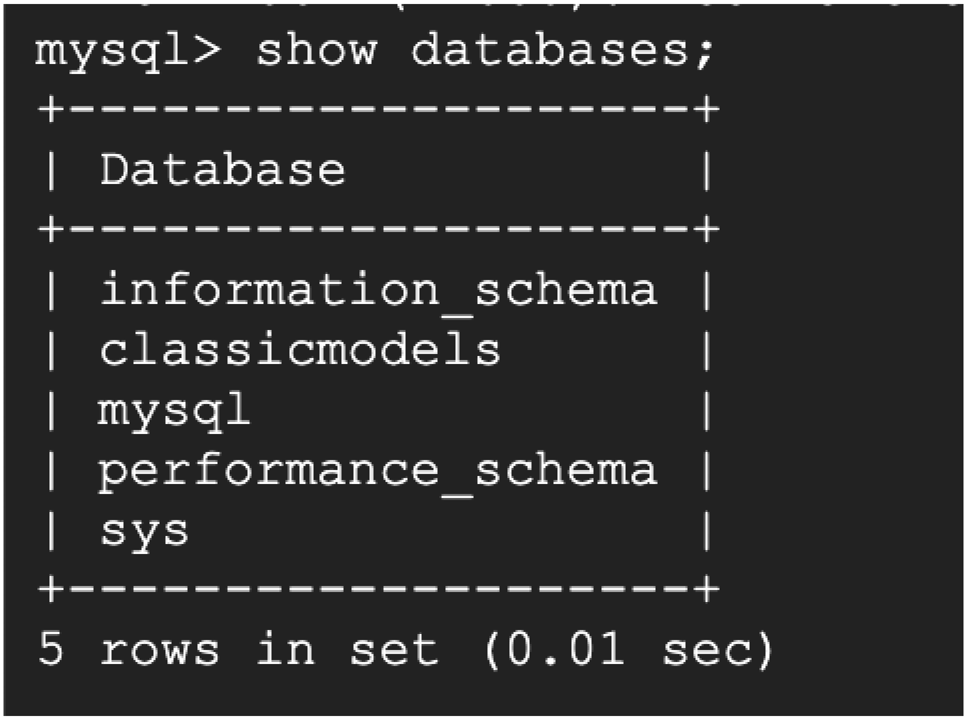
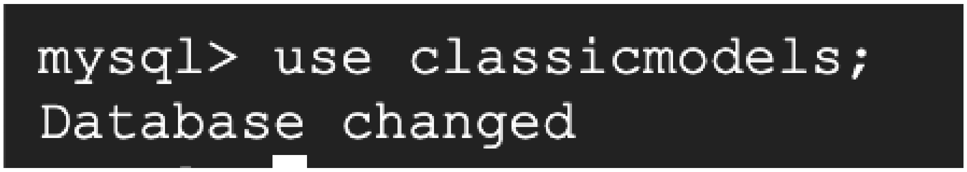
Stage 3: Database Implementation and Indexing (22%)

Implement at least four main tables

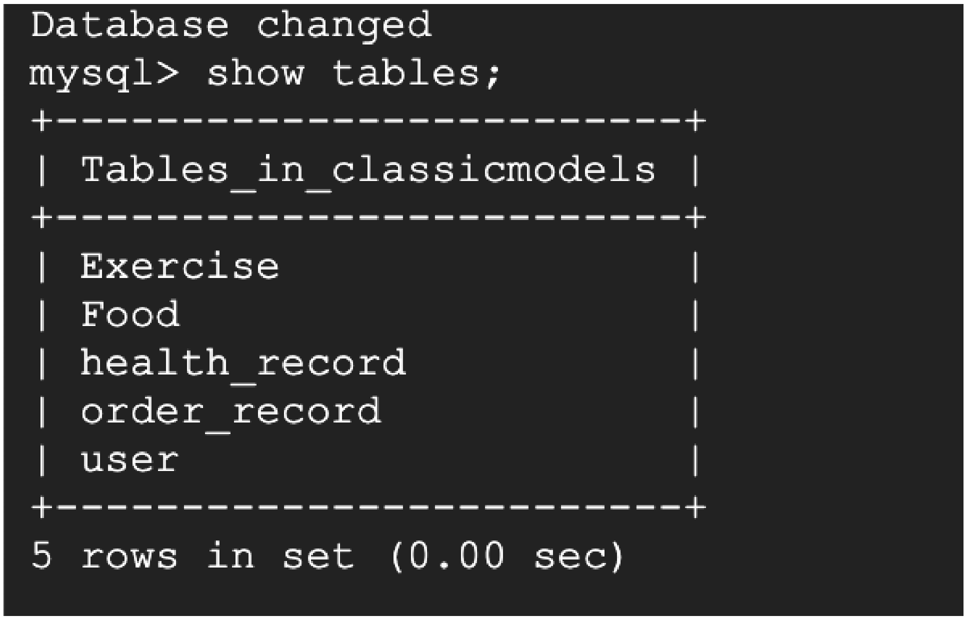
- User
- Health_record
- Food
- Exercise
- Order_records



```
mysql> show databases;
+--------------------+
| Database           |
+--------------------+
| information_schema |
| classicmodels      |
| mysql              |
| performance_schema |
| sys                |
+--------------------+
5 rows in set (0.01 sec)
```



```
mysql> use classicmodels;
Database changed
```



```
Database changed
mysql> show tables;
+------------------------+
| Tables_in_classicmodels |
+------------------------+
| Exercise               |
| Food                   |
| health_record          |
| order_record           |
| user                   |
+------------------------+
5 rows in set (0.00 sec)
```

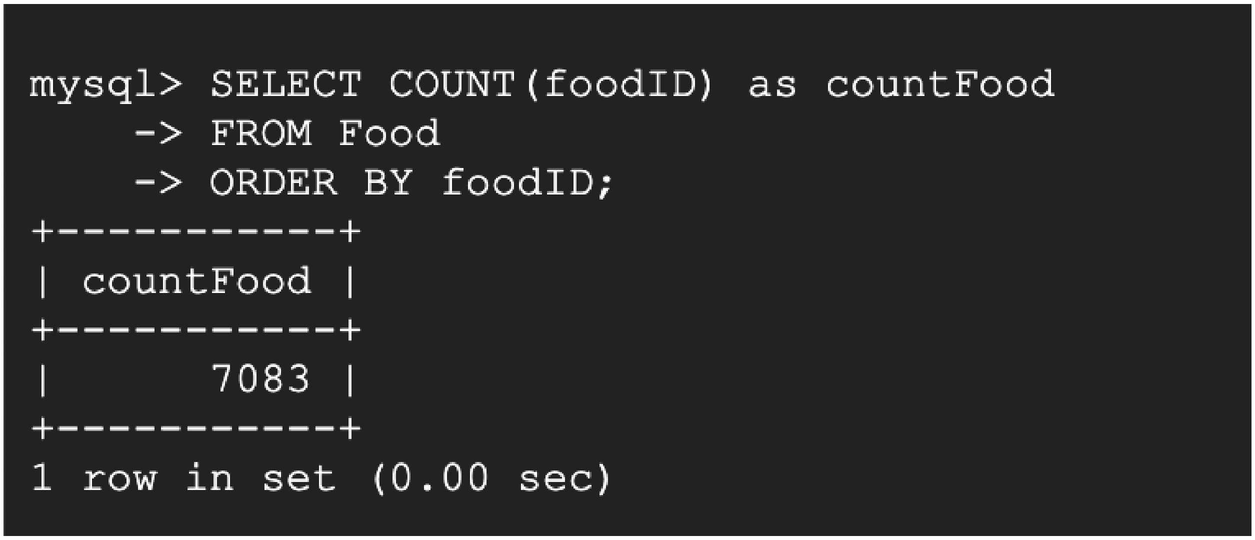
In the Database Design markdown or pdf, provide the Data Definition Language (DDL) commands you all used to create each of these tables in the database.

Markdown file:

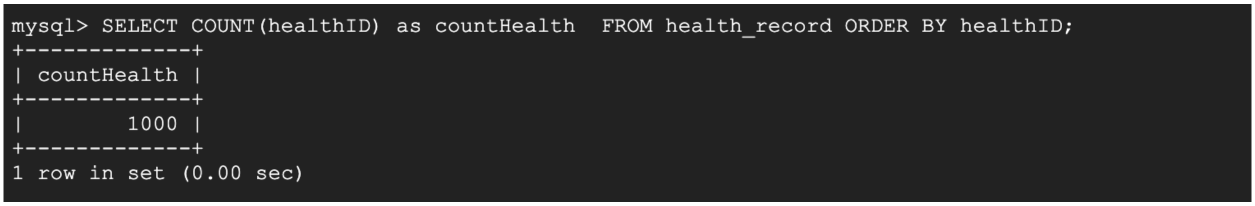
https://s3-us-west-2.amazonaws.com/secure.notion-static.com/2bbe5576-e7a5-4b8c-9825-34b0754a2e6d/ddl.md

Insert data to these tables. You should insert at least 1000 rows each in three of the tables.

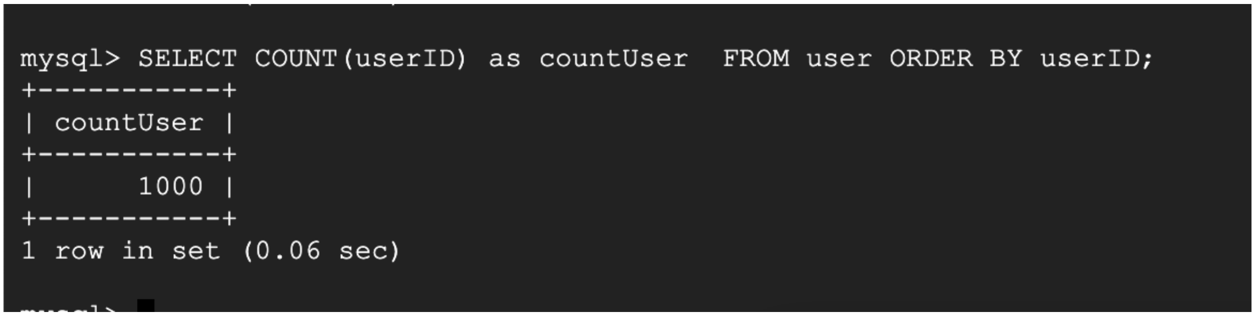
- User
- Health_record
- Food



```
mysql> SELECT COUNT(foodID) as countFood
    -> FROM Food
    -> ORDER BY foodID;
+-----------+
| countFood |
+-----------+
|      7083 |
+-----------+
1 row in set (0.00 sec)
```



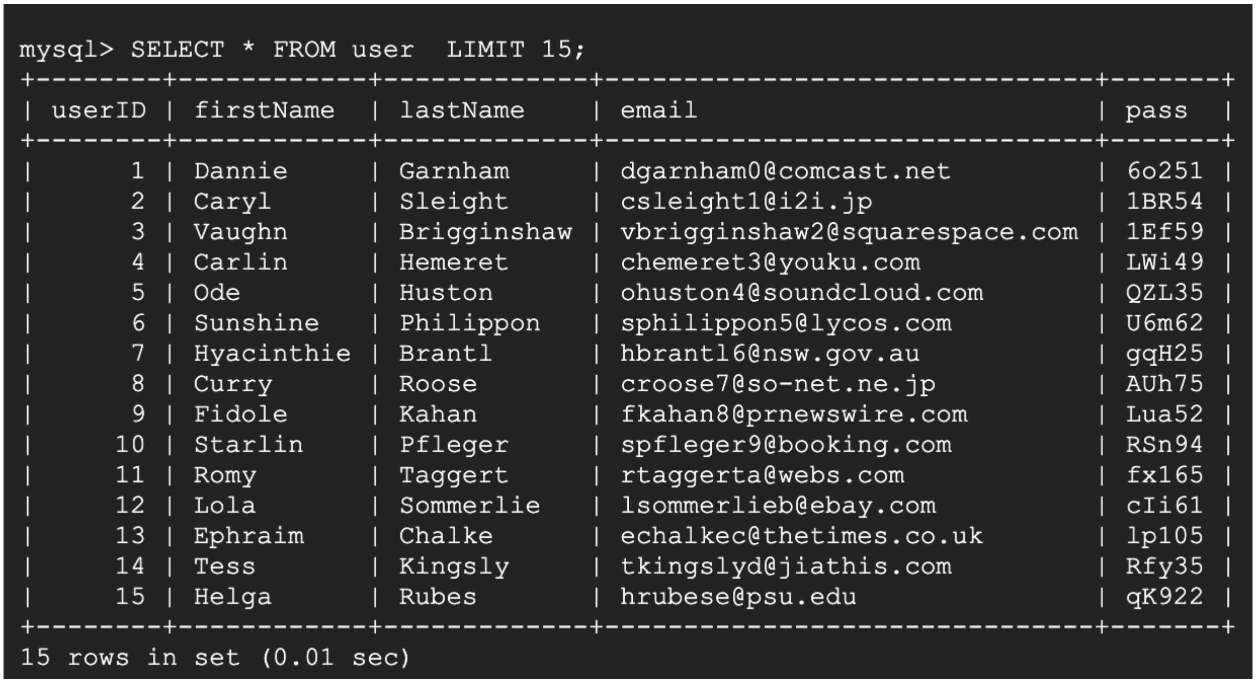
```
mysql> SELECT COUNT(healthID) as countHealth  FROM health_record ORDER BY healthID;
+-------------+
| countHealth |
+-------------+
|        1000 |
+-------------+
1 row in set (0.00 sec)
```



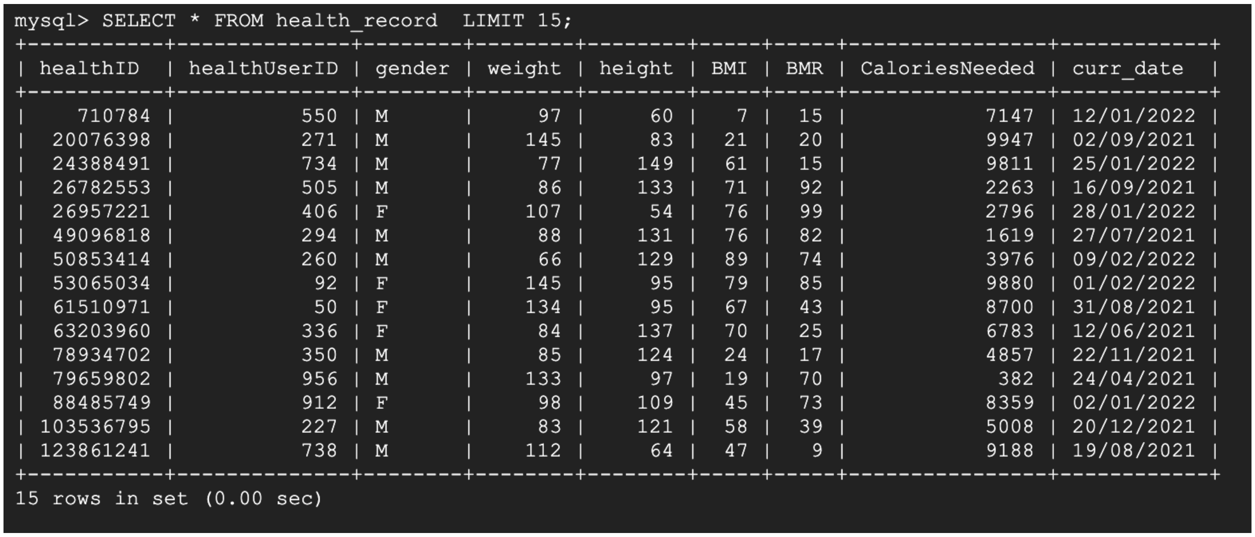
```
mysql> SELECT COUNT(userID) as countUser  FROM user ORDER BY userID;
+-----------+
| countUser |
+-----------+
|      1000 |
+-----------+
1 row in set (0.06 sec)

mysql> 
```

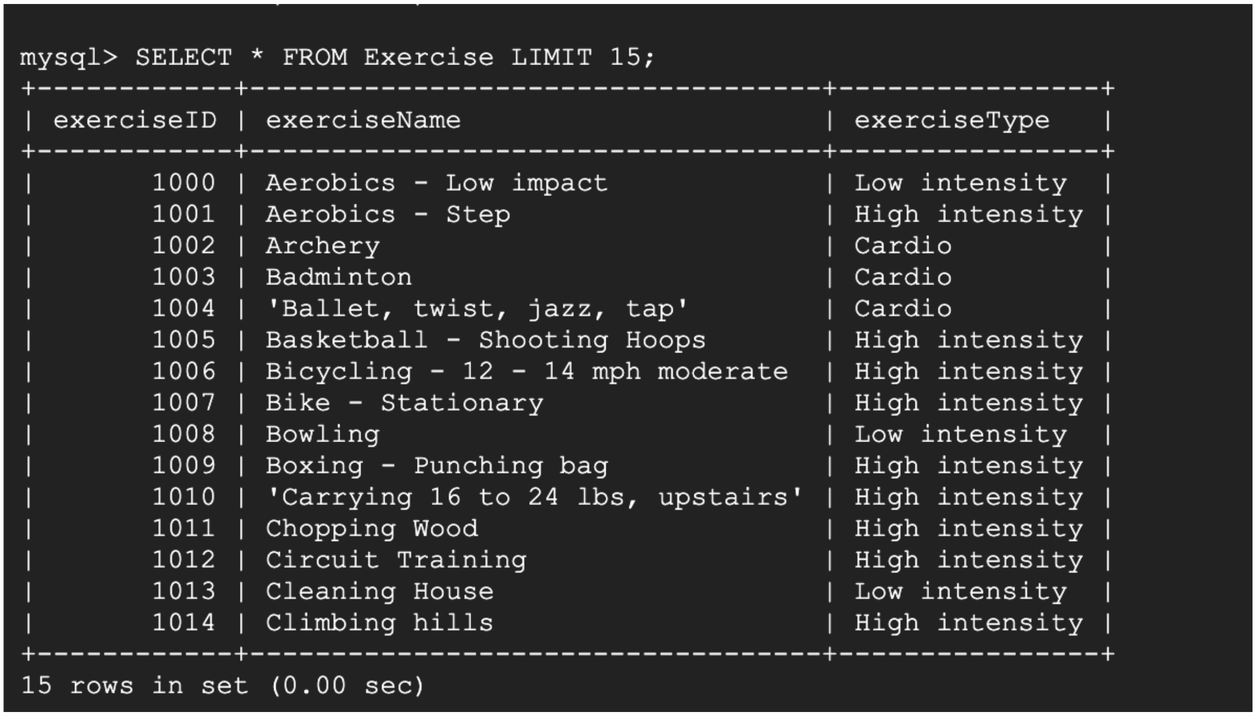
Datas inside our tables



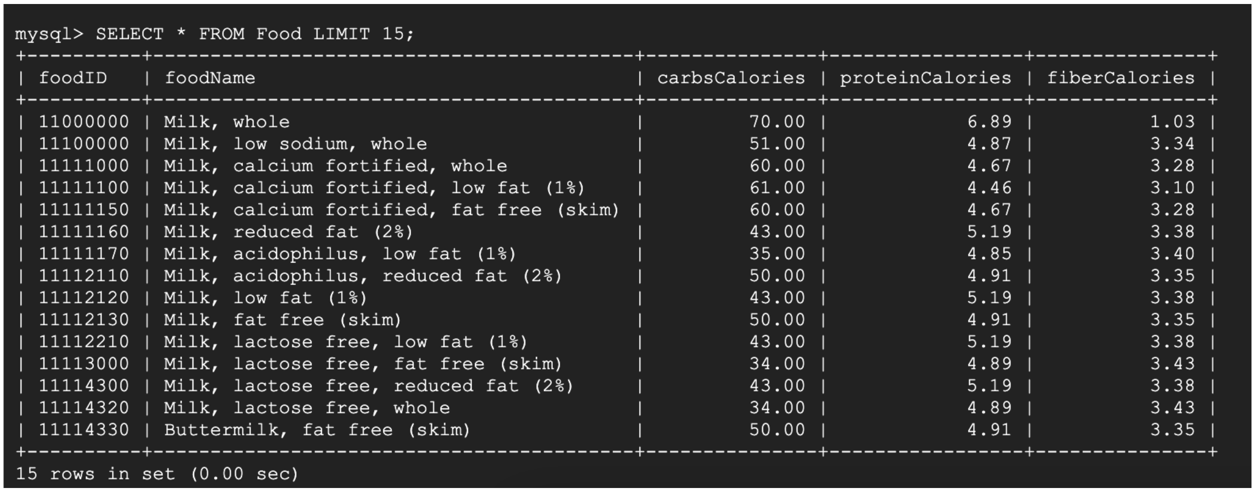
```
mysql> SELECT * FROM user  LIMIT 15;
+--------+------------+-------------+--------------------------------+-------+
| userID | firstName  | lastName    | email                          | pass  |
+--------+------------+-------------+--------------------------------+-------+
|      1 | Dannie     | Garnham     | dgarnham0@comcast.net          | 6o251 |
|      2 | Caryl      | Sleight     | csleight1@i2i.jp               | 1BR54 |
|      3 | Vaughn     | Brigginshaw | vbrigginshaw2@squarespace.com  | 1Ef59 |
|      4 | Carlin     | Hemeret     | chemeret3@youku.com            | LWi49 |
|      5 | Ode        | Huston      | ohuston4@soundcloud.com        | QZL35 |
|      6 | Sunshine   | Philippon   | sphilippon5@lycos.com          | U6m62 |
|      7 | Hyacinthie | Brantl      | hbrantl6@nsw.gov.au            | gqH25 |
|      8 | Curry      | Roose       | croose7@so-net.ne.jp           | AUh75 |
|      9 | Fidole     | Kahan       | fkahan8@prnewswire.com         | Lua52 |
|     10 | Starlin    | Pfleger     | spfleger9@booking.com          | RSn94 |
|     11 | Romy       | Taggert     | rtaggerta@webs.com             | fx165 |
|     12 | Lola       | Sommerlie   | lsommerlieb@ebay.com           | cIi61 |
|     13 | Ephraim    | Chalke      | echalkec@thetimes.co.uk        | lp105 |
|     14 | Tess       | Kingsly     | tkingslyd@jiathis.com          | Rfy35 |
|     15 | Helga      | Rubes       | hrubese@psu.edu                | qK922 |
+--------+------------+-------------+--------------------------------+-------+
15 rows in set (0.01 sec)
```



```
mysql> SELECT * FROM health_record  LIMIT 15;
+-----------+--------------+--------+--------+--------+-----+-----+----------------+------------+
| healthID  | healthUserID | gender | weight | height | BMI | BMR | CaloriesNeeded | curr_date  |
+-----------+--------------+--------+--------+--------+-----+-----+----------------+------------+
|    710784 |          550 | M      |     97 |     60 |   7 |  15 |           7147 | 12/01/2022 |
|  20076398 |          271 | M      |    145 |     83 |  21 |  20 |           9947 | 02/09/2021 |
|  24388491 |          734 | M      |     77 |    149 |  61 |  15 |           9811 | 25/01/2022 |
|  26782553 |          505 | M      |     86 |    133 |  71 |  92 |           2263 | 16/09/2021 |
|  26957221 |          406 | F      |    107 |     54 |  76 |  99 |           2796 | 28/01/2022 |
|  49096818 |          294 | M      |     88 |    131 |  76 |  82 |           1619 | 27/07/2021 |
|  50853414 |          260 | M      |     66 |    129 |  89 |  74 |           3976 | 09/02/2022 |
|  53065034 |           92 | F      |    145 |     95 |  79 |  85 |           9880 | 01/02/2022 |
|  61510971 |           50 | F      |    134 |     95 |  67 |  43 |           8700 | 31/08/2021 |
|  63203960 |          336 | F      |     84 |    137 |  70 |  25 |           6783 | 12/06/2021 |
|  78934702 |          350 | M      |     85 |    124 |  24 |  17 |           4857 | 22/11/2021 |
|  79659802 |          956 | M      |    133 |     97 |  19 |  70 |            382 | 24/04/2021 |
|  88485749 |          912 | F      |     98 |    109 |  45 |  73 |           8359 | 02/01/2022 |
| 103536795 |          227 | M      |     83 |    121 |  58 |  39 |           5008 | 20/12/2021 |
| 123861241 |          738 | M      |    112 |     64 |  47 |   9 |           9188 | 19/08/2021 |
+-----------+--------------+--------+--------+--------+-----+-----+----------------+------------+
15 rows in set (0.00 sec)
```



```
mysql> SELECT * FROM Exercise LIMIT 15;
+------------+-----------------------------------+----------------+
| exerciseID | exerciseName                      | exerciseType   |
+------------+-----------------------------------+----------------+
|       1000 | Aerobics - Low impact             | Low intensity  |
|       1001 | Aerobics - Step                   | High intensity |
|       1002 | Archery                           | Cardio         |
|       1003 | Badminton                         | Cardio         |
|       1004 | 'Ballet, twist, jazz, tap'        | Cardio         |
|       1005 | Basketball - Shooting Hoops       | High intensity |
|       1006 | Bicycling - 12 - 14 mph moderate  | High intensity |
|       1007 | Bike - Stationary                 | High intensity |
|       1008 | Bowling                           | Low intensity  |
|       1009 | Boxing - Punching bag             | High intensity |
|       1010 | 'Carrying 16 to 24 lbs, upstairs' | High intensity |
|       1011 | Chopping Wood                     | High intensity |
|       1012 | Circuit Training                  | High intensity |
|       1013 | Cleaning House                    | Low intensity  |
|       1014 | Climbing hills                    | High intensity |
+------------+-----------------------------------+----------------+
15 rows in set (0.00 sec)
```



```
mysql> SELECT * FROM Food LIMIT 15;
+----------+------------------------------------------+---------------+-----------------+---------------+
| foodID   | foodName                                 | carbsCalories | proteinCalories | fiberCalories |
+----------+------------------------------------------+---------------+-----------------+---------------+
| 11000000 | Milk, whole                              |         70.00 |            6.89 |          1.03 |
| 11100000 | Milk, low sodium, whole                  |         51.00 |            4.87 |          3.34 |
| 11111000 | Milk, calcium fortified, whole           |         60.00 |            4.67 |          3.28 |
| 11111100 | Milk, calcium fortified, low fat (1%)    |         61.00 |            4.46 |          3.10 |
| 11111150 | Milk, calcium fortified, fat free (skim) |         60.00 |            4.67 |          3.28 |
| 11111160 | Milk, reduced fat (2%)                   |         43.00 |            5.19 |          3.38 |
| 11111170 | Milk, acidophilus, low fat (1%)          |         35.00 |            4.85 |          3.40 |
| 11112110 | Milk, acidophilus, reduced fat (2%)      |         50.00 |            4.91 |          3.35 |
| 11112120 | Milk, low fat (1%)                       |         43.00 |            5.19 |          3.38 |
| 11112130 | Milk, fat free (skim)                    |         50.00 |            4.91 |          3.35 |
| 11112210 | Milk, lactose free, low fat (1%)         |         43.00 |            5.19 |          3.38 |
| 11113000 | Milk, lactose free, fat free (skim)      |         34.00 |            4.89 |          3.43 |
| 11114300 | Milk, lactose free, reduced fat (2%)     |         43.00 |            5.19 |          3.38 |
| 11114320 | Milk, lactose free, whole                |         34.00 |            4.89 |          3.43 |
| 11114330 | Buttermilk, fat free (skim)              |         50.00 |            4.91 |          3.35 |
+----------+------------------------------------------+---------------+-----------------+---------------+
15 rows in set (0.00 sec)
```

order_record is empty and datas will be added once the program runs.

2 advanced SQL Queries:

1st Advanced SQL Query:

→ The Categories filter

In our website there will be a filter in a form of checkbox where it can display foods that has high carbs/protein/fiber content. It will be something like:

Categories

□ High Carbohydrates

□ High Protein

□ High Fiber

Our first Advanced SQL Query will be when an instance where the user pick 2 filters, like:

Categories

☑ ~~High Carbohydrates~~

☑ ~~High Protein~~

□ High Fiber

```
SELECT f1.foodName, f1.carbsCalories, f1.proteinCalories, f1.fiberCalories
FROM Food f1
WHERE f1.carbsCalories > (SELECT AVG(f2.carbsCalories)
                          FROM Food f2))
AND f1.foodName IN
(SELECT f3.foodName
FROM Food f3
WHERE f3.proteinCalories > (SELECT AVG(f4.proteinCalories)
                            FROM Food f4));
```

This is an advanced SQL Query because it uses subquery and set operation (INTERSECT), which is replaced by AND ___ IN since to achiever the INTERSECT operation functionality since INTERSECT is not available in MYSQL.

Below are another cases which is similar to the first example:

Categories

□ High Carbohydrates

☑ ~~High Protein~~

☑ ~~High Fiber~~

```
(SELECT f1.foodName, f1.carbsCalories, f1.proteinCalories, f1.fiberCalories
FROM Food f1
WHERE f1.proteinCalories > (SELECT AVG(f2.proteinCalories)
                            FROM Food f2))
AND f1.foodName IN
(SELECT f3.foodName, f3.carbsCalories, f3.proteinCalories, f3.fiberCalories
FROM Food f3
WHERE f3.fiberCalories > (SELECT AVG(f4.fiberCalories)
                          FROM Food f4))
ORDER BY foodName;
```

Categories

☑ ~~High Carbohydrates~~

□ High Protein

☑ ~~High Fiber~~

```
(SELECT f1.foodName, f1.carbsCalories, f1.proteinCalories, f1.fiberCalories
FROM Food f1
WHERE f1.carbsCalories > (SELECT AVG(f2.carbsCalories)
                          FROM Food f2))
WHERE f1.foodName, f1.carbsCalories, f1.proteinCalories, f1.fiberCalories IN
(SELECT f3.foodName, f3.carbsCalories, f3.proteinCalories, f3.fiberCalories
FROM Food f3
WHERE f3.fiberCalories > (SELECT AVG(f4.fiberCalories)
                          FROM Food f4))
ORDER BY foodName;
```

Categories

☑ ~~Carbohydrates~~

☑ ~~Protein~~

☑ ~~Fiber~~

```
(SELECT f1.foodName, f1.carbsCalories, f1.proteinCalories, f1.fiberCalories
FROM Food f1
WHERE f1.carbsCalories > (SELECT AVG(f2.carbsCalories)
                          FROM Food f2))
AND f1.foodName IN
(SELECT f3.foodName, f3.carbsCalories, f3.proteinCalories, f3.fiberCalories
FROM Food f3
WHERE f3.fiberCalories > (SELECT AVG(f4.fiberCalories)
                          FROM Food f4))
INTERSECT
(SELECT f5.foodName, f5.carbsCalories, f5.proteinCalories, f5.fiberCalories
FROM Food f5
WHERE f6.proteinCalories > (SELECT AVG(f6.proteinCalories)
                            FROM Food f6))
ORDER BY foodName;
```

2nd Advanced SQL Query:

→ This query returns the history of a user’s BMI. We are querying healthUserID = 1 as an example.

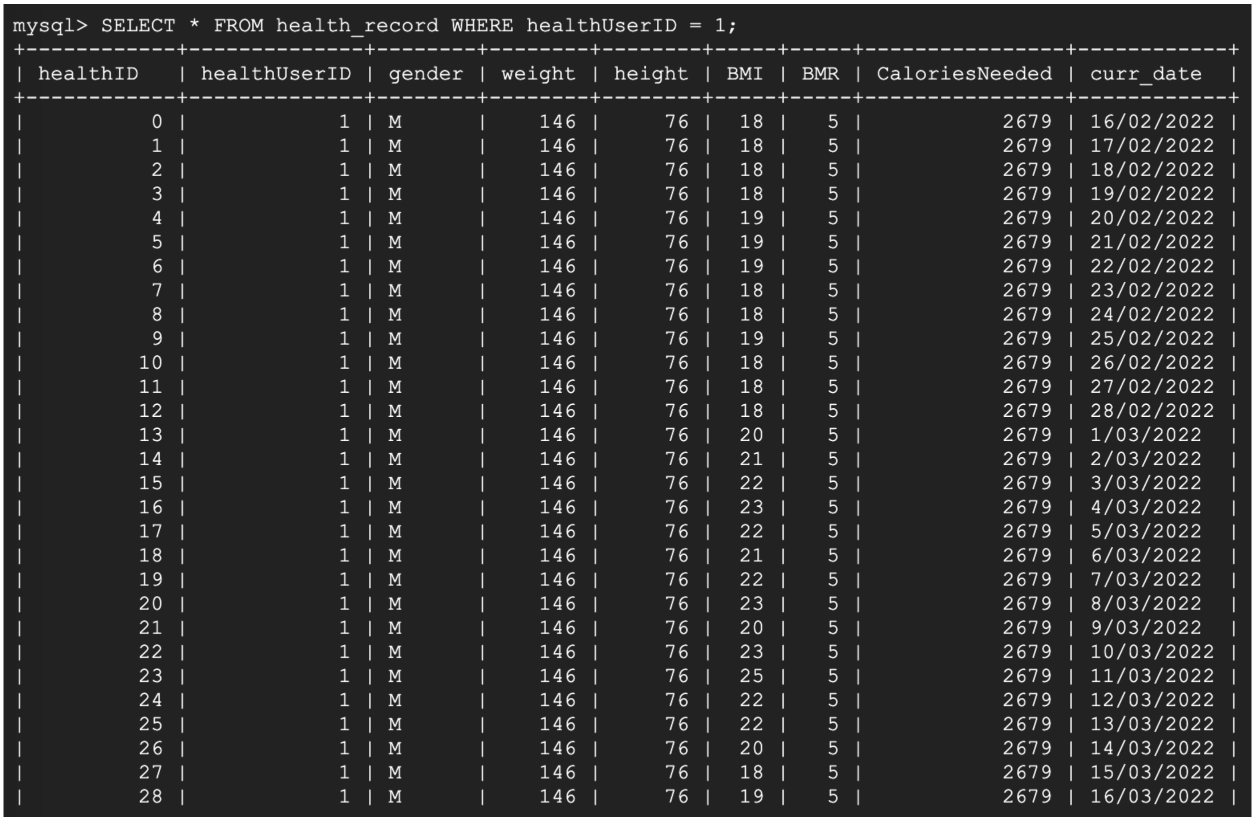
```
SELECT firstName, lastName, MAX(BMI) as maxBMI, MIN(BMI) as minBMI, AVG(BMI) as avgBMI
FROM health_record JOIN user ON (healthUserID = userID)
GROUP BY healthUserID
HAVING healthUserID = 1
```

We are joining based on userID, which is shared amongst health_record and user tables since we want to display the first and last name of the user and their BMI history on the screen. First and last name of the user information are retrieved from user table, whereas the BMI information is retrieved from health_record table.

Since health_record is auto-generated, we insert some datas into the health_record database to mimic a returning user. We are inserting to healthUserID = 1 as an example.

```
INSERT INTO health_record VALUES
(0,1,'M',146,76,18,5,2679,'16/02/2022'),
(1,1,'M',146,76,18.1,5,2679,'17/02/2022'),
(2,1,'M',146,76,18.2,5,2679,'18/02/2022'),
(3,1,'M',146,76,18.4,5,2679,'19/02/2022'),
(4,1,'M',146,76,18.5,5,2679,'20/02/2022'),
(5,1,'M',146,76,18.8,5,2679,'21/02/2022'),
(6,1,'M',146,76,18.9,5,2679,'22/02/2022'),
(7,1,'M',146,76,18.3,5,2679,'23/02/2022'),
(8,1,'M',146,76,18.4,5,2679,'24/02/2022'),
(9,1,'M',146,76,18.8,5,2679,'25/02/2022'),
(10,1,'M',146,76,18.4,5,2679,'26/02/2022'),
(11,1,'M',146,76,18,5,2679,'27/02/2022'),
(12,1,'M',146,76,18.2,5,2679,'28/02/2022'),
(13,1,'M',146,76,20,5,2679,'1/03/2022'),
(14,1,'M',146,76,21,5,2679,'2/03/2022'),
(15,1,'M',146,76,22,5,2679,'3/03/2022'),
(16,1,'M',146,76,23,5,2679,'4/03/2022'),
(17,1,'M',146,76,22,5,2679,'5/03/2022'),
(18,1,'M',146,76,21,5,2679,'6/03/2022'),
(19,1,'M',146,76,22,5,2679,'7/03/2022'),
(20,1,'M',146,76,23,5,2679,'8/03/2022'),
(21,1,'M',146,76,20,5,2679,'9/03/2022'),
(22,1,'M',146,76,23,5,2679,'10/03/2022'),
(23,1,'M',146,76,25,5,2679,'11/03/2022'),
(24,1,'M',146,76,22,5,2679,'12/03/2022'),
(25,1,'M',146,76,22,5,2679,'13/03/2022'),
(26,1,'M',146,76,20,5,2679,'14/03/2022'),
(27,1,'M',146,76,18.3,5,2679,'15/03/2022'),
(28,1,'M',146,76,18.5,5,2679,'16/03/2022'),
(29,1,'M',146,76,18.8,5,2679,'17/03/2022'),
(30,1,'M',146,76,19,5,2679,'18/03/2022'),
(31,1,'M',146,76,19.1,5,2679,'19/03/2022'),
(32,1,'M',146,76,18.3,5,2679,'20/03/2022'),
(33,1,'M',146,76,18.4,5,2679,'21/03/2022'),
(34,1,'M',146,76,19.6,5,2679,'22/03/2022'),
(35,1,'M',146,76,19.9,5,2679,'23/03/2022'),
(36,1,'M',146,76,19.8,5,2679,'24/03/2022'),
(37,1,'M',146,76,19.8,5,2679,'25/03/2022'),
(38,1,'M',146,76,18.8,5,2679,'26/03/2022'),
(39,1,'M',146,76,17.8,5,2679,'27/03/2022'),
(40,1,'M',146,76,17.8,5,2679,'28/03/2022'),
(41,1,'M',146,76,18.2,5,2679,'29/03/2022'),
(42,1,'M',146,76,18.9,5,2679,'30/03/2022'),
(43,1,'M',146,76,18.8,5,2679,'31/03/2022'),
(44,1,'M',146,76,18.5,5,2679,'1/04/2022'),
(45,1,'M',146,76,18.9,5,2679,'2/04/2022'),
(46,1,'M',146,76,18.4,5,2679,'3/04/2022'),
(47,1,'M',146,76,18.7,5,2679,'4/04/2022'),
(48,1,'M',146,76,16,5,2679,'5/04/2022'),
(49,1,'M',146,76,16,5,2679,'6/04/2022'),
(50,1,'M',146,76,16.5,5,2679,'7/04/2022');
```

Checking if the insertion is successful:

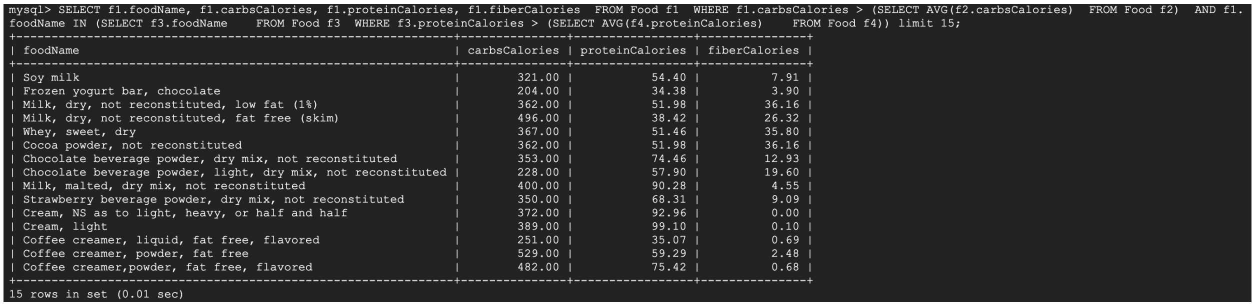


```
mysql> SELECT * FROM health_record WHERE healthUserID = 1;
+------------+--------------+--------+--------+--------+-----+-----+----------------+------------+
| healthID   | healthUserID | gender | weight | height | BMI | BMR | CaloriesNeeded | curr_date  |
+------------+--------------+--------+--------+--------+-----+-----+----------------+------------+
|          0 |            1 | M      |    146 |     76 |  18 |   5 |           2679 | 16/02/2022 |
|          1 |            1 | M      |    146 |     76 |  18 |   5 |           2679 | 17/02/2022 |
|          2 |            1 | M      |    146 |     76 |  18 |   5 |           2679 | 18/02/2022 |
|          3 |            1 | M      |    146 |     76 |  18 |   5 |           2679 | 19/02/2022 |
|          4 |            1 | M      |    146 |     76 |  19 |   5 |           2679 | 20/02/2022 |
|          5 |            1 | M      |    146 |     76 |  19 |   5 |           2679 | 21/02/2022 |
|          6 |            1 | M      |    146 |     76 |  19 |   5 |           2679 | 22/02/2022 |
|          7 |            1 | M      |    146 |     76 |  18 |   5 |           2679 | 23/02/2022 |
|          8 |            1 | M      |    146 |     76 |  18 |   5 |           2679 | 24/02/2022 |
|          9 |            1 | M      |    146 |     76 |  19 |   5 |           2679 | 25/02/2022 |
|         10 |            1 | M      |    146 |     76 |  18 |   5 |           2679 | 26/02/2022 |
|         11 |            1 | M      |    146 |     76 |  18 |   5 |           2679 | 27/02/2022 |
|         12 |            1 | M      |    146 |     76 |  18 |   5 |           2679 | 28/02/2022 |
|         13 |            1 | M      |    146 |     76 |  20 |   5 |           2679 | 1/03/2022  |
|         14 |            1 | M      |    146 |     76 |  21 |   5 |           2679 | 2/03/2022  |
|         15 |            1 | M      |    146 |     76 |  22 |   5 |           2679 | 3/03/2022  |
|         16 |            1 | M      |    146 |     76 |  23 |   5 |           2679 | 4/03/2022  |
|         17 |            1 | M      |    146 |     76 |  22 |   5 |           2679 | 5/03/2022  |
|         18 |            1 | M      |    146 |     76 |  21 |   5 |           2679 | 6/03/2022  |
|         19 |            1 | M      |    146 |     76 |  22 |   5 |           2679 | 7/03/2022  |
|         20 |            1 | M      |    146 |     76 |  23 |   5 |           2679 | 8/03/2022  |
|         21 |            1 | M      |    146 |     76 |  20 |   5 |           2679 | 9/03/2022  |
|         22 |            1 | M      |    146 |     76 |  23 |   5 |           2679 | 10/03/2022 |
|         23 |            1 | M      |    146 |     76 |  25 |   5 |           2679 | 11/03/2022 |
|         24 |            1 | M      |    146 |     76 |  22 |   5 |           2679 | 12/03/2022 |
|         25 |            1 | M      |    146 |     76 |  22 |   5 |           2679 | 13/03/2022 |
|         26 |            1 | M      |    146 |     76 |  20 |   5 |           2679 | 14/03/2022 |
|         27 |            1 | M      |    146 |     76 |  18 |   5 |           2679 | 15/03/2022 |
|         28 |            1 | M      |    146 |     76 |  19 |   5 |           2679 | 16/03/2022 |
```

This is an advanced Query because we join relations and contains an aggregation via GROUP BY.

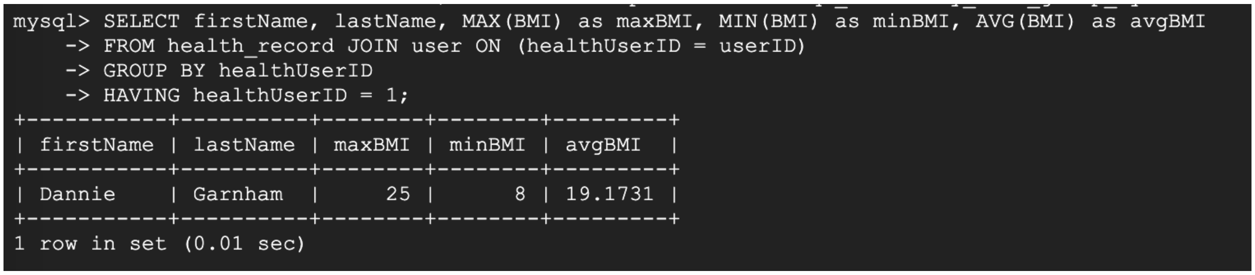
Execute your advanced SQL queries and provide a screenshot of the top 15 rows of each query result (you can use the LIMIT clause to select the top 15 rows).

— ss of top 15 of Advanced Query #1 —



```
mysql> SELECT f1.foodName, f1.carbsCalories, f1.proteinCalories, f1.fiberCalories  FROM Food f1  WHERE f1.carbsCalories > (SELECT AVG(f2.carbsCalories)  FROM Food f2)  AND f1.
foodName IN (SELECT f3.foodName    FROM Food f3  WHERE f3.proteinCalories > (SELECT AVG(f4.proteinCalories)    FROM Food f4)) limit 15;
+-------------------------------------------------------------+---------------+-----------------+---------------+
| foodName                                                    | carbsCalories | proteinCalories | fiberCalories |
+-------------------------------------------------------------+---------------+-----------------+---------------+
| Soy milk                                                    |        321.00 |           54.40 |          7.91 |
| Frozen yogurt bar, chocolate                                |        204.00 |           34.38 |          3.90 |
| Milk, dry, not reconstituted, low fat (1%)                  |        362.00 |           51.98 |         36.16 |
| Milk, dry, not reconstituted, fat free (skim)               |        496.00 |           38.42 |         26.32 |
| Whey, sweet, dry                                            |        367.00 |           51.46 |         35.80 |
| Cocoa powder, not reconstituted                             |        362.00 |           51.98 |         36.16 |
| Chocolate beverage powder, dry mix, not reconstituted       |        353.00 |           74.46 |         12.93 |
| Chocolate beverage powder, light, dry mix, not reconstituted |        228.00 |           57.90 |         19.60 |
| Milk, malted, dry mix, not reconstituted                    |        400.00 |           90.28 |          4.55 |
| Strawberry beverage powder, dry mix, not reconstituted      |        350.00 |           68.31 |          9.09 |
| Cream, NS as to light, heavy, or half and half              |        372.00 |           92.96 |          0.00 |
| Cream, light                                                |        389.00 |           99.10 |          0.10 |
| Coffee creamer, liquid, fat free, flavored                  |        251.00 |           35.07 |          0.69 |
| Coffee creamer, powder, fat free                            |        529.00 |           59.29 |          2.48 |
| Coffee creamer,powder, fat free, flavored                   |        482.00 |           75.42 |          0.68 |
+-------------------------------------------------------------+---------------+-----------------+---------------+
15 rows in set (0.01 sec)
```

— ss of top 15 of Advanced Query #2—



```
mysql> SELECT firstName, lastName, MAX(BMI) as maxBMI, MIN(BMI) as minBMI, AVG(BMI) as avgBMI
    -> FROM health_record JOIN user ON (healthUserID = userID)
    -> GROUP BY healthUserID
    -> HAVING healthUserID = 1;
+-----------+----------+--------+--------+---------+
| firstName | lastName | maxBMI | minBMI | avgBMI  |
+-----------+----------+--------+--------+---------+
| Dannie    | Garnham  |     25 |      8 | 19.1731 |
+-----------+----------+--------+--------+---------+
1 row in set (0.01 sec)
```

Advanced Query #2 only returns one row because it is querying a specific user's health records history to retrieve their maximum, minimum, and average BMI. The information we show on our website will be personalized according to which user is logged in, hence it makes sense to query one user at a time. In this case we are using healthUserID = 1 as an example.

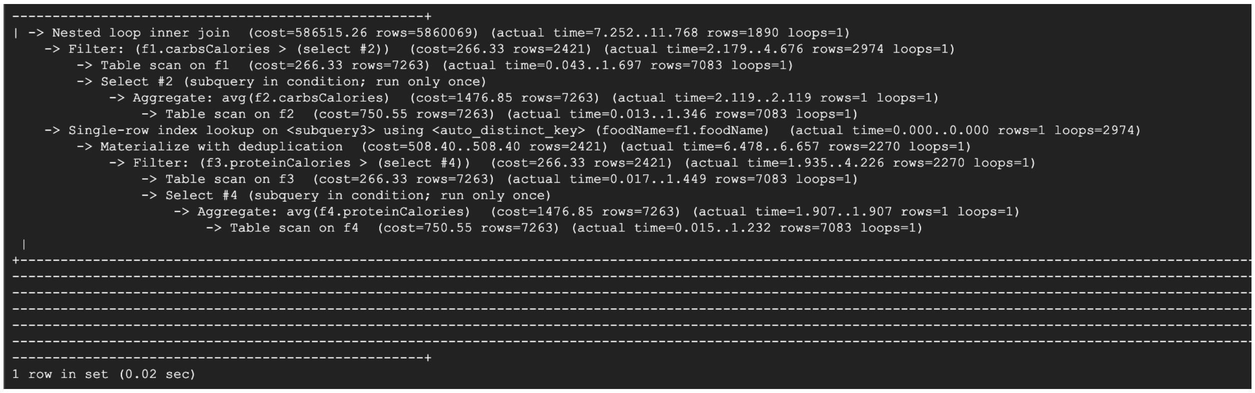
INDEXING

Advanced Query One:

```
EXPLAIN ANALYZE
SELECT f1.foodName, f1.carbsCalories, f1.proteinCalories, f1.fiberCalories
FROM Food f1
WHERE f1.carbsCalories > (SELECT AVG(f2.carbsCalories)
                          FROM Food f2)
AND f1.foodName IN (SELECT f3.foodName
                    FROM Food f3
                    WHERE f3.proteinCalories > (SELECT AVG(f4.proteinCalories)
                                                FROM Food f4));
```

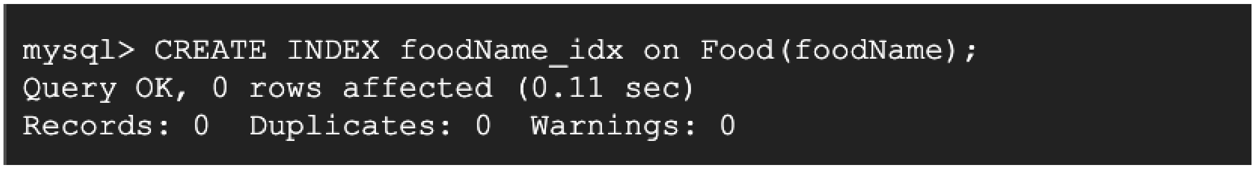
Index #1:

Without indexing:



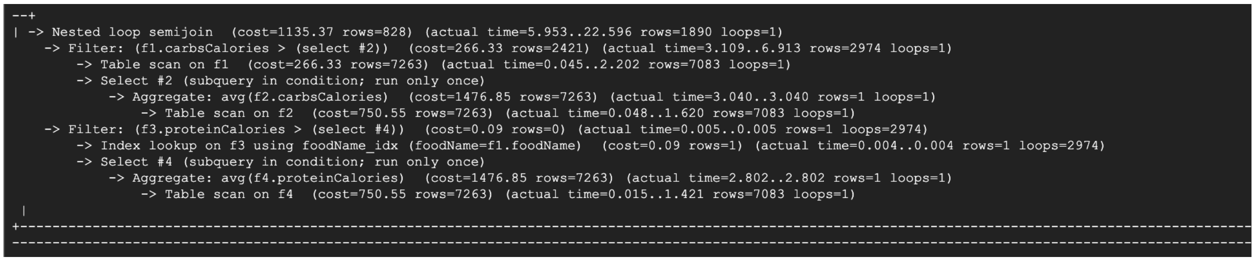
```
---------------------------------------------------+
| -> Nested loop inner join  (cost=586515.26 rows=5860069) (actual time=7.252..11.768 rows=1890 loops=1)
    -> Filter: (f1.carbsCalories > (select #2))  (cost=266.33 rows=2421) (actual time=2.179..4.676 rows=2974 loops=1)
        -> Table scan on f1  (cost=266.33 rows=7263) (actual time=0.043..1.697 rows=7083 loops=1)
        -> Select #2 (subquery in condition; run only once)
            -> Aggregate: avg(f2.carbsCalories)  (cost=1476.85 rows=7263) (actual time=2.119..2.119 rows=1 loops=1)
                -> Table scan on f2  (cost=750.55 rows=7263) (actual time=0.013..1.346 rows=7083 loops=1)
    -> Single-row index lookup on <subquery3> using <auto_distinct_key> (foodName=f1.foodName)  (actual time=0.000..0.000 rows=1 loops=2974)
        -> Materialize with deduplication  (cost=508.40..508.40 rows=2421) (actual time=6.478..6.657 rows=2270 loops=1)
            -> Filter: (f3.proteinCalories > (select #4))  (cost=266.33 rows=2421) (actual time=1.935..4.226 rows=2270 loops=1)
                -> Table scan on f3  (cost=266.33 rows=7263) (actual time=0.017..1.449 rows=7083 loops=1)
                -> Select #4 (subquery in condition; run only once)
                    -> Aggregate: avg(f4.proteinCalories)  (cost=1476.85 rows=7263) (actual time=1.907..1.907 rows=1 loops=1)
                        -> Table scan on f4  (cost=750.55 rows=7263) (actual time=0.015..1.232 rows=7083 loops=1)
 |
+------------------------------------------------------------------------------------------------------------------------------------------------------------------
------------------------------------------------------------------------------------------------------------------------------------------------------------------
------------------------------------------------------------------------------------------------------------------------------------------------------------------
------------------------------------------------------------------------------------------------------------------------------------------------------------------
------------------------------------------------------------------------------------------------------------------------------------------------------------------
------------------------------------------------------------------------------------------------------------------------------------------------------------------
---------------------------------------------------+
1 row in set (0.02 sec)
```

Creating an index on foodName:



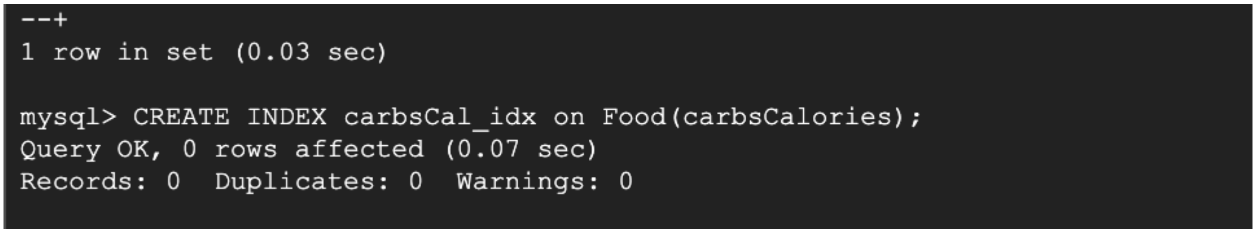
```
mysql> CREATE INDEX foodName_idx on Food(foodName);
Query OK, 0 rows affected (0.11 sec)
Records: 0  Duplicates: 0  Warnings: 0
```

After indexing:



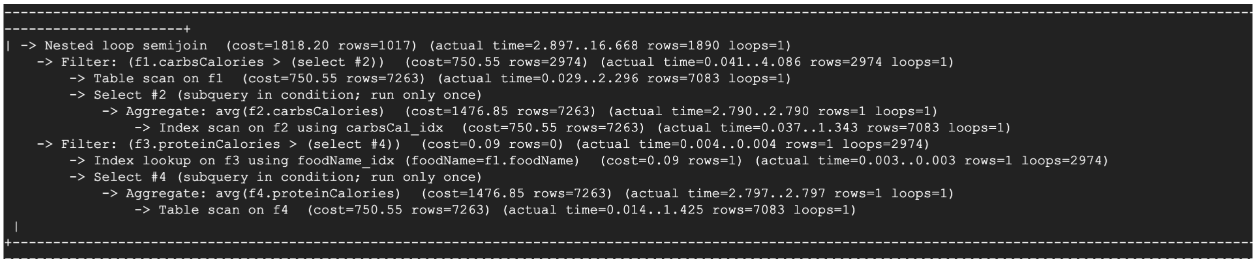
```
--+
| -> Nested loop semijoin  (cost=1135.37 rows=828) (actual time=5.953..22.596 rows=1890 loops=1)
    -> Filter: (f1.carbsCalories > (select #2))  (cost=266.33 rows=2421) (actual time=3.109..6.913 rows=2974 loops=1)
        -> Table scan on f1  (cost=266.33 rows=7263) (actual time=0.045..2.202 rows=7083 loops=1)
        -> Select #2 (subquery in condition; run only once)
            -> Aggregate: avg(f2.carbsCalories)  (cost=1476.85 rows=7263) (actual time=3.040..3.040 rows=1 loops=1)
                -> Table scan on f2  (cost=750.55 rows=7263) (actual time=0.048..1.620 rows=7083 loops=1)
    -> Filter: (f3.proteinCalories > (select #4))  (cost=0.09 rows=0) (actual time=0.005..0.005 rows=1 loops=2974)
        -> Index lookup on f3 using foodName_idx (foodName=f1.foodName)  (cost=0.09 rows=1) (actual time=0.004..0.004 rows=1 loops=2974)
        -> Select #4 (subquery in condition; run only once)
            -> Aggregate: avg(f4.proteinCalories)  (cost=1476.85 rows=7263) (actual time=2.802..2.802 rows=1 loops=1)
                -> Table scan on f4  (cost=750.55 rows=7263) (actual time=0.015..1.421 rows=7083 loops=1)
 |
+--------------------------------------------------------------------------------------------------------------------------------------------------------
---------------------------------------------------------------------------------------------------------------------------------------------------------
```

Without the indexing, the cost of the nested loop inner join is 586515.26 and it has to get through 5860069 rows, after creating an index for the food name (foodName_idx) the cost significantly drops to 1135.37 and it only has to go through 828 rows.



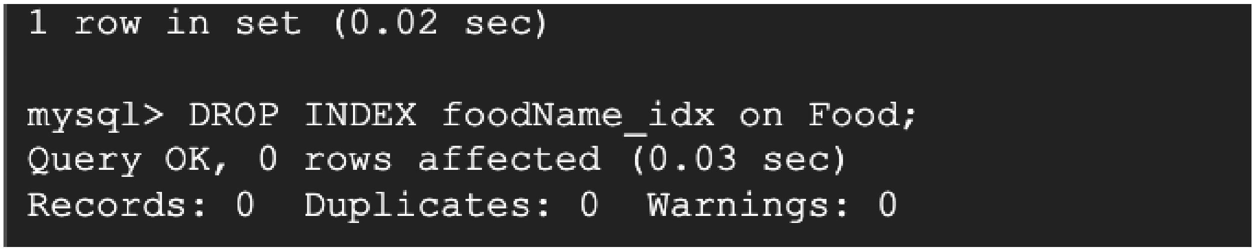
```
--+
1 row in set (0.03 sec)

mysql> CREATE INDEX carbsCal_idx on Food(carbsCalories);
Query OK, 0 rows affected (0.07 sec)
Records: 0  Duplicates: 0  Warnings: 0
```



```
---------------------------------------------------------------------------------------------------------------------------------------------------------
----------------------+
| -> Nested loop semijoin  (cost=1818.20 rows=1017) (actual time=2.897..16.668 rows=1890 loops=1)
    -> Filter: (f1.carbsCalories > (select #2))  (cost=750.55 rows=2974) (actual time=0.041..4.086 rows=2974 loops=1)
        -> Table scan on f1  (cost=750.55 rows=7263) (actual time=0.029..2.296 rows=7083 loops=1)
        -> Select #2 (subquery in condition; run only once)
            -> Aggregate: avg(f2.carbsCalories)  (cost=1476.85 rows=7263) (actual time=2.790..2.790 rows=1 loops=1)
                -> Index scan on f2 using carbsCal_idx  (cost=750.55 rows=7263) (actual time=0.037..1.343 rows=7083 loops=1)
    -> Filter: (f3.proteinCalories > (select #4))  (cost=0.09 rows=0) (actual time=0.004..0.004 rows=1 loops=2974)
        -> Index lookup on f3 using foodName_idx (foodName=f1.foodName)  (cost=0.09 rows=1) (actual time=0.003..0.003 rows=1 loops=2974)
        -> Select #4 (subquery in condition; run only once)
            -> Aggregate: avg(f4.proteinCalories)  (cost=1476.85 rows=7263) (actual time=2.797..2.797 rows=1 loops=1)
                -> Table scan on f4  (cost=750.55 rows=7263) (actual time=0.014..1.425 rows=7083 loops=1)
 |
+--------------------------------------------------------------------------------------------------------------------------------------------------------
---------------------------------------------------------------------------------------------------------------------------------------------------------
```

If we add the carbsCal_idx the cost and rows increases



```
1 row in set (0.02 sec)

mysql> DROP INDEX foodName_idx on Food;
Query OK, 0 rows affected (0.03 sec)
Records: 0  Duplicates: 0  Warnings: 0
```

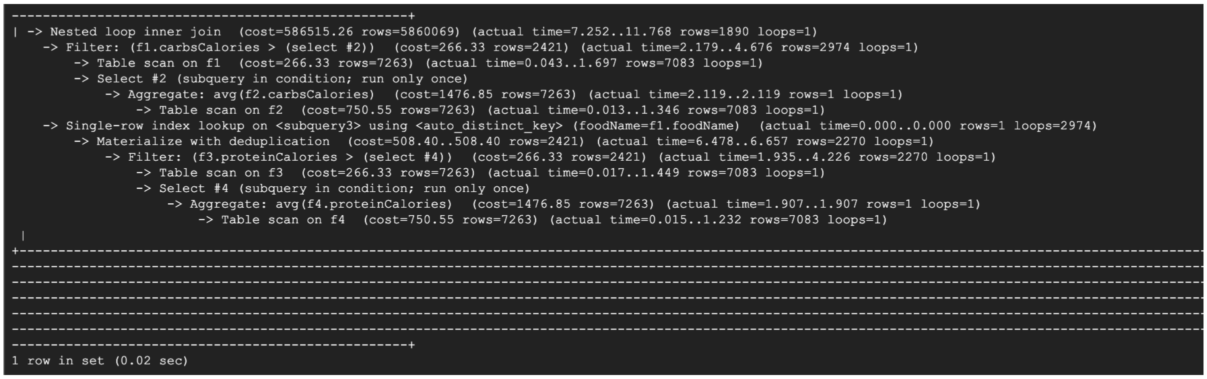
When we drop the foodname_idx, so we only have the carbsCal_idx, the cost and rows went back high.

Also if we replace carbsCalories by proteinCalories or fibreCalories it remains unchanged since the values for all three are same number of rows and the only difference is in the value of the integers so it does not account for any change in cost or number of rows used.

Thus, we know that the **drop of cost and row is driven by the foodName_idx index. So we are going with foodName_idx for this specific query for now.**

Index #2:

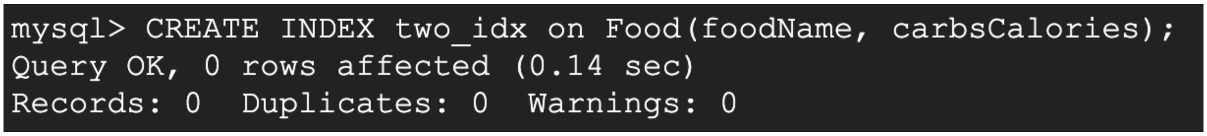
Without Indexing:



```
---------------------------------------------------+
| -> Nested loop inner join  (cost=586515.26 rows=5860069) (actual time=7.252..11.768 rows=1890 loops=1)
    -> Filter: (f1.carbsCalories > (select #2))  (cost=266.33 rows=2421) (actual time=2.179..4.676 rows=2974 loops=1)
        -> Table scan on f1  (cost=266.33 rows=7263) (actual time=0.043..1.697 rows=7083 loops=1)
        -> Select #2 (subquery in condition; run only once)
            -> Aggregate: avg(f2.carbsCalories)  (cost=1476.85 rows=7263) (actual time=2.119..2.119 rows=1 loops=1)
                -> Table scan on f2  (cost=750.55 rows=7263) (actual time=0.013..1.346 rows=7083 loops=1)
    -> Single-row index lookup on <subquery3> using <auto_distinct_key> (foodName=f1.foodName)  (actual time=0.000..0.000 rows=1 loops=2974)
        -> Materialize with deduplication  (cost=508.40..508.40 rows=2421) (actual time=6.478..6.657 rows=2270 loops=1)
            -> Filter: (f3.proteinCalories > (select #4))  (cost=266.33 rows=2421) (actual time=1.935..4.226 rows=2270 loops=1)
                -> Table scan on f3  (cost=266.33 rows=7263) (actual time=0.017..1.449 rows=7083 loops=1)
                -> Select #4 (subquery in condition; run only once)
                    -> Aggregate: avg(f4.proteinCalories)  (cost=1476.85 rows=7263) (actual time=1.907..1.907 rows=1 loops=1)
                        -> Table scan on f4  (cost=750.55 rows=7263) (actual time=0.015..1.232 rows=7083 loops=1)
 |
+----------------------------------------------------------------------------------------------------------------------------------------------------
-----------------------------------------------------------------------------------------------------------------------------------------------------
-----------------------------------------------------------------------------------------------------------------------------------------------------
-----------------------------------------------------------------------------------------------------------------------------------------------------
-----------------------------------------------------------------------------------------------------------------------------------------------------
-----------------------------------------------------------------------------------------------------------------------------------------------------
---------------------------------------------------+
1 row in set (0.02 sec)
```

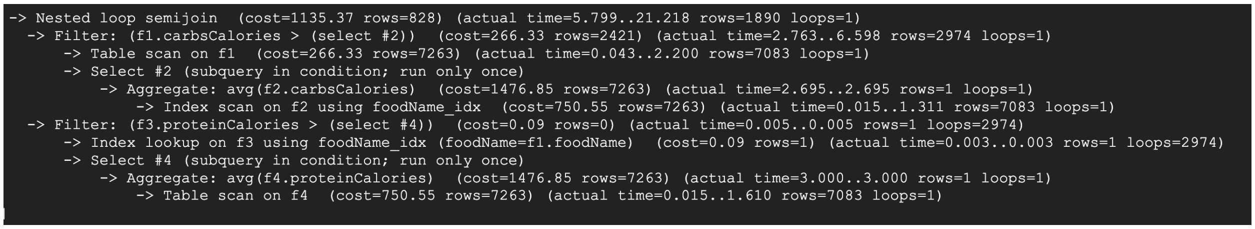
Here, we can see that the cost of the nested loop inner join is 586515.26 and it has to get through 5860069 rows.

After Indexing:



```
mysql> CREATE INDEX two_idx on Food(foodName, carbsCalories);
Query OK, 0 rows affected (0.14 sec)
Records: 0  Duplicates: 0  Warnings: 0
```

After creating an index for the food name and carbsCalories (two_idx) the cost significantly drops to 1135.37 and it only has to go through 828 rows.

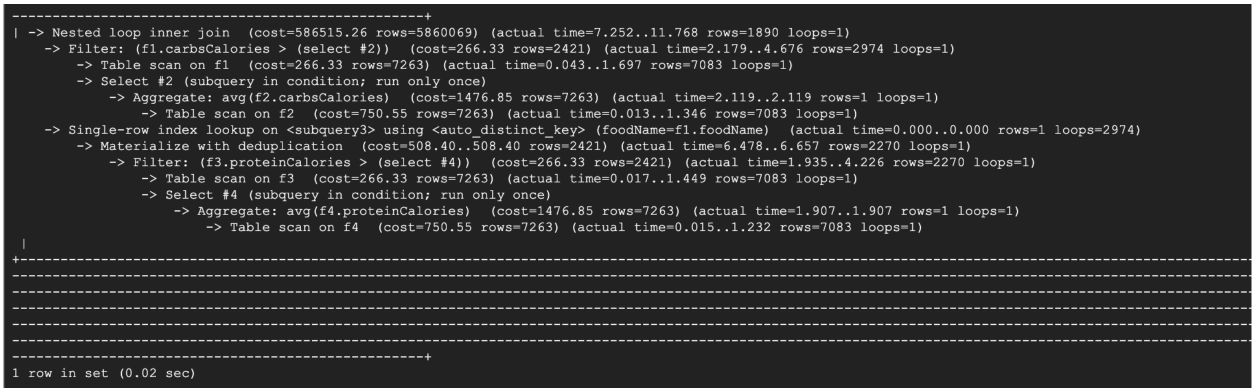


```
-> Nested loop semijoin  (cost=1135.37 rows=828) (actual time=5.799..21.218 rows=1890 loops=1)
  -> Filter: (f1.carbsCalories > (select #2))  (cost=266.33 rows=2421) (actual time=2.763..6.598 rows=2974 loops=1)
      -> Table scan on f1  (cost=266.33 rows=7263) (actual time=0.043..2.200 rows=7083 loops=1)
      -> Select #2 (subquery in condition; run only once)
          -> Aggregate: avg(f2.carbsCalories)  (cost=1476.85 rows=7263) (actual time=2.695..2.695 rows=1 loops=1)
              -> Index scan on f2 using foodName_idx  (cost=750.55 rows=7263) (actual time=0.015..1.311 rows=7083 loops=1)
  -> Filter: (f3.proteinCalories > (select #4))  (cost=0.09 rows=0) (actual time=0.005..0.005 rows=1 loops=2974)
      -> Index lookup on f3 using foodName_idx (foodName=f1.foodName)  (cost=0.09 rows=1) (actual time=0.003..0.003 rows=1 loops=2974)
      -> Select #4 (subquery in condition; run only once)
          -> Aggregate: avg(f4.proteinCalories)  (cost=1476.85 rows=7263) (actual time=3.000..3.000 rows=1 loops=1)
              -> Table scan on f4  (cost=750.55 rows=7263) (actual time=0.015..1.610 rows=7083 loops=1)
```

Our second option can be to use two_index as our index since it significantly reduces our cost and number of rows. However, since it doesn't make any difference from using just foodName_idx, we would go ahead and choose it as the index.

Index #3:

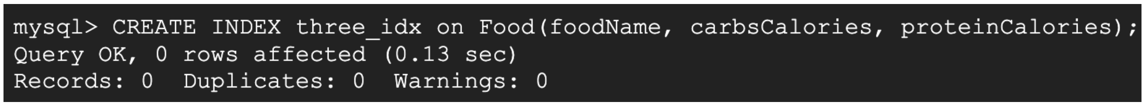
Without Indexing:



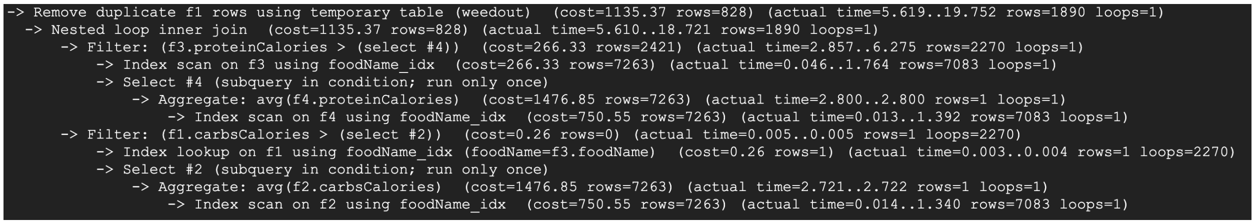
```
---------------------------------------------------+
| -> Nested loop inner join  (cost=586515.26 rows=5860069) (actual time=7.252..11.768 rows=1890 loops=1)
    -> Filter: (f1.carbsCalories > (select #2))  (cost=266.33 rows=2421) (actual time=2.179..4.676 rows=2974 loops=1)
        -> Table scan on f1  (cost=266.33 rows=7263) (actual time=0.043..1.697 rows=7083 loops=1)
        -> Select #2 (subquery in condition; run only once)
            -> Aggregate: avg(f2.carbsCalories)  (cost=1476.85 rows=7263) (actual time=2.119..2.119 rows=1 loops=1)
                -> Table scan on f2  (cost=750.55 rows=7263) (actual time=0.013..1.346 rows=7083 loops=1)
    -> Single-row index lookup on <subquery3> using <auto_distinct_key> (foodName=f1.foodName)  (actual time=0.000..0.000 rows=1 loops=2974)
        -> Materialize with deduplication  (cost=508.40..508.40 rows=2421) (actual time=6.478..6.657 rows=2270 loops=1)
            -> Filter: (f3.proteinCalories > (select #4))  (cost=266.33 rows=2421) (actual time=1.935..4.226 rows=2270 loops=1)
                -> Table scan on f3  (cost=266.33 rows=7263) (actual time=0.017..1.449 rows=7083 loops=1)
                -> Select #4 (subquery in condition; run only once)
                    -> Aggregate: avg(f4.proteinCalories)  (cost=1476.85 rows=7263) (actual time=1.907..1.907 rows=1 loops=1)
                        -> Table scan on f4  (cost=750.55 rows=7263) (actual time=0.015..1.232 rows=7083 loops=1)
 |
+---------------------------------------------------------------------------------------------------------------------------------------------------------------
----------------------------------------------------------------------------------------------------------------------------------------------------------------
----------------------------------------------------------------------------------------------------------------------------------------------------------------
----------------------------------------------------------------------------------------------------------------------------------------------------------------
----------------------------------------------------------------------------------------------------------------------------------------------------------------
----------------------------------------------------------------------------------------------------------------------------------------------------------------
---------------------------------------------------+
1 row in set (0.02 sec)
```

Here, we can see that the cost of the nested loop inner join is 586515.26 and it has to get through 5860069 rows without indexing.

After Indexing:



```
mysql> CREATE INDEX three_idx on Food(foodName, carbsCalories, proteinCalories);
Query OK, 0 rows affected (0.13 sec)
Records: 0  Duplicates: 0  Warnings: 0
```



```
-> Remove duplicate f1 rows using temporary table (weedout)  (cost=1135.37 rows=828) (actual time=5.619..19.752 rows=1890 loops=1)
  -> Nested loop inner join  (cost=1135.37 rows=828) (actual time=5.610..18.721 rows=1890 loops=1)
      -> Filter: (f3.proteinCalories > (select #4))  (cost=266.33 rows=2421) (actual time=2.857..6.275 rows=2270 loops=1)
          -> Index scan on f3 using foodName_idx  (cost=266.33 rows=7263) (actual time=0.046..1.764 rows=7083 loops=1)
          -> Select #4 (subquery in condition; run only once)
              -> Aggregate: avg(f4.proteinCalories)  (cost=1476.85 rows=7263) (actual time=2.800..2.800 rows=1 loops=1)
                  -> Index scan on f4 using foodName_idx  (cost=750.55 rows=7263) (actual time=0.013..1.392 rows=7083 loops=1)
      -> Filter: (f1.carbsCalories > (select #2))  (cost=0.26 rows=0) (actual time=0.005..0.005 rows=1 loops=2270)
          -> Index lookup on f1 using foodName_idx (foodName=f3.foodName)  (cost=0.26 rows=1) (actual time=0.003..0.004 rows=1 loops=2270)
          -> Select #2 (subquery in condition; run only once)
              -> Aggregate: avg(f2.carbsCalories)  (cost=1476.85 rows=7263) (actual time=2.721..2.722 rows=1 loops=1)
                  -> Index scan on f2 using foodName_idx  (cost=750.55 rows=7263) (actual time=0.014..1.340 rows=7083 loops=1)
```

When we index the foodName, carbsCalories, and proteinCalories, we find that it reduces the cost of the nested loop inner join significantly as compared to no indexing at all. However, the nested loop inner join cost remains the same that we get when we added index# 1 and index#2 which is 1135.37 and the rows 828. Moreover, it also adds on another cost of removing the duplicate rows using a temporary table which makes it inefficient as compared to what we get from index #1 and index#2.

Conclusion

Thus, we know that the drop is driven by indexing foodName. So we are going with indexing foodName for this specific query for now.

Advanced Query 2:

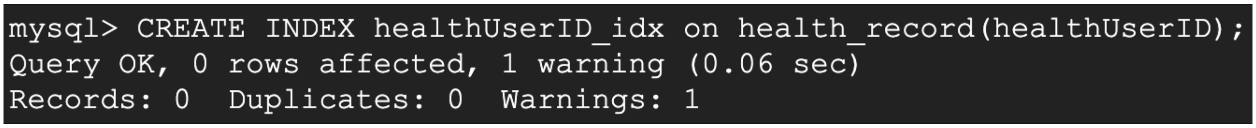
```
EXPLAIN ANALYZE
SELECT firstName, lastName, MAX(BMI) as maxBMI, MIN(BMI) as minBMI, AVG(BMI) as avgBMI
FROM health_record JOIN user ON (healthUserID = userID)
GROUP BY healthUserID
HAVING healthUserID = 1;
```

Index #1

Before indexing:

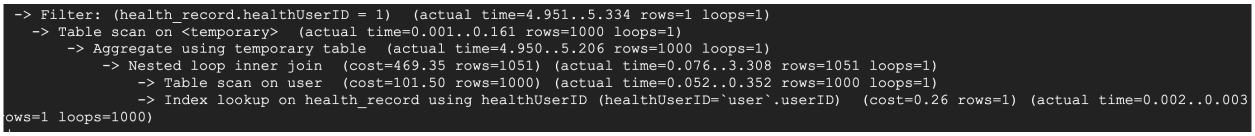
Before indexing our analysis for the table looks like this where the actual time for filter is 5.178..5.573 and the time for aggregate is 5.176..5.426

Creating an index on healthUserId as healthUserId_idx:



```
mysql> CREATE INDEX healthUserID_idx on health_record(healthUserID);
Query OK, 0 rows affected, 1 warning (0.06 sec)
Records: 0  Duplicates: 0  Warnings: 1
```

After indexing:



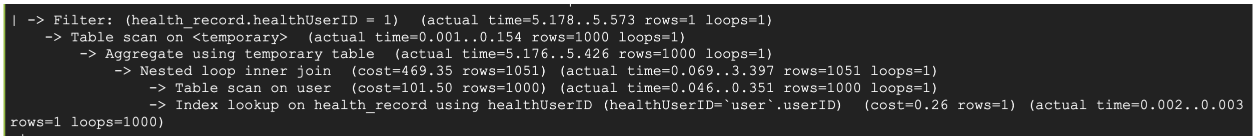
```
 -> Filter: (health_record.healthUserID = 1)  (actual time=4.951..5.334 rows=1 loops=1)
   -> Table scan on <temporary>  (actual time=0.001..0.161 rows=1000 loops=1)
       -> Aggregate using temporary table  (actual time=4.950..5.206 rows=1000 loops=1)
           -> Nested loop inner join  (cost=469.35 rows=1051) (actual time=0.076..3.308 rows=1051 loops=1)
               -> Table scan on user  (cost=101.50 rows=1000) (actual time=0.052..0.352 rows=1000 loops=1)
               -> Index lookup on health_record using healthUserID (healthUserID=`user`.userID)  (cost=0.26 rows=1) (actual time=0.002..0.003
ows=1 loops=1000)
```

After indexing our actual time for filter becomes 4.951..5.344 and the actual time for aggregate becomes 4.950..5.206

We aren’t not selecting h

Index #2:

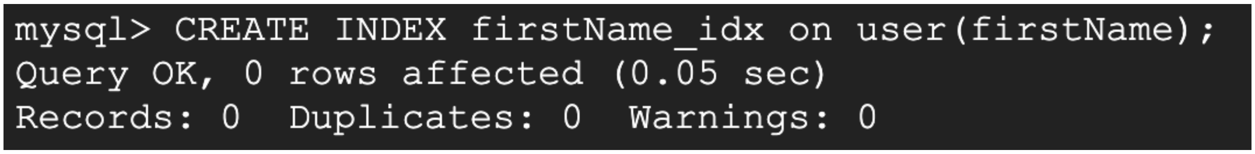
Before indexing:



```
| -> Filter: (health_record.healthUserID = 1)  (actual time=5.178..5.573 rows=1 loops=1)
    -> Table scan on <temporary>  (actual time=0.001..0.154 rows=1000 loops=1)
        -> Aggregate using temporary table  (actual time=5.176..5.426 rows=1000 loops=1)
            -> Nested loop inner join  (cost=469.35 rows=1051) (actual time=0.069..3.397 rows=1051 loops=1)
                -> Table scan on user  (cost=101.50 rows=1000) (actual time=0.046..0.351 rows=1000 loops=1)
                -> Index lookup on health_record using healthUserID (healthUserID=`user`.userID)  (cost=0.26 rows=1) (actual time=0.002..0.003
rows=1 loops=1000)
```

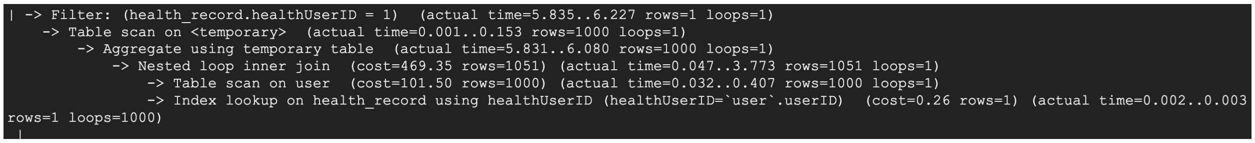
Before indexing our analysis for the table looks like this where the actual time for filter is 5.178..5.573 and the time for aggregate is 5.176..5.426

Creating index on first name as firstName_idx:



```
mysql> CREATE INDEX firstName_idx on user(firstName);
Query OK, 0 rows affected (0.05 sec)
Records: 0  Duplicates: 0  Warnings: 0
```

After indexing:



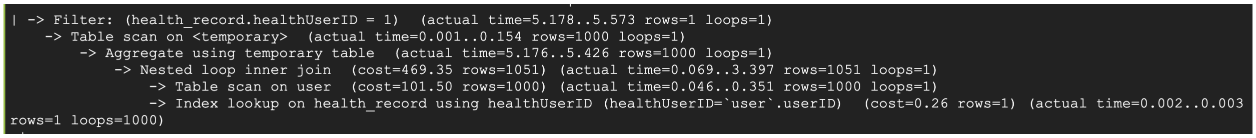
```
| -> Filter: (health_record.healthUserID = 1)  (actual time=5.835..6.227 rows=1 loops=1)
    -> Table scan on <temporary>  (actual time=0.001..0.153 rows=1000 loops=1)
        -> Aggregate using temporary table  (actual time=5.831..6.080 rows=1000 loops=1)
            -> Nested loop inner join  (cost=469.35 rows=1051) (actual time=0.047..3.773 rows=1051 loops=1)
                -> Table scan on user  (cost=101.50 rows=1000) (actual time=0.032..0.407 rows=1000 loops=1)
                -> Index lookup on health_record using healthUserID (healthUserID=`user`.userID)  (cost=0.26 rows=1) (actual time=0.002..0.003
rows=1 loops=1000)
 |
```

After indexing our actual time for filter becomes 5.835..6.227 and the actual time for aggregate becomes 5.831..6.080 which is greater than the original by a small amount

Since this index increases our time for filter and aggregate we won't be using firstName as our index for this query and firstName is a primary key so creating an index for that won’t be of any use.

Index#3:

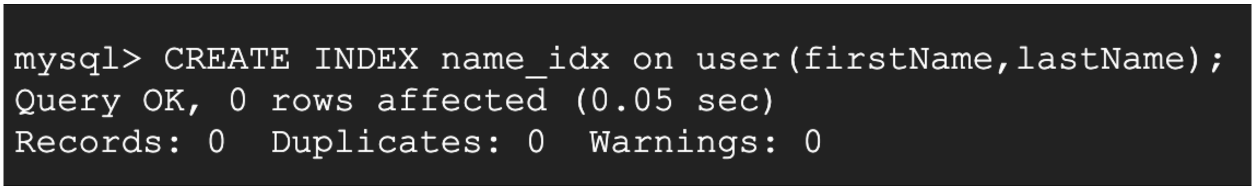
Before indexing:



```
| -> Filter: (health_record.healthUserID = 1)  (actual time=5.178..5.573 rows=1 loops=1)
    -> Table scan on <temporary>  (actual time=0.001..0.154 rows=1000 loops=1)
        -> Aggregate using temporary table  (actual time=5.176..5.426 rows=1000 loops=1)
            -> Nested loop inner join  (cost=469.35 rows=1051) (actual time=0.069..3.397 rows=1051 loops=1)
                -> Table scan on user  (cost=101.50 rows=1000) (actual time=0.046..0.351 rows=1000 loops=1)
                -> Index lookup on health_record using healthUserID (healthUserID=`user`.userID)  (cost=0.26 rows=1) (actual time=0.002..0.003
rows=1 loops=1000)
```

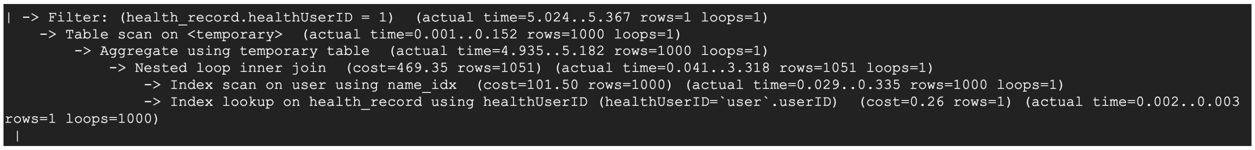
Before indexing our analysis for the table looks like this where the actual time for filter is 5.178..5.573 and the time for aggregate is 5.176..5.426

Creating index on firstName and lastName as name_idx:



```
mysql> CREATE INDEX name_idx on user(firstName,lastName);
Query OK, 0 rows affected (0.05 sec)
Records: 0  Duplicates: 0  Warnings: 0
```

After indexing:

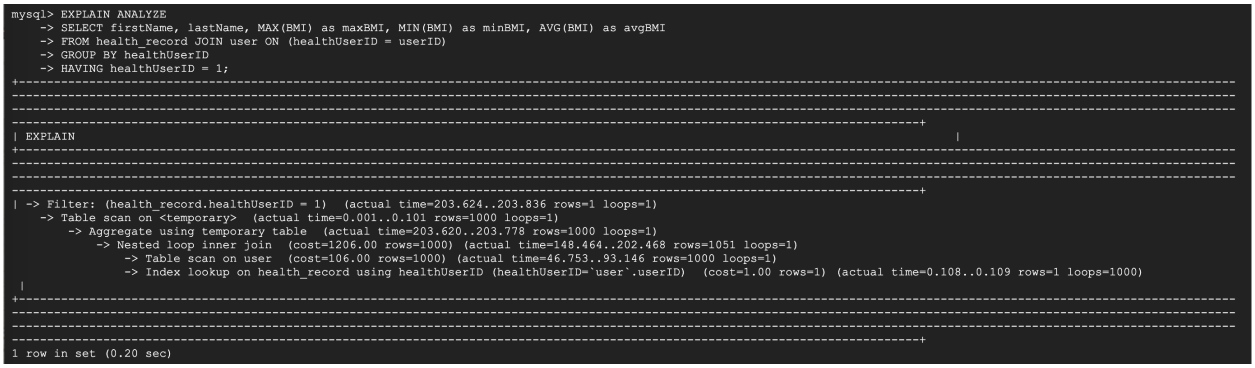


```
| -> Filter: (health_record.healthUserID = 1)  (actual time=5.024..5.367 rows=1 loops=1)
    -> Table scan on <temporary>  (actual time=0.001..0.152 rows=1000 loops=1)
        -> Aggregate using temporary table  (actual time=4.935..5.182 rows=1000 loops=1)
            -> Nested loop inner join  (cost=469.35 rows=1051) (actual time=0.041..3.318 rows=1051 loops=1)
                -> Index scan on user using name_idx  (cost=101.50 rows=1000) (actual time=0.029..0.335 rows=1000 loops=1)
                -> Index lookup on health_record using healthUserID (healthUserID=`user`.userID)  (cost=0.26 rows=1) (actual time=0.002..0.003
rows=1 loops=1000)
 |
```

After indexing our actual time for filter becomes 5.024..5.367 and the actual time for aggregate becomes 4.935..5.182 which provides us with a small change but which is better than our first index. However, creating index on firstName and lastName will be of no use as they are primary keys.

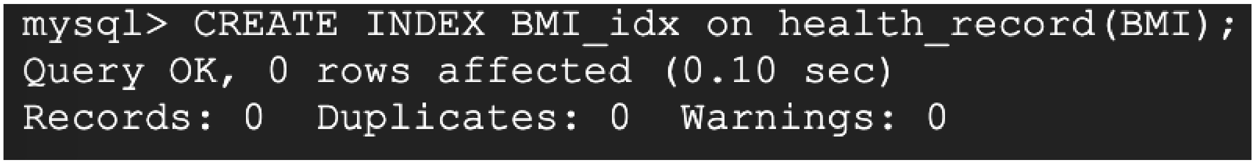
Index#4:

Before indexing



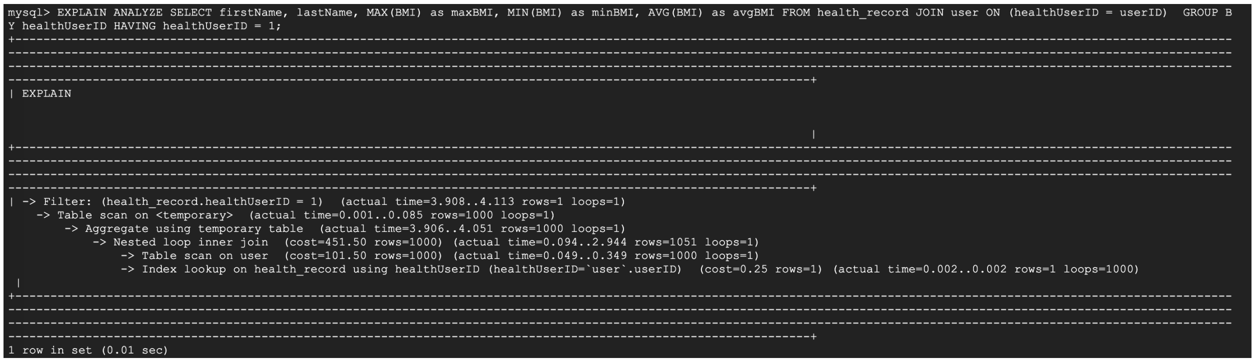
```
mysql> EXPLAIN ANALYZE
    -> SELECT firstName, lastName, MAX(BMI) as maxBMI, MIN(BMI) as minBMI, AVG(BMI) as avgBMI
    -> FROM health_record JOIN user ON (healthUserID = userID)
    -> GROUP BY healthUserID
    -> HAVING healthUserID = 1;
+-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------
------------------------------------------------------------------------------------------------------------------------------------------------------------------------------
------------------------------------------------------------------------------------------------------------------------------------------------------------------------------
---------------------------------------------------------------------------------------------------------------------------------+
| EXPLAIN                                                                                                                            |
+-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------
------------------------------------------------------------------------------------------------------------------------------------------------------------------------------
------------------------------------------------------------------------------------------------------------------------------------------------------------------------------
---------------------------------------------------------------------------------------------------------------------------------+
| -> Filter: (health_record.healthUserID = 1)  (actual time=203.624..203.836 rows=1 loops=1)
    -> Table scan on <temporary>  (actual time=0.001..0.101 rows=1000 loops=1)
        -> Aggregate using temporary table  (actual time=203.620..203.778 rows=1000 loops=1)
            -> Nested loop inner join  (cost=1206.00 rows=1000) (actual time=148.464..202.468 rows=1051 loops=1)
                -> Table scan on user  (cost=106.00 rows=1000) (actual time=46.753..93.146 rows=1000 loops=1)
                -> Index lookup on health_record using healthUserID (healthUserID=`user`.userID)  (cost=1.00 rows=1) (actual time=0.108..0.109 rows=1 loops=1000)
 |
+-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------
------------------------------------------------------------------------------------------------------------------------------------------------------------------------------
------------------------------------------------------------------------------------------------------------------------------------------------------------------------------
---------------------------------------------------------------------------------------------------------------------------------+
1 row in set (0.20 sec)
```

Create index



```
mysql> CREATE INDEX BMI_idx on health_record(BMI);
Query OK, 0 rows affected (0.10 sec)
Records: 0  Duplicates: 0  Warnings: 0
```

After indexing



```
mysql> EXPLAIN ANALYZE SELECT firstName, lastName, MAX(BMI) as maxBMI, MIN(BMI) as minBMI, AVG(BMI) as avgBMI FROM health_record JOIN user ON (healthUserID = userID)  GROUP B
Y healthUserID HAVING healthUserID = 1;
+-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------
------------------------------------------------------------------------------------------------------------------------------------------------------------------------------
------------------------------------------------------------------------------------------------------------------------------------------------------------------------------
------------------------------------------------------------------------------------------------------------------+
| EXPLAIN


                                                                                                                  |
+-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------
------------------------------------------------------------------------------------------------------------------------------------------------------------------------------
------------------------------------------------------------------------------------------------------------------------------------------------------------------------------
------------------------------------------------------------------------------------------------------------------+
| -> Filter: (health_record.healthUserID = 1)  (actual time=3.908..4.113 rows=1 loops=1)
    -> Table scan on <temporary>  (actual time=0.001..0.085 rows=1000 loops=1)
        -> Aggregate using temporary table  (actual time=3.906..4.051 rows=1000 loops=1)
            -> Nested loop inner join  (cost=451.50 rows=1000) (actual time=0.094..2.944 rows=1051 loops=1)
                -> Table scan on user  (cost=101.50 rows=1000) (actual time=0.049..0.349 rows=1000 loops=1)
                -> Index lookup on health_record using healthUserID (healthUserID=`user`.userID)  (cost=0.25 rows=1) (actual time=0.002..0.002 rows=1 loops=1000)
 |
+-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------
------------------------------------------------------------------------------------------------------------------------------------------------------------------------------
------------------------------------------------------------------------------------------------------------------------------------------------------------------------------
------------------------------------------------------------------------------------------------------------------+
1 row in set (0.01 sec)
```

After indexing BMI, the actual time for the filter becomes 3.908 and the actual time for aggregate is 3.906. Thus, we can see that out of all the indexes and without indexing the best result is provided by indexing BMI. It improves the performance of the data matching by reducing the time taken to match the query values for BMI. Hence, we would be using this index for the above listed query.

Conclusion

Hence, we can see that the most effective index was after indexing BMI which would be the best choice for us to use for the above 2nd advanced query as it helped reduce the time significantly and is the most efficient one out of all the other.

Latest SQL Code:

https://s3-us-west-2.amazonaws.com/secure.notion-static.com/8338eaee-a39d-4d82-908a-2fdfbd20b585/database_latest.sql