959..0.00968 rows=1.11 loops=170)
                                                  -> Filter: <if>(outer_field_is_not_null, ((<cache>(Items.ReceiptID) = Items.ReceiptID) or (Items.ReceiptID is null)), true)  (cost=2.87 rows=19.4) (actual time=0.00573..0.00578 rows=1 loops=170)
                                                      -> Alternative plans for IN subquery: Index lookup unless ReceiptID IS NULL  (cost=2.87 rows=19.4) (actual time=0.00538..0.00542 rows=1 loops=170)
                                                          -> Covering index lookup on Items using index_3 (ReceiptID=<cache>(Items.ReceiptID) or NULL)  (cost=2.87 rows=19.4) (actual time=0.00524..0.00528 rows=1 loops=170)
                                                          -> Table scan on Items  (cost=24.2 rows=9691) (never executed)
                                                  -> Covering index lookup on Contributes using Contributes_ibfk_2 (ItemID=Items.ItemID)  (cost=0.277 rows=4.98) (actual time=0.00367..0.0037 rows=1.11 loops=170)
                                              -> Single-row index lookup on <subquery5> using <auto_distinct_key> (UserID=Contributes.UserID)  (cost=5.81..5.81 rows=1) (actual time=0.00593..0.00593 rows=0.116 loops=189)
                                                  -> Materialize with deduplication  (cost=2.56..2.56 rows=3) (actual time=0.00622..0.00622 rows=3 loops=170)
                                                      -> Filter: (Accounts.UserID is not null)  (cost=2.26 rows=3) (actual time=0.002..0.00439 rows=3 loops=170)
                                                          -> Filter: (Accounts.UserID in (38,50,296))  (cost=2.26 rows=3) (actual time=0.00186..0.00405 rows=3 loops=170)
                                                              -> Covering index range scan on Accounts using PRIMARY over (UserID = 38) OR (UserID = 50) OR (UserID = 296)  (cost=2.26 rows=3) (actual time=0.00151..0.00344 rows=3 loops=170)
    -> Select #6 (subquery in condition; run only once)
        -> Aggregate: count(Accounts.UserID)  (cost=2.56 rows=1) (actual time=0.0473..0.0474 rows=1 loops=1)
            -> Filter: (Accounts.UserID in (38,50,296))  (cost=2.26 rows=3) (actual time=0.0416..0.0438 rows=3 loops=1)
                -> Covering index range scan on Accounts using PRIMARY over (UserID = 38) OR (UserID = 50) OR (UserID = 296)  (cost=2.26 rows=3) (actual time=0.0409..0.0429 rows=3 loops=1)
  -> Filter: (Receipts.PurchaseDate like '2023%')  (cost=0.626 rows=0.111) (actual time=0.13..0.13 rows=1 loops=1)
      -> Single-row index lookup on Receipts using PRIMARY (ReceiptID=A.ReceiptID)  (cost=0.626 rows=1) (actual time=0.11..0.11 rows=1 loops=1)
```

Receipts.PurchaseDate (Cost Down)

```
-> Nested loop inner join  (cost=24.5 rows=1.66) (actual time=5.68..5.68 rows=1 loops=1)
  -> Filter: ((A.totalUsers = (select #6)) and (A.ReceiptID is not null))  (cost=1.29..19.3 rows=14.9) (actual time=5.66..5.66 rows=1 loops=1)
      -> Table scan on A  (cost=2.5..2.5 rows=0) (actual time=5.63..5.63 rows=1 loops=1)
          -> Materialize  (cost=0..0 rows=0) (actual time=5.63..5.63 rows=1 loops=1)
              -> Table scan on <temporary>  (actual time=5.61..5.61 rows=1 loops=1)
                  -> Aggregate using temporary table  (actual time=5.61..5.61 rows=1 loops=1)
                      -> Nested loop inner join  (cost=69.4 rows=149) (actual time=2.57..5.59 rows=3 loops=1)
                          -> Nested loop inner join  (cost=17.1 rows=149) (actual time=0.74..0.822 rows=170 loops=1)
                              -> Filter: (Accounts.UserID in (38,50,296))  (cost=1.36 rows=3) (actual time=0.715..0.721 rows=3 loops=1)
                                  -> Covering index range scan on Accounts using PRIMARY over (UserID = 38) OR (UserID = 50) OR (UserID = 296)  (cost=1.36 rows=3) (actual time=0.0301..0.0348 rows=3 loops=1)
                              -> Covering index lookup on Contributes using PRIMARY (UserID=Accounts.UserID)  (cost=1.93 rows=49.8) (actual time=0.0172..0.0285 rows=56.7 loops=3)
                          -> Filter: <in_optimizer>(Items.ReceiptID,<exists>(select #4) is false)  (cost=0.251 rows=1) (actual time=0.0279..0.0279 rows=0.0176 loops=170)
                              -> Single-row index lookup on Items using PRIMARY (ItemID=Contributes.ItemID)  (cost=0.251 rows=1) (actual time=0.00211..0.00214 rows=1 loops=170)
                              -> Select #4 (subquery in condition; dependent)
                                  -> Limit: 1 row(s)  (cost=57 rows=1) (actual time=0.0153..0.0153 rows=0.982 loops=170)
                                      -> Filter: <if>(outer_field_is_not_null, <is_not_null_test>(Items.ReceiptID), true)  (cost=57 rows=298) (actual time=0.0151..0.0151 rows=0.982 loops=170)
                                          -> Nested loop antijoin  (cost=57 rows=298) (actual time=0.0149..0.0149 rows=0.982 loops=170)
                                              -> Nested loop inner join  (cost=17.2 rows=99.4) (actual time=0.00885..0.00899 rows=1.11 loops=170)
                                                  -> Filter: <if>(outer_field_is_not_null, ((<cache>(Items.ReceiptID) = Items.ReceiptID) or (Items.ReceiptID is null)), true)  (cost=2.25 rows=20) (actual time=0.00496..0.00506 rows=1 loops=170)
                                                      -> Alternative plans for IN subquery: Index lookup unless ReceiptID IS NULL  (cost=2.25 rows=20) (actual time=0.0047..0.0048 rows=1 loops=170)
                                                          -> Covering index lookup on Items using ReceiptID (ReceiptID=<cache>(Items.ReceiptID) or NULL)  (cost=2.25 rows=20) (actual time=0.00439..0.00449 rows=1 loops=170)
                                                          -> Table scan on Items  (cost=24.2 rows=9987) (never executed)
                                                  -> Covering index lookup on Contributes using Contributes_ibfk_2 (ItemID=Items.ItemID)  (cost=0.276 rows=4.98) (actual time=0.00369..0.00372 rows=1.11 loops=170)
                                              -> Single-row index lookup on <subquery5> using <auto_distinct_key> (UserID=Contributes.UserID)  (cost=5.01..5.01 rows=1) (actual time=0.00509..0.00509 rows=0.116 loops=189)
                                                  -> Materialize with deduplication  (cost=1.66..1.66 rows=3) (actual time=0.00517..0.00517 rows=3 loops=170)
                                                      -> Filter: (Accounts.UserID is not null)  (cost=1.36 rows=3) (actual time=0.00182..0.00425 rows=3 loops=170)
                                                          -> Filter: (Accounts.UserID in (38,50,296))  (cost=1.36 rows=3) (actual time=0.00167..0.00383 rows=3 loops=170)
                                                              -> Covering index range scan on Accounts using PRIMARY over (UserID = 38) OR (UserID = 50) OR (UserID = 296)  (cost=1.36 rows=3) (actual time=0.00146..0.00336 rows=3 loops=170)
    -> Select #6 (subquery in condition; run only once)
        -> Aggregate: count(Accounts.UserID)  (cost=1.66 rows=1) (actual time=0.011..0.0111 rows=1 loops=1)
            -> Filter: (Accounts.UserID in (38,50,296))  (cost=1.36 rows=3) (actual time=0.00568..0.00775 rows=3 loops=1)
                -> Covering index range scan on Accounts using PRIMARY over (UserID = 38) OR (UserID = 50) OR (UserID = 296)  (cost=1.36 rows=3) (actual time=0.00494..0.00673 rows=3 loops=1)
  -> Filter: (Receipts.PurchaseDate like '2023%')  (cost=0.251 rows=0.111) (actual time=0.0192..0.0194 rows=1 loops=1)
      -> Single-row index lookup on Receipts using PRIMARY (ReceiptID=A.ReceiptID)  (cost=0.251 rows=1) (actual time=0.0124..0.0125 rows=1 loops=1)
```

Items.ReceiptID and Receipts.PurchaseDate (Cost Down but less)



```
-> Nested loop inner join  (cost=24.5 rows=1.66) (actual time=4.01..4.01 rows=1 loops=1)
  -> Filter: ((A.totalUsers = (select #6)) and (A.ReceiptID is not null))  (cost=1.29..19.3 rows=14.9) (actual time=3.95..3.95 rows=1 loops=1)
      -> Table scan on A  (cost=2.5..2.5 rows=0) (actual time=3.9..3.9 rows=1 loops=1)
          -> Materialize  (cost=0..0 rows=0) (actual time=3.9..3.9 rows=1 loops=1)
              -> Table scan on <temporary>  (actual time=3.87..3.87 rows=1 loops=1)
                  -> Aggregate using temporary table  (actual time=3.87..3.87 rows=1 loops=1)
                      -> Nested loop inner join  (cost=69.4 rows=149) (actual time=0.683..3.86 rows=3 loops=1)
                          -> Nested loop inner join  (cost=17.1 rows=149) (actual time=0.0682..0.184 rows=170 loops=1)
                              -> Filter: (Accounts.UserID in (38,50,296))  (cost=1.36 rows=3) (actual time=0.0339..0.061 rows=3 loops=1)
                                  -> Covering index range scan on Accounts using PRIMARY over (UserID = 38) OR (UserID = 50) OR (UserID = 296)  (cost=1.36 rows=3) (actual time=0.0321..0.058 rows=3 loops=1)
                              -> Covering index lookup on Contributes using PRIMARY (UserID=Accounts.UserID)  (cost=1.93 rows=49.8) (actual time=0.0238..0.0358 rows=56.7 loops=3)
                          -> Filter: <in_optimizer>(Items.ReceiptID,<exists>(select #4) is false)  (cost=0.251 rows=1) (actual time=0.0214..0.0214 rows=0.0176 loops=170)
                              -> Single-row index lookup on Items using PRIMARY (ItemID=Contributes.ItemID)  (cost=0.251 rows=1) (actual time=0.00257..0.00262 rows=1 loops=170)
                              -> Select #4 (subquery in condition; dependent)
                                  -> Limit: 1 row(s)  (cost=57.7 rows=1) (actual time=0.0171..0.0172 rows=0.982 loops=170)
                                      -> Filter: <if>(outer_field_is_not_null, <is_not_null_test>(Items.ReceiptID), true)  (cost=57.7 rows=298) (actual time=0.017..0.017 rows=0.982 loops=170)
                                          -> Nested loop antijoin  (cost=57.7 rows=298) (actual time=0.0168..0.0168 rows=0.982 loops=170)
                                              -> Nested loop inner join  (cost=17.9 rows=99.4) (actual time=0.00957..0.0097 rows=1.11 loops=170)
                                                  -> Filter: <if>(outer_field_is_not_null, ((<cache>(Items.ReceiptID) = Items.ReceiptID) or (Items.ReceiptID is null)), true)  (cost=2.93 rows=20) (actual time=0.00549..0.00556 rows=1 loops=170)
                                                      -> Alternative plans for IN subquery: Index lookup unless ReceiptID IS NULL  (cost=2.93 rows=20) (actual time=0.00523..0.0053 rows=1 loops=170)
                                                          -> Covering index lookup on Items using index_1 (ReceiptID=<cache>(Items.ReceiptID) or NULL)  (cost=2.93 rows=20) (actual time=0.00508..0.00515 rows=1 loops=170)
                                                          -> Table scan on Items  (cost=24.2 rows=9987) (never executed)
                                                  -> Covering index lookup on Contributes using Contributes_ibfk_2 (ItemID=Items.ItemID)  (cost=0.276 rows=4.98) (actual time=0.00385..0.00389 rows=1.11 loops=170)
                                              -> Single-row index lookup on <subquery5> using <auto_distinct_key> (UserID=Contributes.UserID)  (cost=5.01..5.01 rows=1) (actual time=0.00611..0.00611 rows=0.116 loops=189)
                                                  -> Materialize with deduplication  (cost=1.66..1.66 rows=3) (actual time=0.00633..0.00633 rows=3 loops=170)
                                                      -> Filter: (Accounts.UserID is not null)  (cost=1.36 rows=3) (actual time=0.00193..0.00454 rows=3 loops=170)
                                                          -> Filter: (Accounts.UserID in (38,50,296))  (cost=1.36 rows=3) (actual time=0.00177..0.00416 rows=3 loops=170)
                                                              -> Covering index range scan on Accounts using PRIMARY over (UserID = 38) OR (UserID = 50) OR (UserID = 296)  (cost=1.36 rows=3) (actual time=0.00155..0.00364 rows=3 loops=170)
      -> Select #6 (subquery in condition; run only once)
          -> Aggregate: count(Accounts.UserID)  (cost=1.66 rows=1) (actual time=0.0349..0.035 rows=1 loops=1)
              -> Filter: (Accounts.UserID in (38,50,296))  (cost=1.36 rows=3) (actual time=0.0236..0.0295 rows=3 loops=1)
                  -> Covering index range scan on Accounts using PRIMARY over (UserID = 38) OR (UserID = 50) OR (UserID = 296)  (cost=1.36 rows=3) (actual time=0.0228..0.028 rows=3 loops=1)
  -> Filter: (Receipts.PurchaseDate like '2023%')  (cost=0.251 rows=0.111) (actual time=0.0579..0.058 rows=1 loops=1)
      -> Single-row index lookup on Receipts using PRIMARY (ReceiptID=A.ReceiptID)  (cost=0.251 rows=1) (actual time=0.0498..0.0499 rows=1 loops=1)
```

Justification:

In my query, the main attributes that are used are ReceiptID, PurchaseDate, and UserID. We are testing indexing on the first two as the UserID is already a primary. Receipts.PurchaseDate decreased the cost as it makes it easier to find the year. Items.ReceiptID isn't directly used for searching so in theory, it shouldn't heavily affect the cost. The cost increased slightly possibly due to some other underlying reason such as memory or storage, the real reason is unclear at the moment. The indexing on both acts as expected, decreasing the cost overall due to ReceiptID but increasing slightly from PurchaseDate.