





Filter Enter property name or value									
<input type="checkbox"/>	Status	Instance ID  	Issues	Cloud SQL edition	Type	Public IP address	Private IP address	Instance connection name	Actions
<input type="checkbox"/>		wisteria-sql	<div>Allows unencrypted direct connections</div> <div>Auditing not enabled</div> <div>SHOW 2 MORE</div>	Enterprise	MySQL 8.0	34.30.254.32		cs411-sp25-wisteria-us-	

USER:

Data tables being used:

DDL:

```
CREATE TABLE userData (
  UserId INT NOT NULL PRIMARY KEY,
  FirstName VARCHAR(255),
  LastName VARCHAR(255),
  EmailId VARCHAR(255),
  PasswordField VARCHAR(255),
  UserLocationId INT,
  FOREIGN KEY (UserLocationId) REFERENCES locationData(LocationId)
);
```

LOCATION:

Data tables being used:

<https://www.kaggle.com/datasets/liewyousheng/geolocation/data> (new location)

DDL:

```
CREATE TABLE locationData ( LocationId INT, Latitude DECIMAL(8,5), Longitude DECIMAL(8,5), City VARCHAR(255), Country VARCHAR(255))
```

PRODUCT:

Data tables being used:

US food imports

<https://www.ers.usda.gov/topics/international-markets-us-trade/us-agricultural-trade/data>

Relevant table: U.S. Food Imports

DDL:

```
CREATE TABLE productData (  
    ProductId INT NOT NULL PRIMARY KEY,  
    ProductName VARCHAR(255) NOT NULL,  
    FoodGroup VARCHAR(255),  
    LocationId INT,  
    EC_Id INT,  
    FOREIGN KEY (LocationId) REFERENCES locationData(LocationId),  
    FOREIGN KEY (EC_Id) REFERENCES environmentalCost(EC_Id)  
);
```

GROCERY PRODUCT:

Data tables being used:

DDL:

```
CREATE TABLE groceryProduct(UserId INT, ProductId INT, Quantity INT)
```

ENVIRONMENTAL COST:

Data tables being used:

Environment Impact of Food Production(kaggle)

DDL:

```
CREATE TABLE environmentalCost(EC_Id INT, ProductName VARCHAR(255),  
    CarbonFootprint_per_kg DECIMAL(6,2), LandUse_per_kg DECIMAL(6,2),  
    WaterUse_per_kg DECIMAL(8,2), Eutrophication_per_kg DECIMAL(6,2), Processing  
    GH DECIMAL(8,3), Transport GH DECIMAL(8,3), Packaging GH DECIMAL(8,3), Retailing  
    GH DECIMAL(8,3), TotalEmissions DECIMAL(8,3))
```

DATA PROOF:

We populated the environmentalCost, locationData, and productData tables with 1000+ rows each.

Editor 1

RUN

FORMAT

CLEAR

Valid

1 SELECT COUNT(*) FROM environmentalCost;

RESULTS

COUNT(*)

1000

Editor 1

RUN

FORMAT

CLEAR

Valid

1 SELECT COUNT(*) FROM locationData;

RESULTS

COUNT(*)

20069

Editor 1

RUN

FORMAT

CLEAR

Valid

1 SELECT COUNT(*) FROM productData;

RESULTS

COUNT(*)

1050

We also input some dummy user data in the userData and groceryProduct tables:

Editor 1

RUN

FORMAT

CLEAR

Valid

1 SELECT * FROM userData;

RESULTS

UserId	FirstName	LastName	EmailId
1	Alice	Green	alice@example.com
2	Bob	Brown	bob@example.com
3	Charlie	Black	charlie@example.com
4	David	White	david@example.com
5	Eve	Gray	eve@example.com
6	Frank	Blue	frank@example.com
7	Grace	Red	grace@example.com
8	Henry	Yellow	henry@example.com
9	Ivy	Pink	ivy@example.com
10	Jack	Orange	jack@example.com
11	Karen	Purple	karen@example.com
12	Leo	Silver	leo@example.com

Editor 1

RUN

FORMAT

CLEAR

Valid

1 SELECT * FROM userData;

RESULTS

ADVANCED QUERIES

1) Products with a high carbon footprint or water usage

CONCEPTS USED:

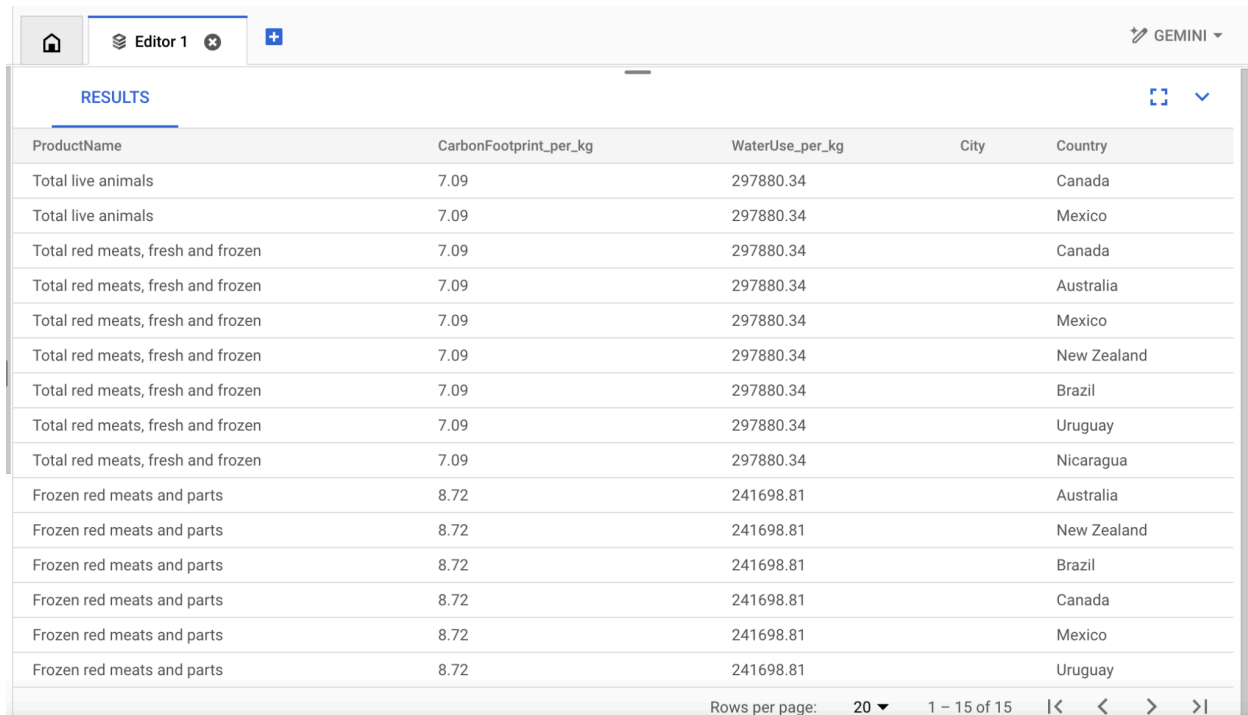
1. Set operator (union)
2. Multiple JOINS (productData, environmentalCost, locationData)

QUERY:

```
(
SELECT
    p.ProductName,
    e.CarbonFootprint_per_kg,
    e.WaterUse_per_kg,
    l.City,
    l.Country
FROM productData AS p
JOIN environmentalCost AS e
    ON p.EC_Id = e.EC_Id
JOIN locationData AS l
    ON p.LocationId = l.LocationId
WHERE e.CarbonFootprint_per_kg > 5
)
UNION
(
SELECT
    p.ProductName,
    e.CarbonFootprint_per_kg,
    e.WaterUse_per_kg,
    l.City,
    l.Country
FROM productData AS p
JOIN environmentalCost AS e
    ON p.EC_Id = e.EC_Id
JOIN locationData AS l
    ON p.LocationId = l.LocationId
WHERE e.WaterUse_per_kg > 100
)
```

)
LIMIT 15;

PROOF:



The screenshot shows a database query results interface. At the top, there's a header bar with a home icon, 'Editor 1', a plus icon, and 'GEMINI'. Below this is a 'RESULTS' section with a table. The table has five columns: 'ProductName', 'CarbonFootprint_per_kg', 'WaterUse_per_kg', 'City', and 'Country'. The data is as follows:

ProductName	CarbonFootprint_per_kg	WaterUse_per_kg	City	Country
Total live animals	7.09	297880.34		Canada
Total live animals	7.09	297880.34		Mexico
Total red meats, fresh and frozen	7.09	297880.34		Canada
Total red meats, fresh and frozen	7.09	297880.34		Australia
Total red meats, fresh and frozen	7.09	297880.34		Mexico
Total red meats, fresh and frozen	7.09	297880.34		New Zealand
Total red meats, fresh and frozen	7.09	297880.34		Brazil
Total red meats, fresh and frozen	7.09	297880.34		Uruguay
Total red meats, fresh and frozen	7.09	297880.34		Nicaragua
Frozen red meats and parts	8.72	241698.81		Australia
Frozen red meats and parts	8.72	241698.81		New Zealand
Frozen red meats and parts	8.72	241698.81		Brazil
Frozen red meats and parts	8.72	241698.81		Canada
Frozen red meats and parts	8.72	241698.81		Mexico
Frozen red meats and parts	8.72	241698.81		Uruguay

At the bottom of the table, there's a footer bar that says 'Rows per page: 20' and '1 - 15 of 15'.

This SQL query retrieves products that have either a high carbon footprint (greater than 5 kg CO₂ per kg) or high water usage (greater than 100 liters per kg). It achieves this using a UNION operator, which merges the results of two SELECT statements while eliminating duplicates. Each SELECT statement joins three tables: productData, environmentalCost, and locationData. The productData table contains product names and links to environmental and location data via foreign keys (EC_Id and LocationId). The environmentalCost table provides carbon footprint and water usage metrics, while the locationData table contains geographic information. The LIMIT 15 clause ensures that only 15 results are returned. The query structure allows for identifying environmentally impactful products efficiently, but performance optimizations were explored through indexing.

INDEXING:

Initial performance: -> Limit: 15 row(s) (cost=1782..1782 rows=15) (actual time=1.59..1.59 rows=15 loops=1) -> Table scan on <union temporary> (cost=1782..1793 rows=700) (actual time=1.59..1.59 rows=15 loops=1) -> Union materialize with deduplication (cost=1782..1782 rows=700) (actual

time=1.58..1.58 rows=15 loops=1) -> Limit table size: 15 unique row(s) -> Nested loop inner join (cost=856 rows=350) (actual time=1.01..1.45 rows=15 loops=1) -> Nested loop inner join (cost=474 rows=350) (actual time=0.854..0.977 rows=15 loops=1) -> Filter: ((p.EC_Id is not null) and (p.LocationId is not null)) (cost=107 rows=1050) (actual time=0.815..0.864 rows=29 loops=1) -> Table scan on p (cost=107 rows=1050) (actual time=0.807..0.826 rows=29 loops=1) -> Filter: (e.CarbonFootprint_per_kg > 5.00) (cost=0.25 rows=0.333) (actual time=0.00344..0.00351 rows=0.517 loops=29) -> Single-row index lookup on e using PRIMARY (EC_Id=p.EC_Id) (cost=0.25 rows=1) (actual time=0.00236..0.0024 rows=1 loops=29) -> Single-row index lookup on l using PRIMARY (LocationId=p.LocationId) (cost=0.991 rows=1) (actual time=0.0311..0.0311 rows=1 loops=15) -> Limit table size: 15 unique row(s) -> Nested loop inner join (cost=856 rows=350) (never executed) -> Nested loop inner join (cost=474 rows=350) (never executed) -> Filter: ((p.EC_Id is not null) and (p.LocationId is not null)) (cost=107 rows=1050) (never executed) -> Table scan on p (cost=107 rows=1050) (never executed) -> Filter: (e.WaterUse_per_kg > 100.00) (cost=0.25 rows=0.333) (never executed) -> Single-row index lookup on e using PRIMARY (EC_Id=p.EC_Id) (cost=0.25 rows=1) (never executed) -> Single-row index lookup on l using PRIMARY (LocationId=p.LocationId) (cost=0.991 rows=1) (never executed)

The first attempt at optimization involved indexing EC_Id and LocationId in the productData table, which serve as foreign keys linking productData to environmentalCost and locationData, respectively. The goal was to speed up joins by allowing the database to quickly locate matching rows in productData when retrieving related environmental and location data. However, after applying these indexes, we observed no significant improvement in query performance. This is likely because the database was already using the primary key indexes of environmentalCost and locationData for join operations, making additional indexing on productData redundant in this case.

```
CREATE INDEX idx_productdata_ecid
ON productData (EC_Id);
```

```
CREATE INDEX idx_productdata_locationid
ON productData (LocationId);
```

Result: -> Limit: 15 row(s) (cost=1782..1782 rows=15) (actual time=0.327..0.33 rows=15 loops=1) -> Table scan on <union temporary> (cost=1782..1793 rows=700) (actual time=0.325..0.328 rows=15 loops=1) -> Union materialize with deduplication (cost=1782..1782 rows=700) (actual time=0.323..0.323 rows=15 loops=1) -> Limit table size: 15 unique row(s) -> Nested loop inner join (cost=856 rows=350) (actual time=0.196..0.279 rows=15 loops=1) -> Nested loop inner join (cost=474 rows=350) (actual time=0.127..0.175 rows=15 loops=1) -> Filter: ((p.EC_Id is not null) and (p.LocationId is not null)) (cost=107 rows=1050) (actual time=0.104..0.115 rows=29 loops=1) -> Table scan on p (cost=107 rows=1050) (actual time=0.102..0.109 rows=29 loops=1) -> Filter: (e.CarbonFootprint_per_kg > 5.00) (cost=0.25 rows=0.333) (actual time=0.00176..0.00181 rows=0.517 loops=29) -> Single-row index lookup on e using PRIMARY (EC_Id=p.EC_Id) (cost=0.25 rows=1) (actual time=0.00129..0.00132 rows=1

loops=29) -> Single-row index lookup on l using PRIMARY (LocationId=p.LocationId) (cost=0.991 rows=1) (actual time=0.00667..0.00672 rows=1 loops=15) -> Limit table size: 15 unique row(s) -> Nested loop inner join (cost=856 rows=350) (never executed) -> Nested loop inner join (cost=474 rows=350) (never executed) -> Filter: ((p.EC_Id is not null) and (p.LocationId is not null)) (cost=107 rows=1050) (never executed) -> Table scan on p (cost=107 rows=1050) (never executed) -> Filter: (e.WaterUse_per_kg > 100.00) (cost=0.25 rows=0.333) (never executed) -> Single-row index lookup on e using PRIMARY (EC_Id=p.EC_Id) (cost=0.25 rows=1) (never executed) -> Single-row index lookup on l using PRIMARY (LocationId=p.LocationId) (cost=0.991 rows=1) (never executed)

There is no improvement in costs between the indexed and non-indexed versions of the query. We believe this is because we were already utilizing primary key indexes for the joins, ensuring that the foreign key columns were accessed properly without needing any extra indexing. Since these foreign key columns are efficiently accessed through the structure of the primary key indexes, adding explicit indexes on them does not really enhance the performance of the query. As a result, the overall execution time remains basically unchanged despite the additional indexing.

We also tried adding indexes on the non-primary key columns used in the WHERE clause:

```
CREATE INDEX idx_environmentalcost_carbonfootprint
ON environmentalCost (CarbonFootprint_per_kg);
```

-> Limit: 15 row(s) (cost=2848..2848 rows=15) (actual time=0.262..0.266 rows=15 loops=1) -> Table scan on <union temporary> (cost=2848..2870 rows=1595) (actual time=0.261..0.263 rows=15 loops=1) -> Union materialize with deduplication (cost=2848..2848 rows=1595) (actual time=0.259..0.259 rows=15 loops=1) -> Limit table size: 15 unique row(s) -> Nested loop inner join (cost=1070 rows=547) (actual time=0.116..0.222 rows=15 loops=1) -> Nested loop inner join (cost=474 rows=547) (actual time=0.0555..0.0959 rows=15 loops=1) -> Filter: ((p.EC_Id is not null) and (p.LocationId is not null)) (cost=107 rows=1050) (actual time=0.0414..0.0505 rows=29 loops=1) -> Table scan on p (cost=107 rows=1050) (actual time=0.0397..0.0453 rows=29 loops=1) -> Filter: (e.CarbonFootprint_per_kg > 5.00) (cost=0.25 rows=0.52) (actual time=0.00133..0.00138 rows=0.517 loops=29) -> Single-row index lookup on e using PRIMARY (EC_Id=p.EC_Id) (cost=0.25 rows=1) (actual time=0.00101..0.00105 rows=1 loops=29) -> Single-row index lookup on l using PRIMARY (LocationId=p.LocationId) (cost=0.991 rows=1) (actual time=0.0081..0.00813 rows=1 loops=15) -> Limit table size: 15 unique row(s) -> Nested loop inner join (cost=1618 rows=1049) (never executed) -> Nested loop inner join (cost=474 rows=1049) (never executed) -> Filter: ((p.EC_Id is not null) and (p.LocationId is not null)) (cost=107 rows=1050) (never executed) -> Table scan on p (cost=107 rows=1050) (never executed) -> Filter: (e.WaterUse_per_kg > 100.00) (cost=0.25 rows=0.999) (never executed) -> Single-row index lookup on e using PRIMARY

(EC_Id=p.EC_Id) (cost=0.25 rows=1) (never executed) -> Single-row index lookup on l using PRIMARY
(LocationId=p.LocationId) (cost=0.99 rows=1) (never executed)

CREATE INDEX idx_environmentalcost_wateruse

ON environmentalCost (WaterUse_per_kg);

-> Limit: 15 row(s) (cost=2848..2848 rows=15) (actual time=0.262..0.266 rows=15 loops=1) -> Table scan on <union temporary> (cost=2848..2870 rows=1595) (actual time=0.261..0.263 rows=15 loops=1) -> Union materialize with deduplication (cost=2848..2848 rows=1595) (actual time=0.259..0.259 rows=15 loops=1) -> Limit table size: 15 unique row(s) -> Nested loop inner join (cost=1070 rows=547) (actual time=0.116..0.222 rows=15 loops=1) -> Nested loop inner join (cost=474 rows=547) (actual time=0.0555..0.0959 rows=15 loops=1) -> Filter: ((p.EC_Id is not null) and (p.LocationId is not null)) (cost=107 rows=1050) (actual time=0.0414..0.0505 rows=29 loops=1) -> Table scan on p (cost=107 rows=1050) (actual time=0.0397..0.0453 rows=29 loops=1) -> Filter: (e.CarbonFootprint_per_kg > 5.00) (cost=0.25 rows=0.52) (actual time=0.00133..0.00138 rows=0.517 loops=29) -> Single-row index lookup on e using PRIMARY (EC_Id=p.EC_Id) (cost=0.25 rows=1) (actual time=0.00101..0.00105 rows=1 loops=29) -> Single-row index lookup on l using PRIMARY (LocationId=p.LocationId) (cost=0.991 rows=1) (actual time=0.0081..0.00813 rows=1 loops=15) -> Limit table size: 15 unique row(s) -> Nested loop inner join (cost=1618 rows=1049) (never executed) -> Nested loop inner join (cost=474 rows=1049) (never executed) -> Filter: ((p.EC_Id is not null) and (p.LocationId is not null)) (cost=107 rows=1050) (never executed) -> Table scan on p (cost=107 rows=1050) (never executed) -> Filter: (e.WaterUse_per_kg > 100.00) (cost=0.25 rows=0.999) (never executed) -> Single-row index lookup on e using PRIMARY (EC_Id=p.EC_Id) (cost=0.25 rows=1) (never executed) -> Single-row index lookup on l using PRIMARY (LocationId=p.LocationId) (cost=0.99 rows=1) (never executed)

This actually increased the total cost estimate from 1782 to 2848, but since, after the index lookup, we still see “Filter: (e.CarbonFootprint_per_kg > 5.00)”, it seems like the index is not directly finding matching rows. Overall, indexing did not improve this query’s performance, and we believe this is because the database was already using primary keys for joining tables, and the filtering was all applied after joins, which meant that indexing on filter columns did not improve the cost.

2) Finding the highest total carbon footprint product in each user's grocery list

CONCEPTS USED:

1. Multiple JOINS (productData, groceryProduct, environmentalCost, locationData, userData)
2. Subqueries

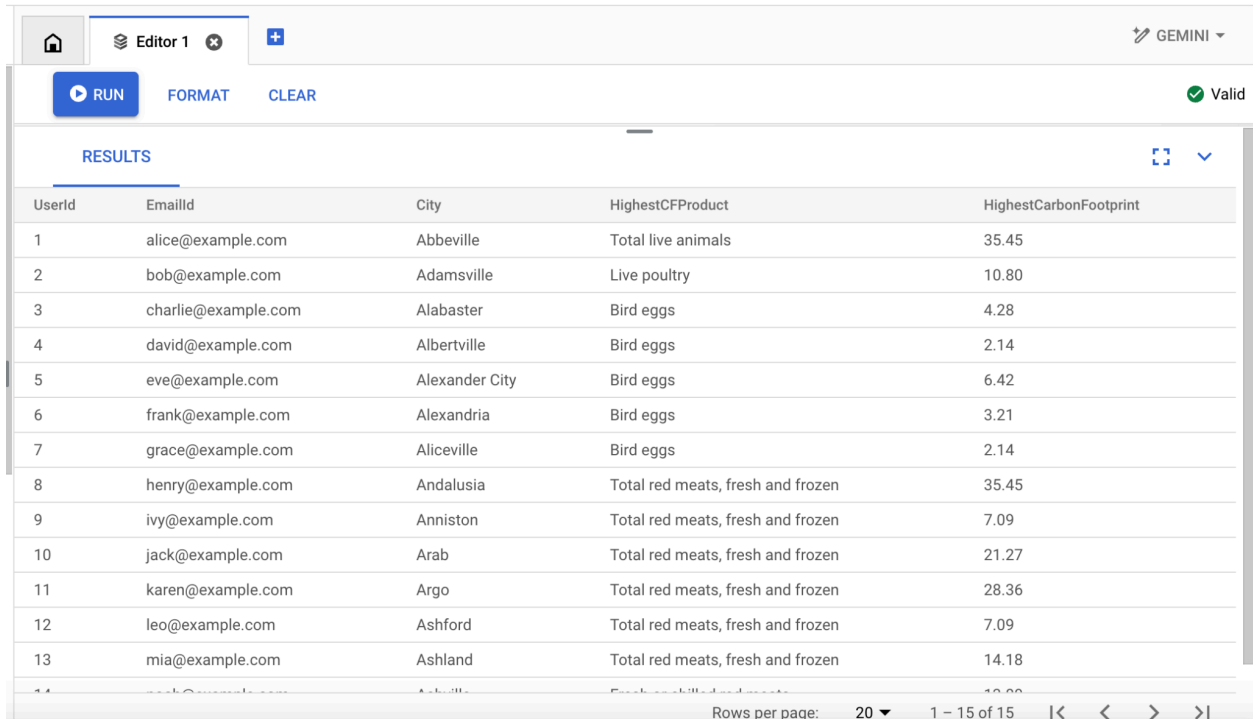
QUERY:

```
SELECT
  u.UserId,
  u.EmailId,
  l.City,
  (
    SELECT p.ProductName
    FROM groceryProduct AS gp2
    JOIN productData AS p
      ON gp2.ProductId = p.ProductId
    JOIN environmentalCost AS ec
      ON p.EC_Id = ec.EC_Id
    WHERE gp2.UserId = u.UserId
    ORDER BY (ec.CarbonFootprint_per_kg * gp2.Quantity) DESC
    LIMIT 1
  ) AS HighestCFProduct,
  (
    SELECT MAX(ec2.CarbonFootprint_per_kg * gp2.Quantity)
    FROM groceryProduct AS gp2
    JOIN productData AS p2
      ON gp2.ProductId = p2.ProductId
    JOIN environmentalCost AS ec2
      ON p2.EC_Id = ec2.EC_Id
    WHERE gp2.UserId = u.UserId
  ) AS HighestCarbonFootprint
FROM userData AS u
```

```

JOIN locationData AS l
  ON u.UserLocationId = l.LocationId
LIMIT 15;
PROOF:

```



UserId	EmailId	City	HighestCFProduct	HighestCarbonFootprint
1	alice@example.com	Abbeville	Total live animals	35.45
2	bob@example.com	Adamsville	Live poultry	10.80
3	charlie@example.com	Alabaster	Bird eggs	4.28
4	david@example.com	Albertville	Bird eggs	2.14
5	eve@example.com	Alexander City	Bird eggs	6.42
6	frank@example.com	Alexandria	Bird eggs	3.21
7	grace@example.com	Aliceville	Bird eggs	2.14
8	henry@example.com	Andalusia	Total red meats, fresh and frozen	35.45
9	ivy@example.com	Anniston	Total red meats, fresh and frozen	7.09
10	jack@example.com	Arab	Total red meats, fresh and frozen	21.27
11	karen@example.com	Argo	Total red meats, fresh and frozen	28.36
12	leo@example.com	Ashford	Total red meats, fresh and frozen	7.09
13	mia@example.com	Ashland	Total red meats, fresh and frozen	14.18
14	nash@example.com	Ashville	Freekeh/Medium meats	10.00
15	olivia@example.com	Atlanta	Freekeh/Medium meats	10.00

This query finds the product with the highest total carbon footprint in each user's grocery list by leveraging multiple JOIN operations and subqueries. The main query retrieves user information (UserId, EmailId) from the userData table and joins it with locationData to include the user's city. The first subquery, within the SELECT statement, determines the highest carbon footprint product for each user by joining groceryProduct with productData and environmentalCost, then ordering products by their total carbon footprint (CarbonFootprint_per_kg * Quantity) in descending order and selecting the top product. The second subquery calculates the actual highest carbon footprint value for each user by applying MAX() to the same computed metric. The LIMIT 15 clause restricts the results to 15 users, making the query more efficient by reducing the dataset size.

INDEXING:

Initial performance: -> Limit: 15 row(s) (cost=27.6 rows=15) (actual time=0.234..0.361 rows=15 loops=1) -> Nested loop inner join (cost=27.6 rows=23) (actual time=0.233..0.358 rows=15 loops=1) -> Filter: (u.UserLocationId is not null) (cost=2.55 rows=23) (actual time=0.206..0.216 rows=15 loops=1) -> Table scan on u (cost=2.55 rows=23) (actual time=0.179..0.187 rows=15 loops=1) -> Single-row index lookup on l using PRIMARY (LocationId=u.UserLocationId) (cost=0.995 rows=1) (actual time=0.00739..0.00743 rows=1 loops=15) -> Select #2 (subquery in projection; dependent) -> Limit: 1 row(s) (actual time=0.0332..0.0333 rows=1 loops=15) -> Sort: `(ec.CarbonFootprint_per_kg * gp2.Quantity)` DESC, limit input to 1 row(s) per chunk (actual time=0.0329..0.0329 rows=1 loops=15) -> Stream results (cost=1.19 rows=1.17) (actual time=0.0204..0.0276 rows=1.27 loops=15) -> Nested loop inner join (cost=1.19 rows=1.17) (actual time=0.0133..0.0201 rows=1.27 loops=15) -> Nested loop inner join (cost=0.778 rows=1.17) (actual time=0.00987..0.0162 rows=1.27 loops=15) -> Index lookup on gp2 using PRIMARY (UserId=u.UserId) (cost=0.367 rows=1.17) (actual time=0.00471..0.00611 rows=1.27 loops=15) -> Filter: (p.EC_Id is not null) (cost=0.335 rows=1) (actual time=0.00568..0.0058 rows=1 loops=19) -> Single-row index lookup on p using PRIMARY (ProductId=gp2.ProductId) (cost=0.335 rows=1) (actual time=0.00513..0.00517 rows=1 loops=19) -> Single-row index lookup on ec using PRIMARY (EC_Id=p.EC_Id) (cost=0.335 rows=1) (actual time=0.00254..0.00258 rows=1 loops=19) -> Select #3 (subquery in projection; dependent) -> Aggregate: max((ec2.CarbonFootprint_per_kg * gp2.Quantity)) (cost=1.31 rows=1) (actual time=0.0191..0.0192 rows=1 loops=15) -> Nested loop inner join (cost=1.19 rows=1.17) (actual time=0.0131..0.0152 rows=1.27 loops=15) -> Nested loop inner join (cost=0.778 rows=1.17) (actual time=0.00882..0.0106 rows=1.27 loops=15) -> Index lookup on gp2 using PRIMARY (UserId=u.UserId) (cost=0.367 rows=1.17) (actual time=0.00559..0.00652 rows=1.27 loops=15) -> Filter: (p2.EC_Id is not null) (cost=0.335 rows=1) (actual time=0.00271..0.00282 rows=1 loops=19) -> Single-row index lookup on p2 using PRIMARY (ProductId=gp2.ProductId) (cost=0.335 rows=1) (actual time=0.00249..0.00252 rows=1 loops=19) -> Single-row index lookup on ec2 using PRIMARY (EC_Id=p2.EC_Id) (cost=0.335 rows=1) (actual time=0.00321..0.00324 rows=1 loops=19)

First we can try indexing on the main query's join, in order to speed up joining users to their location.

[CREATE INDEX idx_userdata_location ON userData\(UserLocationId\);](#)

Results: -> Limit: 15 row(s) (cost=27.6 rows=15) (actual time=0.234..0.361 rows=15 loops=1) -> Nested loop inner join (cost=27.6 rows=23) (actual time=0.233..0.358 rows=15 loops=1) -> Filter: (u.UserLocationId is not null) (cost=2.55 rows=23) (actual time=0.206..0.216 rows=15 loops=1) -> Table scan on u (cost=2.55 rows=23) (actual time=0.179..0.187 rows=15 loops=1) -> Single-row index lookup on l using PRIMARY (LocationId=u.UserLocationId) (cost=0.995 rows=1) (actual time=0.00739..0.00743 rows=1 loops=15) -> Select #2 (subquery in projection; dependent) -> Limit: 1 row(s) (actual time=0.0332..0.0333 rows=1 loops=15) -> Sort: `(ec.CarbonFootprint_per_kg * gp2.Quantity)` DESC, limit input to 1 row(s) per chunk (actual time=0.0329..0.0329 rows=1 loops=15) -> Stream results (cost=1.19 rows=1.17) (actual time=0.0204..0.0276 rows=1.27 loops=15) -> Nested loop inner join (cost=1.19 rows=1.17) (actual time=0.0133..0.0201 rows=1.27 loops=15) -> Nested loop inner join (cost=0.778 rows=1.17) (actual time=0.00987..0.0162 rows=1.27 loops=15) -> Index lookup on gp2 using PRIMARY (UserId=u.UserId) (cost=0.367 rows=1.17) (actual time=0.00471..0.00611 rows=1.27 loops=15) -> Filter: (p.EC_Id is not null) (cost=0.335 rows=1) (actual time=0.00568..0.0058 rows=1 loops=19) -> Single-row

index lookup on p using PRIMARY (ProductId=gp2.ProductId) (cost=0.335 rows=1) (actual time=0.00513..0.00517 rows=1 loops=19) -> Single-row index lookup on ec using PRIMARY (EC_Id=p.EC_Id) (cost=0.335 rows=1) (actual time=0.00254..0.00258 rows=1 loops=19) -> Select #3 (subquery in projection; dependent) -> Aggregate: max((ec2.CarbonFootprint_per_kg * gp2.Quantity)) (cost=1.31 rows=1) (actual time=0.0191..0.0192 rows=1 loops=15) -> Nested loop inner join (cost=1.19 rows=1.17) (actual time=0.0131..0.0152 rows=1.27 loops=15) -> Nested loop inner join (cost=0.778 rows=1.17) (actual time=0.00882..0.0106 rows=1.27 loops=15) -> Index lookup on gp2 using PRIMARY (UserId=u.UserId) (cost=0.367 rows=1.17) (actual time=0.00559..0.00652 rows=1.27 loops=15) -> Filter: (p2.EC_Id is not null) (cost=0.335 rows=1) (actual time=0.00271..0.00282 rows=1 loops=19) -> Single-row index lookup on p2 using PRIMARY (ProductId=gp2.ProductId) (cost=0.335 rows=1) (actual time=0.00249..0.00252 rows=1 loops=19) -> Single-row index lookup on ec2 using PRIMARY (EC_Id=p2.EC_Id) (cost=0.335 rows=1) (actual time=0.00321..0.00324 rows=1 loops=19)

Both plans have identical cost estimates for the main operations (Limit: 15 row(s) (cost=27.6 rows=15)). The results indicate that adding an index on UserLocationId did not improve performance, as the execution plan remains identical to the initial query. The table scan on userData still persists, suggesting that the optimizer did not utilize the index. This could be because the database engine finds a full table scan to be just as efficient due to the small number of rows in userData, or it may already have an efficient strategy using the primary key on locationData. Since the total query cost and execution time remain the same, we should explore optimizing the first subquery instead.

Let's instead try to make indexes for the first subquery now:

`CREATE INDEX idx_groceryproduct_user_product ON groceryProduct(UserId, ProductId);`

Results: -> Limit: 15 row(s) (cost=27.6 rows=15) (actual time=0.137..0.175 rows=15 loops=1) -> Nested loop inner join (cost=27.6 rows=23) (actual time=0.135..0.172 rows=15 loops=1) -> Filter: (u.UserLocationId is not null) (cost=2.55 rows=23) (actual time=0.111..0.117 rows=15 loops=1) -> Table scan on u (cost=2.55 rows=23) (actual time=0.109..0.113 rows=15 loops=1) -> Single-row index lookup on l using PRIMARY (LocationId=u.UserLocationId) (cost=0.995 rows=1) (actual time=0.00324..0.00327 rows=1 loops=15) -> Select #2 (subquery in projection; dependent) -> Limit: 1 row(s) (actual time=0.0152..0.0153 rows=1 loops=15) -> Sort: `(ec.CarbonFootprint_per_kg * gp2.Quantity)` DESC, limit input to 1 row(s) per chunk (actual time=0.015..0.015 rows=1 loops=15) -> Stream results (cost=1.19 rows=1.17) (actual time=0.0106..0.013 rows=1.27 loops=15) -> Nested loop inner join (cost=1.19 rows=1.17) (actual time=0.00887..0.0109 rows=1.27 loops=15) -> Nested loop inner join (cost=0.778 rows=1.17) (actual time=0.00644..0.00819 rows=1.27 loops=15) -> Index lookup on gp2 using PRIMARY (UserId=u.UserId) (cost=0.367 rows=1.17) (actual time=0.00317..0.00397 rows=1.27 loops=15) -> Filter: (p.EC_Id is not null) (cost=0.335 rows=1) (actual time=0.00285..0.00295 rows=1 loops=19) -> Single-row

index lookup on p using PRIMARY (ProductId=gp2.ProductId) (cost=0.335 rows=1) (actual time=0.00264..0.00268 rows=1 loops=19) -> Single-row index lookup on ec using PRIMARY (EC_Id=p.EC_Id) (cost=0.335 rows=1) (actual time=0.00174..0.00178 rows=1 loops=19) -> Select #3 (subquery in projection; dependent) -> Aggregate: max((ec2.CarbonFootprint_per_kg * gp2.Quantity)) (cost=1.31 rows=1) (actual time=0.00872..0.00877 rows=1 loops=15) -> Nested loop inner join (cost=1.19 rows=1.17) (actual time=0.00578..0.00734 rows=1.27 loops=15) -> Nested loop inner join (cost=0.778 rows=1.17) (actual time=0.00409..0.00538 rows=1.27 loops=15) -> Index lookup on gp2 using PRIMARY (UserId=u.UserId) (cost=0.367 rows=1.17) (actual time=0.00191..0.00252 rows=1.27 loops=15) -> Filter: (p2.EC_Id is not null) (cost=0.335 rows=1) (actual time=0.00183..0.00192 rows=1 loops=19) -> Single-row index lookup on p2 using PRIMARY (ProductId=gp2.ProductId) (cost=0.335 rows=1) (actual time=0.00165..0.00168 rows=1 loops=19) -> Single-row index lookup on ec2 using PRIMARY (EC_Id=p2.EC_Id) (cost=0.335 rows=1) (actual time=0.00114..0.00118 rows=1 loops=19)

CREATE INDEX idx_productdata_ecid ON productData(EC_Id);
 CREATE INDEX idx_environmentalcost_carbon ON environmentalCost(EC_Id,
 CarbonFootprint_per_kg);

Results: -> Limit: 15 row(s) (cost=27.6 rows=15) (actual time=0.137..0.175 rows=15 loops=1) -> Nested loop inner join (cost=27.6 rows=23) (actual time=0.135..0.172 rows=15 loops=1) -> Filter: (u.UserLocationId is not null) (cost=2.55 rows=23) (actual time=0.111..0.117 rows=15 loops=1) -> Table scan on u (cost=2.55 rows=23) (actual time=0.109..0.113 rows=15 loops=1) -> Single-row index lookup on l using PRIMARY (LocationId=u.UserLocationId) (cost=0.995 rows=1) (actual time=0.00324..0.00327 rows=1 loops=15) -> Select #2 (subquery in projection; dependent) -> Limit: 1 row(s) (actual time=0.0152..0.0153 rows=1 loops=15) -> Sort: `(ec.CarbonFootprint_per_kg * gp2.Quantity)` DESC, limit input to 1 row(s) per chunk (actual time=0.015..0.015 rows=1 loops=15) -> Stream results (cost=1.19 rows=1.17) (actual time=0.0106..0.013 rows=1.27 loops=15) -> Nested loop inner join (cost=1.19 rows=1.17) (actual time=0.00887..0.0109 rows=1.27 loops=15) -> Nested loop inner join (cost=0.778 rows=1.17) (actual time=0.00644..0.00819 rows=1.27 loops=15) -> Index lookup on gp2 using PRIMARY (UserId=u.UserId) (cost=0.367 rows=1.17) (actual time=0.00317..0.00397 rows=1.27 loops=15) -> Filter: (p.EC_Id is not null) (cost=0.335 rows=1) (actual time=0.00285..0.00295 rows=1 loops=19) -> Single-row index lookup on p using PRIMARY (ProductId=gp2.ProductId) (cost=0.335 rows=1) (actual time=0.00264..0.00268 rows=1 loops=19) -> Single-row index lookup on ec using PRIMARY (EC_Id=p.EC_Id) (cost=0.335 rows=1) (actual time=0.00174..0.00178 rows=1 loops=19) -> Select #3 (subquery in projection; dependent) -> Aggregate: max((ec2.CarbonFootprint_per_kg * gp2.Quantity)) (cost=1.31 rows=1) (actual time=0.00872..0.00877 rows=1 loops=15) -> Nested loop inner join (cost=1.19 rows=1.17) (actual time=0.00578..0.00734 rows=1.27 loops=15) -> Nested loop inner join (cost=0.778 rows=1.17) (actual time=0.00409..0.00538 rows=1.27 loops=15) -> Index lookup on gp2 using PRIMARY (UserId=u.UserId) (cost=0.367 rows=1.17) (actual time=0.00191..0.00252 rows=1.27 loops=15) -> Filter: (p2.EC_Id is not null) (cost=0.335 rows=1) (actual time=0.00183..0.00192 rows=1 loops=19) -> Single-row index lookup on p2 using PRIMARY (ProductId=gp2.ProductId) (cost=0.335 rows=1) (actual time=0.00165..0.00168 rows=1 loops=19) -> Single-row index lookup on ec2 using PRIMARY (EC_Id=p2.EC_Id) (cost=0.335 rows=1) (actual time=0.00114..0.00118 rows=1 loops=19)

This also does not improve cost, since the joins already use primary key columns, and by default the database has the best possible indexes for primary keys so adding secondary indexes doesn't provide a benefit. The performance costs of the query likely stem mostly from the subquery, as we can see that "WHERE gp2.UserId = u.UserId" makes this query run for each of the users.

3) Finding the locations with the highest average carbon footprint

CONCEPTS USED:

1. Multiple JOINS (locationData, productData, environmentalCost)
2. GROUP BY (average CF per location)
3. Subquery in the HAVING clause (comparing each location's average CF to the overall average)

QUERY:

```
SELECT
    l.LocationId,
    l.Country,
    AVG(ec.CarbonFootprint_per_kg) AS AvgCarbonFootprint
FROM locationData AS l
JOIN productData AS p
    ON l.LocationId = p.LocationId
JOIN environmentalCost AS ec
    ON p.EC_Id = ec.EC_Id
GROUP BY l.LocationId, l.City, l.Country
HAVING AVG(ec.CarbonFootprint_per_kg) > (
    SELECT AVG(ec2.CarbonFootprint_per_kg)
    FROM environmentalCost AS ec2
)
ORDER BY AvgCarbonFootprint DESC
LIMIT 15;
```

PROOF:

RESULTS



LocationId	Country	AvgCarbonFootprint
19853	Bulgaria	9.930000
19984	Norway	9.757778
19892	Faroe Islands	9.750000
20034	Sweden	8.650000
19886	El Salvador	8.600000
20056	Uruguay	8.520000
19879	Denmark	8.170000
20026	South Korea	7.440000
19991	Paraguay	7.300000
19925	Ireland	7.020000
19910	Guatemala	6.766324
19871	Costa Rica	6.710000
19977	Nicaragua	6.558929
19863	Chile	6.545833
20045	Tunisia	6.470000

Rows per page: 20 1 – 15 of 15 |< < > >|

INDEXING:

Initial performance: -> Limit: 15 row(s) (actual time=7.92..7.92 rows=15 loops=1) -> Sort: AvgCarbonFootprint DESC (actual time=7.92..7.92 rows=15 loops=1) -> Filter: (??? > (select #2)) (actual time=7.76..7.86 rows=36 loops=1) -> Table scan on <temporary> (actual time=7.26..7.28 rows=60 loops=1) -> Aggregate using temporary table (actual time=7.26..7.26 rows=60 loops=1) -> Nested loop inner join (cost=1619 rows=1050) (actual time=0.498..5.28 rows=1050 loops=1) -> Nested loop inner join (cost=474 rows=1050) (actual time=0.427..3.31 rows=1050 loops=1) -> Filter: ((p.EC_Id is not null) and (p.LocationId is not null)) (cost=107 rows=1050) (actual time=0.394..1.8 rows=1050 loops=1) -> Table scan on p (cost=107 rows=1050) (actual time=0.391..1.64 rows=1050 loops=1) -> Single-row index lookup on ec using PRIMARY (EC_Id=p.EC_Id) (cost=0.25 rows=1) (actual time=0.00123..0.00126 rows=1 loops=1050) -> Single-row index lookup on l using PRIMARY (LocationId=p.LocationId) (cost=0.99 rows=1) (actual time=0.00165..0.00168 rows=1 loops=1050) -> Select #2 (subquery in condition; run only once) -> Aggregate: avg(ec2.CarbonFootprint_per_kg) (cost=202 rows=1) (actual time=0.39..0.39 rows=1 loops=1) -> Covering index scan on ec2 using idx_environmentalcost_carbonfootprint (cost=102 rows=1001) (actual time=0.084..0.281 rows=1000 loops=1)

We can first try to index the main query joins.

CREATE INDEX idx_productdata_locationid ON productData(LocationId);

-> Limit: 15 row(s) (actual time=5.48..5.48 rows=15 loops=1) -> Sort: AvgCarbonFootprint DESC (actual time=5.48..5.48 rows=15 loops=1) -> Filter: (??? > (select #2)) (actual time=5.39..5.43 rows=36 loops=1) -> Table scan on <temporary> (actual time=5.03..5.04 rows=60 loops=1) -> Aggregate using temporary table (actual time=5.03..5.03 rows=60 loops=1) -> Nested loop inner join (cost=1619 rows=1050) (actual

time=0.225..3.23 rows=1050 loops=1) -> Nested loop inner join (cost=474 rows=1050) (actual time=0.123..1.72 rows=1050 loops=1) -> Filter: ((p.EC_Id is not null) and (p.LocationId is not null)) (cost=107 rows=1050) (actual time=0.0759..0.659 rows=1050 loops=1) -> Table scan on p (cost=107 rows=1050) (actual time=0.0732..0.515 rows=1050 loops=1) -> Single-row index lookup on ec using PRIMARY (EC_Id=p.EC_Id) (cost=0.25 rows=1) (actual time=802e-6..830e-6 rows=1 loops=1050) -> Single-row index lookup on l using PRIMARY (LocationId=p.LocationId) (cost=0.99 rows=1) (actual time=0.00123..0.00126 rows=1 loops=1050) -> Select #2 (subquery in condition; run only once) -> Aggregate: avg(ec2.CarbonFootprint_per_kg) (cost=202 rows=1) (actual time=0.33..0.33 rows=1 loops=1) -> Covering index scan on ec2 using idx_environmentalcost_carbonfootprint (cost=102 rows=1001) (actual time=0.0478..0.221 rows=1000 loops=1)

CREATE INDEX idx_productdata_ecid ON productData(EC_Id);

-> Limit: 15 row(s) (actual time=5.48..5.48 rows=15 loops=1) -> Sort: AvgCarbonFootprint DESC (actual time=5.48..5.48 rows=15 loops=1) -> Filter: (??? > (select #2)) (actual time=5.39..5.43 rows=36 loops=1) -> Table scan on <temporary> (actual time=5.03..5.04 rows=60 loops=1) -> Aggregate using temporary table (actual time=5.03..5.03 rows=60 loops=1) -> Nested loop inner join (cost=1619 rows=1050) (actual time=0.225..3.23 rows=1050 loops=1) -> Nested loop inner join (cost=474 rows=1050) (actual time=0.123..1.72 rows=1050 loops=1) -> Filter: ((p.EC_Id is not null) and (p.LocationId is not null)) (cost=107 rows=1050) (actual time=0.0759..0.659 rows=1050 loops=1) -> Table scan on p (cost=107 rows=1050) (actual time=0.0732..0.515 rows=1050 loops=1) -> Single-row index lookup on ec using PRIMARY (EC_Id=p.EC_Id) (cost=0.25 rows=1) (actual time=802e-6..830e-6 rows=1 loops=1050) -> Single-row index lookup on l using PRIMARY (LocationId=p.LocationId) (cost=0.99 rows=1) (actual time=0.00123..0.00126 rows=1 loops=1050) -> Select #2 (subquery in condition; run only once) -> Aggregate: avg(ec2.CarbonFootprint_per_kg) (cost=202 rows=1) (actual time=0.33..0.33 rows=1 loops=1) -> Covering index scan on ec2 using idx_environmentalcost_carbonfootprint (cost=102 rows=1001) (actual time=0.0478..0.221 rows=1000 loops=1)

We can also try to index for the subquery:

CREATE INDEX idx_environmentalcost_carbon_avg ON environmentalCost(CarbonFootprint_per_kg);

This results in:

-> Limit: 15 row(s) (actual time=4.92..4.92 rows=15 loops=1) -> Sort: AvgCarbonFootprint DESC (actual time=4.92..4.92 rows=15 loops=1) -> Filter: (??? > (select #2)) (actual time=4.85..4.88 rows=36 loops=1) -> Table scan on <temporary> (actual time=4.42..4.43 rows=60 loops=1) -> Aggregate using temporary table (actual time=4.42..4.42 rows=60 loops=1) -> Nested loop inner join (cost=1619 rows=1050) (actual time=0.0977..2.69 rows=1050 loops=1) -> Nested loop inner join (cost=474 rows=1050) (actual time=0.0743..1.36 rows=1050 loops=1) -> Filter: ((p.EC_Id is not null) and (p.LocationId is not null)) (cost=107 rows=1050) (actual time=0.0588..0.55 rows=1050 loops=1) -> Table scan on p (cost=107 rows=1050) (actual time=0.057..0.43 rows=1050 loops=1) -> Single-row index lookup on ec using PRIMARY (EC_Id=p.EC_Id) (cost=0.25 rows=1) (actual time=535e-6..564e-6 rows=1 loops=1050) -> Single-row index lookup on l using PRIMARY (LocationId=p.LocationId) (cost=0.99 rows=1) (actual

time=0.00103..0.00106 rows=1 loops=1050) -> Select #2 (subquery in condition; run only once) -> Aggregate: avg(ec2.CarbonFootprint_per_kg) (cost=202 rows=1) (actual time=0.352..0.352 rows=1 loops=1) -> Covering index scan on ec2 using idx_environmentalcost_carbonfootprint (cost=102 rows=1001) (actual time=0.053..0.241 rows=1000 loops=1)

This has an identical cost. Through indexing we did not improve the cost of this query, and we believe this is because the structure of the query gets rid of the performance benefit of indexing. Since we have the having clause which filters after the aggregation(the average), we need to utilize all rows. Since we need to use all of the rows in the sorting and aggregation, our performance doesn't improve over all in this specific query.

- 4) Finding users who have an average carbon footprint lower than the average across all users

CONCEPTS:

1. JOINS (userData, groceryProduct, productData, environmentalCost)
2. GROUP BY (to find user averages)
3. Subquery in HAVING (comparing against the global average for all users/products)

QUERY:

```
SELECT
    u.UserId,
    u.FirstName,
    u.LastName,
    AVG(ec.CarbonFootprint_per_kg) AS UserAvgCarbonFootprint
FROM userData AS u
JOIN groceryProduct AS gp
    ON u.UserId = gp.UserId
JOIN productData AS p
    ON gp.ProductId = p.ProductId
JOIN environmentalCost AS ec
    ON p.EC_Id = ec.EC_Id
GROUP BY u.UserId, u.FirstName, u.LastName
HAVING
    AVG(ec.CarbonFootprint_per_kg) < (
        SELECT AVG(ec2.CarbonFootprint_per_kg)
        FROM userData AS u2
        JOIN groceryProduct AS gp2
            ON u2.UserId = gp2.UserId
        JOIN productData AS p2
            ON gp2.ProductId = p2.ProductId
        JOIN environmentalCost AS ec2
            ON p2.EC_Id = ec2.EC_Id
    )
```

ORDER BY UserAvgCarbonFootprint ASC;

PROOF:

RESULTS			
UserId	FirstName	LastName	UserAvgCarbonFootprint
3	Charlie	Black	1.070000
4	David	White	1.070000
5	Eve	Gray	1.070000
6	Frank	Blue	1.070000
7	Grace	Red	1.070000
18	Rachel	Olive	1.070000
19	Steve	Cyan	1.070000
20	Tina	Magenta	1.070000
21	Uma	Lavender	1.070000
22	Victor	Indigo	1.070000
23	Wendy	Coral	1.070000
2	Bob	Brown	3.600000
17	Quinn	Maroon	3.600000

Rows per page: 20 1 – 13 of 13

INDEXING:

Initial performance: -> Sort: UserAvgCarbonFootprint (actual time=2.96..2.97 rows=13 loops=1) -> Filter: (??? < (select #2)) (actual time=2.91..2.93 rows=13 loops=1) -> Table scan on <temporary> (actual time=2.66..2.67 rows=23 loops=1) -> Aggregate using temporary table (actual time=2.66..2.66 rows=23 loops=1) -> Nested loop inner join (cost=29.9 rows=27) (actual time=2.3..2.52 rows=27 loops=1) -> Nested loop inner join (cost=20.5 rows=27) (actual time=0.0956..0.29 rows=27 loops=1) -> Nested loop inner join (cost=11 rows=27) (actual time=0.0807..0.186 rows=27 loops=1) -> Table scan on u (cost=2.55 rows=23) (actual time=0.0576..0.0783 rows=23 loops=1) -> Covering index lookup on gp using idx_groceryproduct_userid (UserId=u.UserId) (cost=0.255 rows=1.17) (actual time=0.00313..0.00425 rows=1.17 loops=23) -> Filter: (p.EC_Id is not null) (cost=0.254 rows=1) (actual time=0.0034..0.00354 rows=1 loops=27) -> Single-row index lookup on p using PRIMARY (ProductId=gp.ProductId) (cost=0.254 rows=1) (actual time=0.00317..0.00322 rows=1 loops=27) -> Single-row index lookup on ec using PRIMARY (EC_Id=p.EC_Id) (cost=0.254 rows=1) (actual time=0.0823..0.0823 rows=1 loops=27) -> Select #2 (subquery in condition; run only once) -> Aggregate: avg(ec2.CarbonFootprint_per_kg) (cost=32.6 rows=1) (actual time=0.201..0.201 rows=1 loops=1) -> Nested loop inner join (cost=29.9 rows=27) (actual time=0.0492..0.188 rows=27 loops=1) -> Nested loop inner join (cost=20.5 rows=27) (actual time=0.041..0.158 rows=27 loops=1) -> Nested loop inner join (cost=11 rows=27) (actual time=0.0314..0.0941 rows=27 loops=1) -> Covering index scan on u2 using idx_userdata_location (cost=2.55 rows=23) (actual time=0.0217..0.0264 rows=23 loops=1) -> Covering index lookup on gp2 using idx_groceryproduct_userid (UserId=u2.UserId) (cost=0.255 rows=1.17) (actual time=0.00194..0.00264 rows=1.17 loops=23) -> Filter: (p2.EC_Id is not null) (cost=0.254 rows=1) (actual time=0.00204..0.00215 rows=1 loops=27) -> Single-row index lookup on p2 using PRIMARY (ProductId=gp2.ProductId) (cost=0.254 rows=1) (actual time=0.00184..0.00188 rows=1 loops=27) ->

Single-row index lookup on ec2 using PRIMARY (EC_Id=p2.EC_Id) (cost=0.254 rows=1) (actual time=802e-6..840e-6 rows=1 loops=27)

First we can try indexing on the joins in the query.

`CREATE INDEX idx_groceryproduct_user ON groceryProduct(UserId);`

This results in the same performance:

-> Sort: UserAvgCarbonFootprint (actual time=0.483..0.484 rows=13 loops=1) -> Filter: (??? < (select #2)) (actual time=0.455..0.465 rows=13 loops=1) -> Table scan on <temporary> (actual time=0.296..0.3 rows=23 loops=1) -> Aggregate using temporary table (actual time=0.296..0.296 rows=23 loops=1) -> Nested loop inner join (cost=29.9 rows=27) (actual time=0.0795..0.21 rows=27 loops=1) -> Nested loop inner join (cost=20.5 rows=27) (actual time=0.0724..0.183 rows=27 loops=1) -> Nested loop inner join (cost=11 rows=27) (actual time=0.0615..0.12 rows=27 loops=1) -> Table scan on u (cost=2.55 rows=23) (actual time=0.0452..0.0493 rows=23 loops=1) -> Covering index lookup on gp using idx_groceryproduct_userid (UserId=u.UserId) (cost=0.255 rows=1.17) (actual time=0.00222..0.00283 rows=1.17 loops=23) -> Filter: (p.EC_Id is not null) (cost=0.254 rows=1) (actual time=0.00206..0.00215 rows=1 loops=27) -> Single-row index lookup on p using PRIMARY (ProductId=gp.ProductId) (cost=0.254 rows=1) (actual time=0.00189..0.00192 rows=1 loops=27) -> Single-row index lookup on ec using PRIMARY (EC_Id=p.EC_Id) (cost=0.254 rows=1) (actual time=784e-6..814e-6 rows=1 loops=27) -> Select #2 (subquery in condition; run only once) -> Aggregate: avg(ec2.CarbonFootprint_per_kg) (cost=32.6 rows=1) (actual time=0.124..0.124 rows=1 loops=1) -> Nested loop inner join (cost=29.9 rows=27) (actual time=0.0218..0.111 rows=27 loops=1) -> Nested loop inner join (cost=20.5 rows=27) (actual time=0.0187..0.0933 rows=27 loops=1) -> Nested loop inner join (cost=11 rows=27) (actual time=0.0143..0.0533 rows=27 loops=1) -> Covering index scan on u2 using idx_userdata_location (cost=2.55 rows=23) (actual time=0.00995..0.0131 rows=23 loops=1) -> Covering index lookup on gp2 using idx_groceryproduct_userid (UserId=u2.UserId) (cost=0.255 rows=1.17) (actual time=0.00109..0.00154 rows=1.17 loops=23) -> Filter: (p2.EC_Id is not null) (cost=0.254 rows=1) (actual time=0.00123..0.00132 rows=1 loops=27) -> Single-row index lookup on p2 using PRIMARY (ProductId=gp2.ProductId) (cost=0.254 rows=1) (actual time=0.00109..0.00112 rows=1 loops=27) -> Single-row index lookup on ec2 using PRIMARY (EC_Id=p2.EC_Id) (cost=0.254 rows=1) (actual time=447e-6..475e-6 rows=1 loops=27)

`CREATE INDEX idx_groceryproduct_product ON groceryProduct(ProductId);`
`CREATE INDEX idx_productdata_ecid ON productData(EC_Id);`

This also results in the same performance:

-> Sort: UserAvgCarbonFootprint (actual time=0.483..0.484 rows=13 loops=1) -> Filter: (??? < (select #2)) (actual time=0.455..0.465 rows=13 loops=1) -> Table scan on <temporary> (actual time=0.296..0.3 rows=23 loops=1) -> Aggregate using temporary table (actual time=0.296..0.296 rows=23 loops=1) -> Nested loop inner join (cost=29.9 rows=27) (actual time=0.0795..0.21 rows=27 loops=1) -> Nested loop inner join (cost=20.5 rows=27) (actual time=0.0724..0.183 rows=27 loops=1) -> Nested loop inner join (cost=11 rows=27) (actual time=0.0615..0.12 rows=27 loops=1) -> Table scan on u (cost=2.55 rows=23)

(actual time=0.0452..0.0493 rows=23 loops=1) -> Covering index lookup on gp using idx_groceryproduct_userid (UserId=u.UserId) (cost=0.255 rows=1.17) (actual time=0.00222..0.00283 rows=1.17 loops=23) -> Filter: (p.EC_Id is not null) (cost=0.254 rows=1) (actual time=0.00206..0.00215 rows=1 loops=27) -> Single-row index lookup on p using PRIMARY (ProductId=gp.ProductId) (cost=0.254 rows=1) (actual time=0.00189..0.00192 rows=1 loops=27) -> Single-row index lookup on ec using PRIMARY (EC_Id=p.EC_Id) (cost=0.254 rows=1) (actual time=784e-6..814e-6 rows=1 loops=27) -> Select #2 (subquery in condition; run only once) -> Aggregate: avg(ec2.CarbonFootprint_per_kg) (cost=32.6 rows=1) (actual time=0.124..0.124 rows=1 loops=1) -> Nested loop inner join (cost=29.9 rows=27) (actual time=0.0218..0.111 rows=27 loops=1) -> Nested loop inner join (cost=20.5 rows=27) (actual time=0.0187..0.0933 rows=27 loops=1) -> Nested loop inner join (cost=11 rows=27) (actual time=0.0143..0.0533 rows=27 loops=1) -> Covering index scan on u2 using idx_userdata_location (cost=2.55 rows=23) (actual time=0.00995..0.0131 rows=23 loops=1) -> Covering index lookup on gp2 using idx_groceryproduct_userid (UserId=u2.UserId) (cost=0.255 rows=1.17) (actual time=0.00109..0.00154 rows=1.17 loops=23) -> Filter: (p2.EC_Id is not null) (cost=0.254 rows=1) (actual time=0.00123..0.00132 rows=1 loops=27) -> Single-row index lookup on p2 using PRIMARY (ProductId=gp2.ProductId) (cost=0.254 rows=1) (actual time=0.00109..0.00112 rows=1 loops=27) -> Single-row index lookup on ec2 using PRIMARY (EC_Id=p2.EC_Id) (cost=0.254 rows=1) (actual time=447e-6..475e-6 rows=1 loops=27)

We can also try indexing on the joins in the subquery:

[CREATE INDEX idx_environmentalcost_carbon ON environmentalCost\(CarbonFootprint_per_kg\);](#)

-> Sort: UserAvgCarbonFootprint (actual time=0.817..0.82 rows=13 loops=1) -> Filter: (??? < (select #2)) (actual time=0.766..0.786 rows=13 loops=1) -> Table scan on <temporary> (actual time=0.537..0.545 rows=23 loops=1) -> Aggregate using temporary table (actual time=0.535..0.535 rows=23 loops=1) -> Nested loop inner join (cost=29.9 rows=27) (actual time=0.152..0.351 rows=27 loops=1) -> Nested loop inner join (cost=20.5 rows=27) (actual time=0.105..0.273 rows=27 loops=1) -> Nested loop inner join (cost=11 rows=27) (actual time=0.0893..0.18 rows=27 loops=1) -> Table scan on u (cost=2.55 rows=23) (actual time=0.0584..0.0666 rows=23 loops=1) -> Covering index lookup on gp using idx_groceryproduct_userid (UserId=u.UserId) (cost=0.255 rows=1.17) (actual time=0.00351..0.00457 rows=1.17 loops=23) -> Filter: (p.EC_Id is not null) (cost=0.254 rows=1) (actual time=0.00298..0.00311 rows=1 loops=27) -> Single-row index lookup on p using PRIMARY (ProductId=gp.ProductId) (cost=0.254 rows=1) (actual time=0.00274..0.00278 rows=1 loops=27) -> Single-row index lookup on ec using PRIMARY (EC_Id=p.EC_Id) (cost=0.254 rows=1) (actual time=0.00254..0.00258 rows=1 loops=27) -> Select #2 (subquery in condition; run only once) -> Aggregate: avg(ec2.CarbonFootprint_per_kg) (cost=32.6 rows=1) (actual time=0.197..0.197 rows=1 loops=1) -> Nested loop inner join (cost=29.9 rows=27) (actual time=0.0461..0.184 rows=27 loops=1) -> Nested loop inner join (cost=20.5 rows=27) (actual time=0.0375..0.153 rows=27 loops=1) -> Nested loop inner join (cost=11 rows=27) (actual time=0.0283..0.0912 rows=27 loops=1) -> Covering index scan on u2 using idx_userdata_location (cost=2.55 rows=23) (actual time=0.0146..0.0193 rows=23 loops=1) -> Covering index lookup on gp2 using idx_groceryproduct_userid (UserId=u2.UserId) (cost=0.255 rows=1.17) (actual time=0.00202..0.00269 rows=1.17 loops=23) -> Filter: (p2.EC_Id is not null) (cost=0.254 rows=1) (actual time=0.00197..0.00208 rows=1 loops=27) -> Single-row index lookup on p2 using PRIMARY (ProductId=gp2.ProductId) (cost=0.254 rows=1) (actual

time=0.00178..0.00182 rows=1 loops=27) -> Single-row index lookup on ec2 using PRIMARY (EC_Id=p2.EC_Id)
(cost=0.254 rows=1) (actual time=881e-6..917e-6 rows=1 loops=27)

This also did not change the cost. Overall, this indexing didn't reduce the estimated query cost because of the query structure. The group by clause requires us to read all the relevant rows, and the order by requires the results to be sorted fully. Because we need to continuously do this work across the whole table, indexing won't reduce our performance cost significantly.