Project Track1 Stage3: Database Implementation and Indexing

Team008
Muzi Peng(muzip2), Weilong Li(weilong3), Rutuja Narwade(narwade2), Zhuofan Zeng(zz115)

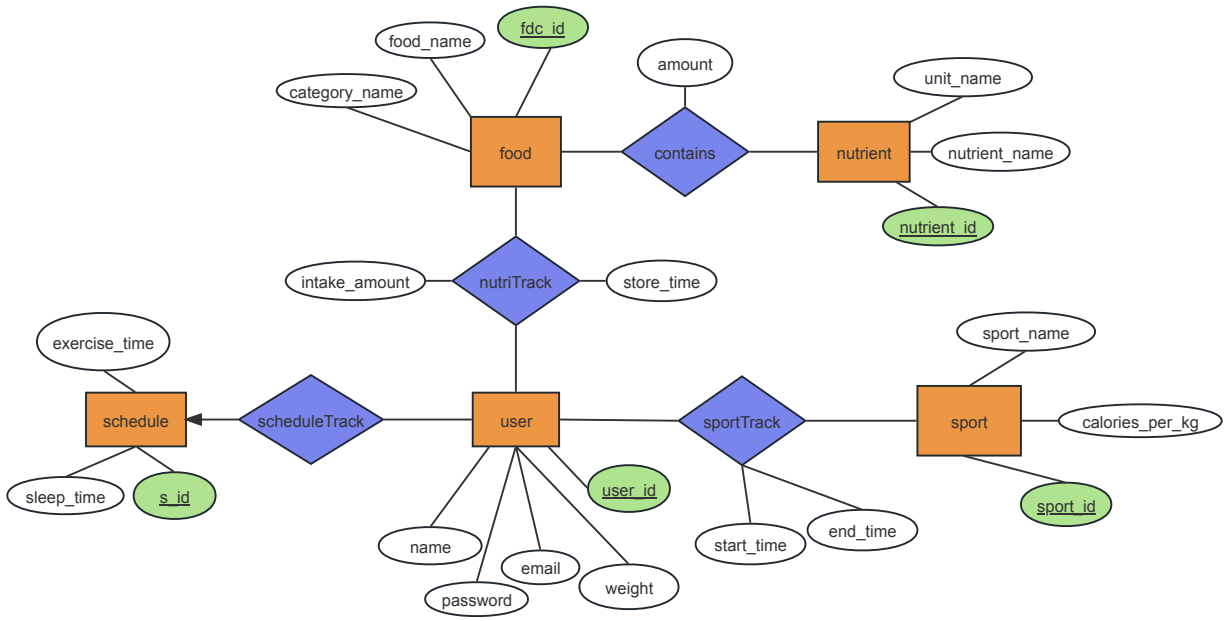


Figure 1: ER Diagram of HealthTrack database

1 Main Tables

We have 8 main tables, including user, schedule, sport, sportTrack, food, nutriTrack, nutrient, and contains.

2 Data Definition Language (DDL) commands

2.1 user table

```
CREATE TABLE user (
    user_id INT,
    name VARCHAR(255),
    email VARCHAR(255),
    password VARCHAR(255),
    weight INT,
    s_id INT,
    PRIMARY KEY (user_id),
    FOREIGN KEY (s_id) REFERENCES schedule(s_id)
);
```

2.2 schedule table

```
CREATE TABLE schedule(
    s_id INT PRIMARY KEY,
```

```
    sleep_time FLOAT,
    exercise_time FLOAT
);
```

2.3 sport table

```
CREATE TABLE sport (
    sport_id INT PRIMARY KEY,
    sport_name VARCHAR(255),
    calories_per_kg FLOAT
);
```

2.4 food table

```
CREATE TABLE food (
    fdc_id INT PRIMARY KEY,
    food_name VARCHAR(255),
    category_name VARCHAR(255)
);
```

2.5 nutrient table:

```
CREATE TABLE nutrient (
    nutrient_id INT PRIMARY KEY,
    nutrient_name VARCHAR(255),
    unit_name VARCHAR(255)
);
```

2.6 contains table

```
CREATE TABLE contains (
    fdc_id INT,
    nutrient_id INT,
    amount FLOAT,
    PRIMARY KEY (fdc_id, nutrient_id),
    FOREIGN KEY (fdc_id) REFERENCES food(fdc_id),
    FOREIGN KEY (nutrient_id) REFERENCES nutrient(nutrient_id)
);
```

2.7 nutriTrack table

```
CREATE TABLE nutriTrack (
    user_id INT,
    fdc_id INT,
    store_time TIMESTAMP,
    intake_amount FLOAT,
    PRIMARY KEY (user_id, fdc_id),
    FOREIGN KEY (user_id) REFERENCES user(user_id),
    FOREIGN KEY (fdc_id) REFERENCES food(fdc_id)
);
```

2.8 sportTrack table

```
CREATE TABLE sportTrack (
    user_id INT,
    sport_id INT,
    start_time TIMESTAMP,
    end_time TIMESTAMP,
    PRIMARY KEY (user_id, sport_id),
    FOREIGN KEY (user_id) REFERENCES user(user_id),
    FOREIGN KEY (sport_id) REFERENCES sport(sport_id)
);
```

3 Insert Data

Rows of each table are shown as follows. The requirement of inserting at least 1000 rows in three different tables is satisfied.

Table	Number of Rows
user	31
schedule	6
sport	249
sportTrack	500
food	**58733**
nutriTrack	**4000**
nutrient	478
contains	**139127**

```
mysql> SELECT COUNT(*) FROM user;
+----------+
| COUNT(*) |
+----------+
|       31 |
+----------+
1 row in set (0.01 sec)

mysql> SELECT COUNT(*) FROM schedule;
+----------+
| COUNT(*) |
+----------+
|        6 |
+----------+
1 row in set (0.08 sec)

mysql> SELECT COUNT(*) FROM sport;
+----------+
| COUNT(*) |
+----------+
|      249 |
+----------+
1 row in set (0.02 sec)

mysql> SELECT COUNT(*) FROM sportTrack;
+----------+
| COUNT(*) |
+----------+
|      500 |
+----------+
1 row in set (0.00 sec)
```

```
mysql> SELECT COUNT(*) FROM food;
+----------+
| COUNT(*) |
+----------+
|    58733 |
+----------+
1 row in set (0.02 sec)

mysql> SELECT COUNT(*) FROM nutriTrack;
+----------+
| COUNT(*) |
+----------+
|     4000 |
+----------+
1 row in set (0.09 sec)

mysql> SELECT COUNT(*) FROM nutrient;
+----------+
| COUNT(*) |
+----------+
|      478 |
+----------+
1 row in set (0.03 sec)

mysql> SELECT COUNT(*) FROM contains;
+----------+
| COUNT(*) |
+----------+
|   139127 |
+----------+
1 row in set (0.50 sec)
```

Figure 2: Results of rows obtained from SQL commands

4 Advanced SQL Queries and Results

4.1 Search food based on nutrients

This advanced query is to obtain the names of foods within the "Fruits and Fruit Juices" category that have either more than 100 unit of protein or a total vitamin content exceeding 100 unit.

```
(SELECT
    f.food_name
FROM
    food f
JOIN
    contains c ON f.fdc_id = c.fdc_id
JOIN
    nutrient n ON c.nutrient_id = n.nutrient_id
WHERE
    n.nutrient_name = 'Protein'
    AND c.amount > 100
    AND f.category_name = 'Fruits and Fruit Juices')

UNION

(SELECT
    f.food_name
FROM
    food f
JOIN
    contains c ON f.fdc_id = c.fdc_id
JOIN
    nutrient n ON c.nutrient_id = n.nutrient_id
WHERE
    n.nutrient_name LIKE 'Vitamin%'
    AND f.category_name = 'Fruits and Fruit Juices'
GROUP BY
    f.food_name
HAVING
    SUM(c.amount) > 100);
```

```
+--------------------------------------------------------------------------------+
| food_name
+--------------------------------------------------------------------------------+
| Kiwi, Pass 2, Region 4, n/a, Yes, Vitamin C - NFY010C02
| Kiwifruit, green, raw
| Melons, cantaloupe, raw
| Strawberries, raw
| BANANAS, OVERRIPE
| BANANAS, RIPE
| BANANAS, SLIGHTLY RIPE
| APPLE JUICE FROM CONCENTRATE WITH ADDED VITAMIN C, SHELF STABLE
| GRAPE JUICE, PURPLE, FROM CONCENTRATE WITH ADDED VITAMIN C, SHELF STABLE
| GRAPE JUICE, WHITE, FROM CONCENTRATE WITH ADDED VITAMIN C, SHELF STABLE
| ORANGE JUICE, REFRIGERATED, NO PULP, NOT FORTIFIED
| ORANGE JUICE, REFRIGERATED, NO PULP, NOT FORTIFIED, FROM CONCENTRATE
| GRAPEFRUIT JUICE, RED, REFRIGERATED, NOT FORTIFIED, NOT FROM CONCENTRATE
| strawberries, fresh, raw
| raspberries, fresh, raw
| blueberries, fresh, raw
| applesauce, unsweetened, with vit C added
| Pineapple, raw
```

Figure 3: result of Search food based on nutrients

4.2 Track and list nutrient intake of one user

This advanced query is to obtain the average intake per nutrient for 'Cxyvz Yynnv' between June and December 2023, focusing on nutrients starting with 'Vitamin', containing 'ose', or ending with 'Fiber'.

```
SELECT
    n.nutrient_name,
    AVG(c.amount * nt.intake_amount / 100) AS avg_intake_per_nutrient
FROM
    nutriTrack nt
JOIN
    contains c ON nt.fdc_id = c.fdc_id
JOIN
    nutrient n ON c.nutrient_id = n.nutrient_id
JOIN
    food f ON nt.fdc_id = f.fdc_id
JOIN
    user u ON nt.user_id = u.user_id
WHERE
    u.name = 'Cxyvz Yynnv' AND
    MONTH(nt.store_time) BETWEEN 6 AND 12 AND
    YEAR(nt.store_time) = 2023 AND
    n.nutrient_name LIKE 'Vitamin%' OR n.nutrient_name LIKE '%ose' OR n.nutrient_name LIKE '%Fiber%'
GROUP BY
    n.nutrient_name
ORDER BY
    avg_intake_per_nutrient DESC;
```

nutrient_name	avg_intake_per_nutrient
Vitamin C, total ascorbic acid	224.27000312805177
Total dietary fiber (AOAC 2011.25)	28.62761665001512
High Molecular Weight Dietary Fiber (HMWDF)	26.885688900096074
Fiber, total dietary	11.1444173028969
Fiber, insoluble	10.509555759297477
Fructose	7.9384766792042845
Glucose	6.964585771650858
Low Molecular Weight Dietary Fiber (LMWDF)	6.62416979530028
Fiber, soluble	3.275444456868702
Sucrose	2.686887228562562
Vitamin B-12	1.9542499324679374
Lactose	1.302208769715391
Maltose	0.2111132543273719
Verbascose	0.1169999973848462
Galactose	0.11343712002829169
Raffinose	0.020549999680370093

Figure 4: result of Track and list nutrient intake of one user

4.3 Track and rank beverages intake of 60kg+ users

This advanced query is to obtain the average daily intake amount of Beverages per user for those weighing over 60kg during June to August 2023.

```
SELECT
    u.name,
    SUM(nt.intake_amount) / COUNT(DISTINCT DATE(nt.store_time)) AS avg_daily_snack_intake_amount
FROM
    nutriTrack nt
```

```
JOIN
    food f ON nt.fdc_id = f.fdc_id
JOIN
    user u ON nt.user_id = u.user_id
WHERE
    f.category_name = 'Beverages'
    AND MONTH(nt.store_time) BETWEEN 6 AND 8
    AND YEAR(nt.store_time) = 2023
    AND u.weight > 60
GROUP BY
    u.name
ORDER BY
    avg_daily_snack_intake_amount DESC;
```

name	avg_daily_snack_intake_amount
Lidavj Qrkpsxrc	500
Vxdki Oilbi	453
Fuulfuwv Xqqovoj	399
Nlzvxagb Kiidtz	384
Mwwah Ntjfsp	365
Chnxlhkw Xqmhmxu	341
Wjmmxg Hwsqyg	329
Cfbsasx Qnbak	312.6666666666667
Mogth Xypvr	302.5
Grbcn Aolwid	283
Sbhpczlc Nolmrhout	270.8
Jiyyqu Aixchqv	257
Isgajn Jsmthj	256.5
Rawompdo Opzxkxf	245.33333333333334
Hegmrxs Krqxufytc	124

Figure 5: result of Track and rank beverages intake of 60kg+ users

4.4 Track and rank UIUC users' burnt calories through strenuous sports

This advanced query is to obtain the average daily calories burned by users under 95kg engaging in activities over 2 calories per kg, with emails containing "illinois".

```
SELECT
    st.user_id,
    SUM(s.calories_per_kg * u.weight *
        TIMESTAMPDIFF(HOUR, st.start_time, st.end_time))
        / COUNT(DISTINCT DATE(st.start_time)) AS total_calories_burned_per_day
FROM
    sportTrack st
JOIN
    sport s ON st.sport_id = s.sport_id
JOIN
    user u ON st.user_id = u.user_id
WHERE
    s.calories_per_kg > 2
    AND u.weight < 95
    AND u.email LIKE '%illinois%'
```

```
GROUP BY
    st.user_id
ORDER BY
    st.user_id;
```

user_id	total_calories_burned_per_day
3	77.83842122554779
11	161.90052088101706
13	92.42227973937989
16	112.58288383483887
18	154.45823669433594
20	133.64986765384674
25	104.27301800251007
27	96.37949895858765
28	46.34085416793823
30	79.28855538368225

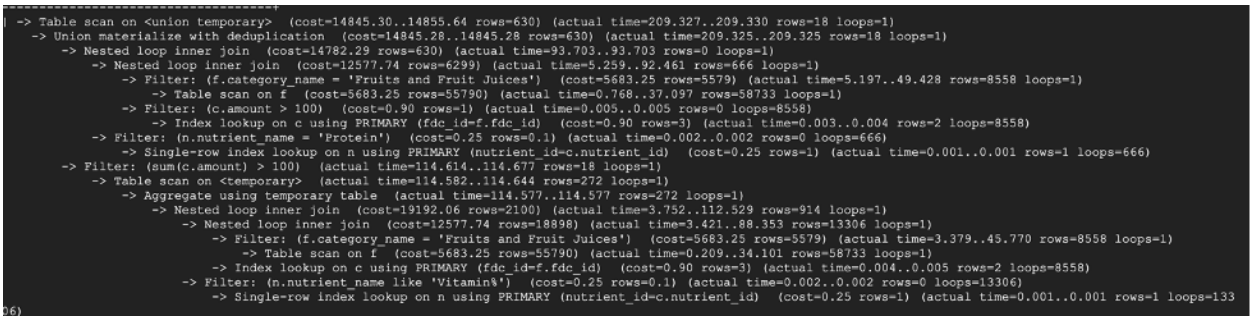
Figure 6: result of Track and rank UIUC users' burnt calories through strenuous sports (This query returns 10 rows so far, but we will add more UIUC users to our application in the future)

5 Indexing

5.1 Search food based on nutrients

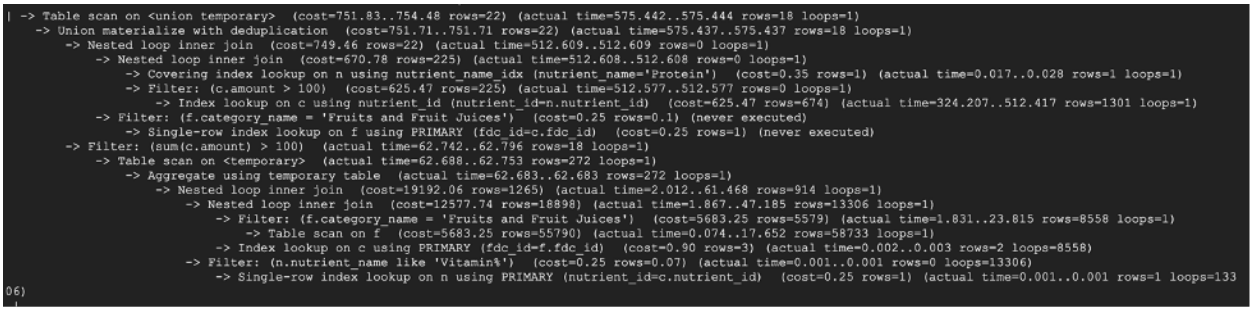
5.1.1 Config 0

The output below showcases the result of the EXPLAIN ANALYZE command of the query without any kind of indexing performed on the query attributes.



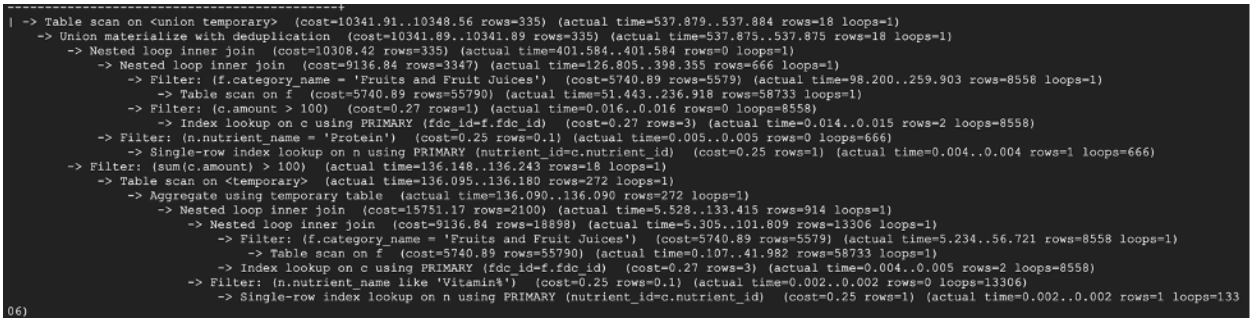
5.1.2 Config 1

```
CREATE INDEX nutrient_name_idx on nutrient(nutrient_name);
```



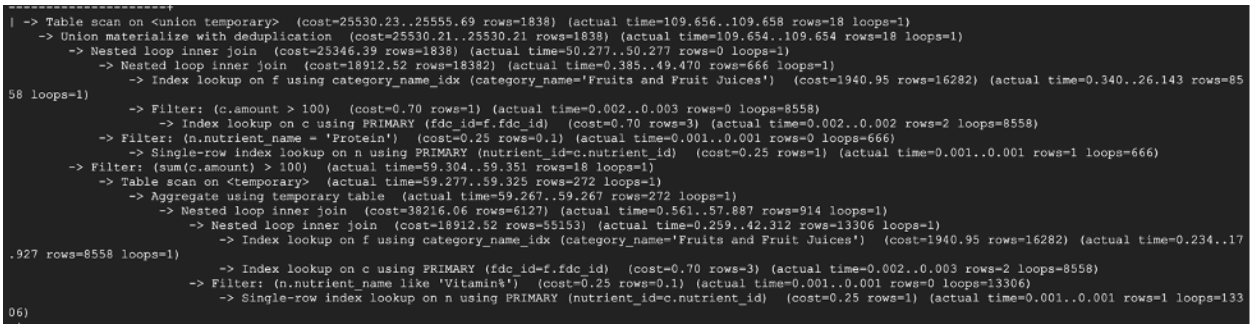
5.1.3 Config 2

```
CREATE INDEX amount_idx on contains(amount);
```



5.1.4 Config 3

```
CREATE INDEX category_name_idx on food(category_name);
```



Justification: As we can see, CREATE INDEX on nutrient(nutrient_name) greatly reduces the cost from 14800+ to 750+, this is because this index allows the database to quickly locate and retrieve the rows matching the specific nutrient names specified in the WHERE clause. Specifically, instead of scanning the entire table to find rows where nutrient_name equals 'Protein' or matches 'Vitamin', the database can directly access the relevant rows through the index, thus cutting down on the time and computational resources needed for the operation.

In contrast, CREATE INDEX on contains(amount) does not shows cost decrease. Furthermore, CREATE INDEX category on food(category_name) even further increase the cost.

Therefore, for this query, we choose to use:

```
CREATE INDEX nutrient_name_idx on nutrient(nutrient_name);
```

5.2 Track and list nutrient intake of one user

5.2.1 Config 0

The output below showcases the result of the EXPLAIN ANALYZE command of the query without any kind of indexing performed on the query attributes.

```
| -> Sort: avg_intake_per_nutrient DESC  (actual time=434.861..434.866 rows=18 loops=1)
    -> Table scan on <temporary>  (actual time=434.826..434.830 rows=18 loops=1)
        -> Aggregate using temporary table  (actual time=434.822..434.822 rows=18 loops=1)
            -> Nested loop inner join  (cost=10587.15 rows=4033) (actual time=126.482..432.887 rows=659 loops=1)
                -> Nested loop inner join  (cost=5844.84 rows=13549) (actual time=53.892..409.992 rows=10250 loops=1)
                    -> Nested loop inner join  (cost=2899.86 rows=4000) (actual time=53.838..311.094 rows=4000 loops=1)
                        -> Nested loop inner join  (cost=412.24 rows=4000) (actual time=0.768..3.404 rows=4000 loops=1)
                            -> Table scan on u  (cost=3.35 rows=31) (actual time=0.054..0.140 rows=31 loops=1)
                            -> Index lookup on nt using PRIMARY (user_id=u.user_id)  (cost=0.70 rows=129) (actual time=0.060..0.095 rows=129 loops=31)
                        -> Single-row covering index lookup on f using PRIMARY (fdc_id=nt.fdc_id)  (cost=0.52 rows=1) (actual time=0.077..0.077 rows=1 loops=4000)
                    -> Index lookup on c using PRIMARY (fdc_id=nt.fdc_id)  (cost=0.40 rows=3) (actual time=0.015..0.024 rows=3 loops=4000)
                -> Filter: (((u.`name` = 'Cxyvz Yynnv') and (month(nt.store_time) between 6 and 12) and (year(nt.store_time) = 2023) and (n.nutrient_name like 'Vita
min%')) or (n.nutrient_name like '%ose') or (n.nutrient_name like '%Fiber%'))  (cost=0.25 rows=0.3) (actual time=0.002..0.002 rows=0 loops=10250)
                    -> Single-row index lookup on n using PRIMARY (nutrient_id=c.nutrient_id)  (cost=0.25 rows=1) (actual time=0.001..0.001 rows=1 loops=10250)
 |
```

5.2.2 Config 1

The output below showcases the result of the EXPLAIN ANALYZE command of the query with an index operation done on the attribute name which increases the cost from 10587.15 to 10750.

```
CREATE INDEX user_name_index on user(name);
```

```
-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------+
| -> Sort: avg_intake_per_nutrient DESC  (actual time=411.212..411.215 rows=18 loops=1)
    -> Table scan on <temporary>  (actual time=411.084..411.090 rows=18 loops=1)
        -> Aggregate using temporary table  (actual time=411.080..411.080 rows=18 loops=1)
            -> Nested loop inner join  (cost=10750.29 rows=4033) (actual time=148.327..409.391 rows=659 loops=1)
                -> Nested loop inner join  (cost=6007.99 rows=13549) (actual time=0.098..327.240 rows=10250 loops=1)
                    -> Nested loop inner join  (cost=3063.00 rows=4000) (actual time=0.067..250.429 rows=4000 loops=1)
                        -> Nested loop inner join  (cost=412.24 rows=4000) (actual time=0.060..2.576 rows=4000 loops=1)
                            -> Covering index scan on u using user_name_index  (cost=3.35 rows=31) (actual time=0.018..0.089 rows=31 loops=1)
                            -> Index lookup on nt using PRIMARY (user_id=u.user_id)  (cost=0.70 rows=129) (actual time=0.037..0.071 rows=129 loops=31)
                        -> Single-row covering index lookup on f using PRIMARY (fdc_id=nt.fdc_id)  (cost=0.56 rows=1) (actual time=0.062..0.062 rows=1 loops=4000)
                    -> Index lookup on c using PRIMARY (fdc_id=nt.fdc_id)  (cost=0.40 rows=3) (actual time=0.006..0.019 rows=3 loops=4000)
                -> Filter: (((u.`name` = 'Cxyvz Yynnv') and (month(nt.store_time) between 6 and 12) and (year(nt.store_time) = 2023) and (n.nutrient_name like 'Vita
min%')) or (n.nutrient_name like '%ose') or (n.nutrient_name like '%Fiber%'))  (cost=0.25 rows=0.3) (actual time=0.008..0.008 rows=0 loops=10250)
                    -> Single-row index lookup on n using PRIMARY (nutrient_id=c.nutrient_id)  (cost=0.25 rows=1) (actual time=0.007..0.007 rows=1 loops=10250)
 |
```

5.2.3 Config 2

The output below showcases the result of the EXPLAIN ANALYZE command of the query with an index operation done on the attribute store_time which reduces the cost from 10587.15 to 9775.55.

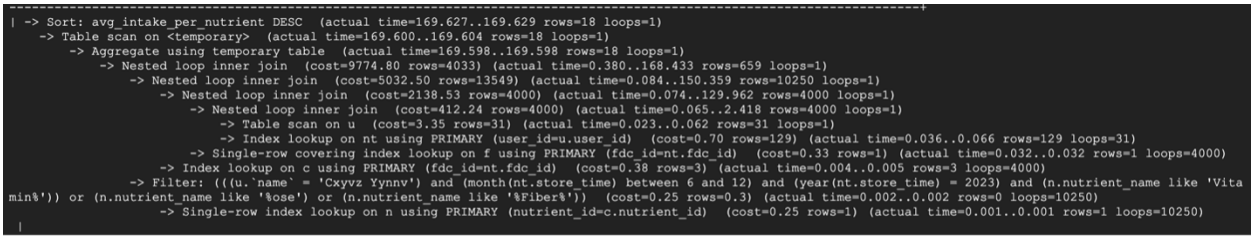
```
CREATE INDEX store_time_idx on nutriTrack(store_time);
```

```
-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------+
| -> Sort: avg_intake_per_nutrient DESC  (actual time=381.763..381.765 rows=18 loops=1)
    -> Table scan on <temporary>  (actual time=381.727..381.731 rows=18 loops=1)
        -> Aggregate using temporary table  (actual time=381.725..381.725 rows=18 loops=1)
            -> Nested loop inner join  (cost=9775.55 rows=4033) (actual time=232.412..380.746 rows=659 loops=1)
                -> Nested loop inner join  (cost=5033.25 rows=13549) (actual time=77.311..313.559 rows=10250 loops=1)
                    -> Nested loop inner join  (cost=2139.28 rows=4000) (actual time=77.241..293.930 rows=4000 loops=1)
                        -> Nested loop inner join  (cost=412.99 rows=4000) (actual time=60.222..62.348 rows=4000 loops=1)
                            -> Table scan on u  (cost=4.10 rows=31) (actual time=60.163..60.196 rows=31 loops=1)
                            -> Index lookup on nt using PRIMARY (user_id=u.user_id)  (cost=0.70 rows=129) (actual time=0.030..0.061 rows=129 loops=31)
                        -> Single-row covering index lookup on f using PRIMARY (fdc_id=nt.fdc_id)  (cost=0.33 rows=1) (actual time=0.058..0.058 rows=1 loops=4000)
                    -> Index lookup on c using PRIMARY (fdc_id=nt.fdc_id)  (cost=0.38 rows=3) (actual time=0.004..0.005 rows=3 loops=4000)
                -> Filter: (((u.`name` = 'Cxyvz Yynnv') and (month(nt.store_time) between 6 and 12) and (year(nt.store_time) = 2023) and (n.nutrient_name like 'Vita
min%')) or (n.nutrient_name like '%ose') or (n.nutrient_name like '%Fiber%'))  (cost=0.25 rows=0.3) (actual time=0.006..0.006 rows=0 loops=10250)
                    -> Single-row index lookup on n using PRIMARY (nutrient_id=c.nutrient_id)  (cost=0.25 rows=1) (actual time=0.006..0.006 rows=1 loops=10250)
 |
```

5.2.4 Config 3

The output below showcases the result of the EXPLAIN ANALYZE command of the query with an index operation done on the attribute nutrient_name which reduces the cost from 10587.15 to 9774.8.

```
CREATE INDEX nutrient_name_idx on nutrient(nutrient_name);
```



Justification: These CREATE INDEX commands create indexes that are based on the name column of the user table. This query involves filtering the records based on the user name, i.e. WHERE u.name = 'Cxyvz Yynnv'. Without the index, the database would need to examine every record in the users table to find users with names matching "Cxyvz Yynnv". If the users table contains a large number of records, this would be a very time-consuming operation.

By creating an index on the name column and other attribute columns belongs to where clause, the database can use these indexes to quickly locate records with i.e. the name "Cxyvz Yynnv" without scanning the entire table. As a result, other parts of the query, such as joins and filters, can be executed faster because the database has quickly narrowed down the dataset for further processing.

Therefore, we choose the three attributes used for creating index.

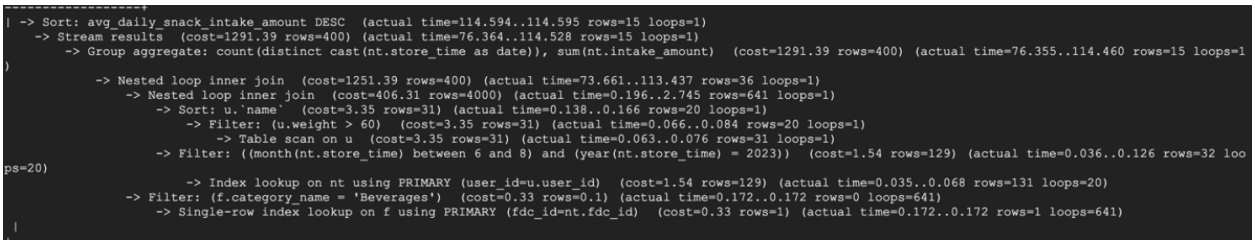
But we find that the cost increases by indexing user_name here, because it adds additional storage requirements and can affect the database's write performance. I guess the increase of cost when indexing user_name because if the column of the index is low selectivity, that is, many rows in the column have the same value, then the effectiveness of the index is greatly reduced. In this case, the database may still need to scan a large number of index items to find the required data, which may not be more efficient than a full table scan. Less selective indexing not only fails to provide performance benefits, but also adds additional storage and maintenance costs.

5.3 Track and rank beverages intake of 60kg+ users

As is shown below, the indexing performed on the attribute 'weight' from the table 'user' significantly reduced the cost of the query. By creating an index on the specified column, it aids in the quick execution of the query on the database which includes complex operations such as multiple joins.

5.3.1 Config 0

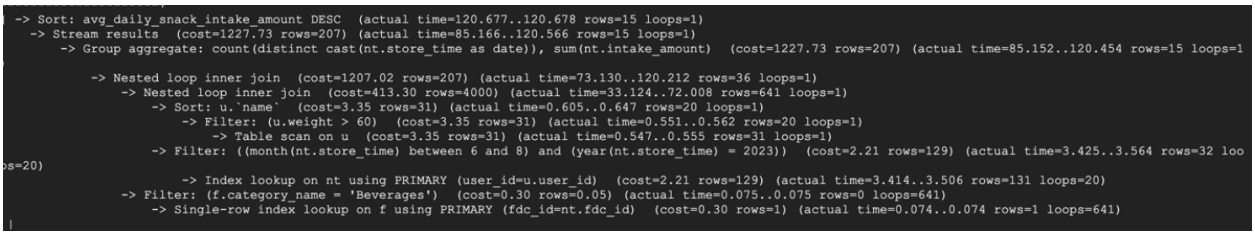
The output below showcases the result of the EXPLAIN ANALYZE command of the query without any kind of indexing performed on the query attributes.



5.3.2 Config 1

The output below showcases the result of the EXPLAIN ANALYZE command of the query with an index operation done on the attribute category_name which reduced the cost from 1291 to 1227.

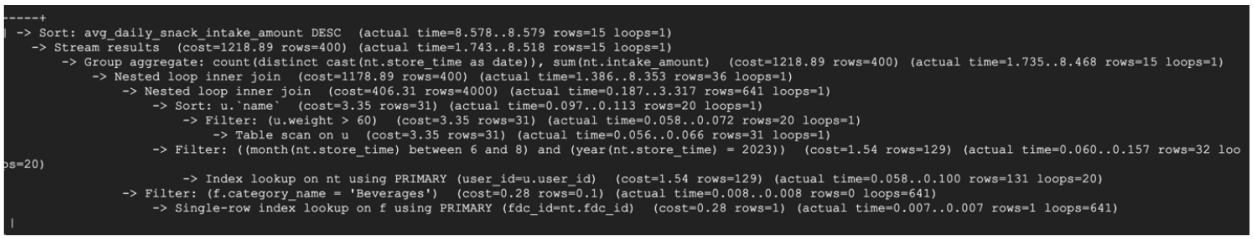
```
CREATE INDEX category_name_idx on food(category_name);
```



5.3.3 Config 2

The output below showcases the result of the EXPLAIN ANALYZE command of the query with an index operation done on the attribute category_name which reduced the cost from 1291 to 1218.

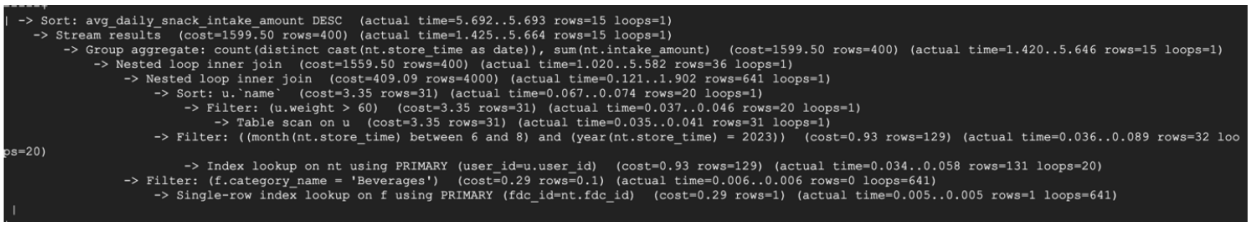
```
CREATE INDEX store_time_idx on nutriTrack(store_time);
```



5.3.4 Config 3

The output below showcases the result of the EXPLAIN ANALYZE command of the query with an index operation done on the attribute category_name which reduced the cost from 1291 to 1599.

```
CREATE INDEX weight_idx on user(weight);
```



Justification: In this query, the WHERE clause is used to filter the records for specific conditions: it qualifies the category_name in the food table to be 'Beverages', the store_time month of the records in the nutriTrack table to be between 6 and 8, the year to be 2023, and the weight in the user table to be greater than 60.

For category_name: The food table may contain a large number of food records in different categories. Creating an index on the category_name column allows the database to quickly locate records with the category 'Beverages', avoiding a full table scan.

For store_time: the nutriTrack table records the time of the user's food intake. Indexing this column (especially if year and month are considered) can quickly filter out records that match the time range. For time-related queries, a time index is particularly useful because it can quickly trim out data within the query time range.

For weight: The weight column in the user table is used to filter users who weigh more than 60 pounds. Creating an index on this column can speed up this filtering process, especially when the user table has a large amount of data.

Therefore, we choose the three attributes used for creating index.

But we find that the cost increases by indexing weight here, because it adds additional storage requirements and can affect the database's write performance. I guess it is because if the column of the index is low selectivity, that is, many rows in the column have the same value, then the effectiveness of the index is greatly reduced. In this case, the database may still need to scan a large number of index items to find the required data, which may not be more efficient than a full table scan. Less selective indexing not only fails to provide performance benefits, but also adds additional storage and maintenance costs.

5.4 Track and rank UIUC users' burnt calories through strenuous sports

5.4.1 Config 0

The output below showcases the result of the EXPLAIN ANALYZE command of the query without any kind of indexing performed on the query attributes.

```
| -> Group aggregate: count(distinct tmp_field), sum(tmp_field)  (actual time=0.920..0.930 rows=10 loops=1)
    -> Sort: st.user_id  (actual time=0.893..0.895 rows=32 loops=1)
        -> Stream results  (cost=11.98 rows=6) (actual time=0.534..0.869 rows=32 loops=1)
            -> Nested loop inner join  (cost=11.98 rows=6) (actual time=0.519..0.831 rows=32 loops=1)
                -> Nested loop inner join  (cost=5.50 rows=19) (actual time=0.479..0.600 rows=160 loops=1)
                    -> Filter: ((u.weight < 95) and (u.email like '%illinois%'))  (cost=3.35 rows=1) (actual time=0.441..0.470 rows=10 loops=1)
                        -> Table scan on u  (cost=3.35 rows=31) (actual time=0.429..0.435 rows=31 loops=1)
                    -> Index lookup on st using PRIMARY (user_id=u.user_id)  (cost=1.66 rows=16) (actual time=0.009..0.011 rows=16 loops=10)
                -> Filter: (s.calories_per_kg > 2)  (cost=0.25 rows=0.3) (actual time=0.001..0.001 rows=0 loops=160)
                    -> Single-row index lookup on s using PRIMARY (sport_id=st.sport_id)  (cost=0.25 rows=1) (actual time=0.001..0.001 rows=1 loops=160)
 |
```

5.4.2 Config 1

The output below showcases the result of the EXPLAIN ANALYZE command of the query with an index operation done on the attribute category_name which resulted in the cost being the same as no indexing.

```
CREATE INDEX calories_per_kg_idx on sport(calories_per_kg);
```

```
| -> Group aggregate: count(distinct tmp_field), sum(tmp_field)  (actual time=0.940..0.950 rows=10 loops=1)
    -> Sort: st.user_id  (actual time=0.909..0.911 rows=32 loops=1)
        -> Stream results  (cost=11.98 rows=3) (actual time=0.570..0.884 rows=32 loops=1)
            -> Nested loop inner join  (cost=11.98 rows=3) (actual time=0.555..0.846 rows=32 loops=1)
                -> Nested loop inner join  (cost=5.50 rows=19) (actual time=0.519..0.628 rows=160 loops=1)
                    -> Filter: ((u.weight < 95) and (u.email like '%illinois%'))  (cost=3.35 rows=1) (actual time=0.464..0.481 rows=10 loops=1)
                        -> Table scan on u  (cost=3.35 rows=31) (actual time=0.453..0.459 rows=31 loops=1)
                    -> Index lookup on st using PRIMARY (user_id=u.user_id)  (cost=1.66 rows=16) (actual time=0.011..0.013 rows=16 loops=10)
                -> Filter: (s.calories_per_kg > 2)  (cost=0.25 rows=0.2) (actual time=0.001..0.001 rows=0 loops=160)
                    -> Single-row index lookup on s using PRIMARY (sport_id=st.sport_id)  (cost=0.25 rows=1) (actual time=0.001..0.001 rows=1 loops=160)
 |
```

5.4.3 Config 2

The output below showcases the result of the EXPLAIN ANALYZE command of the query with an index operation done on the attribute category_name which resulted in an increase in the cost from 11.98 to 29.23.

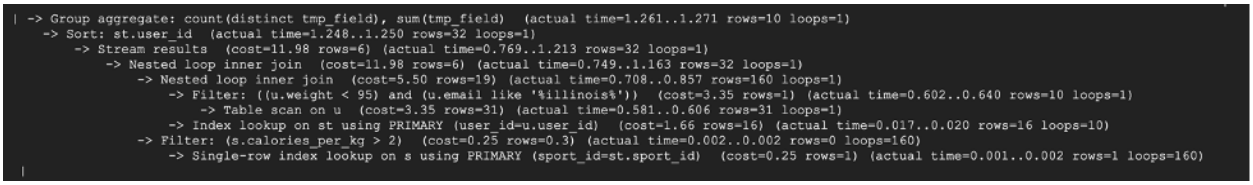
```
CREATE INDEX weight on user(weight);
```

```
| -> Group aggregate: count(distinct tmp_field), sum(tmp_field)  (actual time=0.580..0.590 rows=10 loops=1)
    -> Sort: st.user_id  (actual time=0.528..0.530 rows=32 loops=1)
        -> Stream results  (cost=29.23 rows=19) (actual time=0.114..0.508 rows=32 loops=1)
            -> Nested loop inner join  (cost=29.23 rows=19) (actual time=0.102..0.473 rows=32 loops=1)
                -> Nested loop inner join  (cost=9.79 rows=56) (actual time=0.066..0.195 rows=160 loops=1)
                    -> Filter: ((u.weight < 95) and (u.email like '%illinois%'))  (cost=3.35 rows=3) (actual time=0.041..0.060 rows=10 loops=1)
                        -> Table scan on u  (cost=3.35 rows=31) (actual time=0.033..0.040 rows=31 loops=1)
                    -> Index lookup on st using PRIMARY (user_id=u.user_id)  (cost=0.72 rows=16) (actual time=0.010..0.012 rows=16 loops=10)
                -> Filter: (s.calories_per_kg > 2)  (cost=0.25 rows=0.3) (actual time=0.002..0.002 rows=0 loops=160)
                    -> Single-row index lookup on s using PRIMARY (sport_id=st.sport_id)  (cost=0.25 rows=1) (actual time=0.001..0.001 rows=1 loops=160)
 |
```

5.4.4 Config 3

The output below showcases the result of the EXPLAIN ANALYZE command of the query with an index operation done on the attribute category_name which resulted in the cost being the same as no indexing.

```
CREATE INDEX email_idx on user(email);
```



Justification: As we can see, creating indexes on `sport(calories_per_kg)` and `user(email)` doesn't reduce the query cost in this scenario primarily. We think it may because the filtering based on these columns (`calories_per_kg > 2` and `email LIKE '%illinois%'`) likely doesn't sufficiently narrow down the result set. If the majority of records in `sport` meet the `calories_per_kg` condition and a significant portion of `user` records contain 'illinois' in their emails, the database still ends up scanning a large number of rows.

Therefore, for this query, we don't use any index.