

GCP Connection

Database: Olist_data

```
mysql> show databases;
+-----+
| Database |
+-----+
| information_schema |
| mysql |
| olist_data |
| performance_schema |
| sys |
+-----+
5 rows in set (0.03 sec)
```

TABLES

```
mysql>
mysql> show tables
-> ;
+-----+
| Tables_in_olist_data |
+-----+
| olist_customers_dataset |
| olist_geolocation_dataset |
| olist_order_items_dataset |
| olist_order_payments_dataset |
| olist_order_reviews_dataset |
| olist_orders_dataset |
| olist_products_dataset |
| olist_sellers_dataset |
| product_category_name_translation |
+-----+
9 rows in set (0.09 sec)
```

DDL Command for the table

Table olist_customers_dataset

```
CREATE TABLE customers (  
  customer_id VARCHAR(255) NOT NULL,  
  customer_zip_code_prefix INT,  
  customer_city VARCHAR,  
  customer_state VARCHAR  
  PRIMARY KEY (customer_id)  
);
```

CREATE TABLE geolocation (

```
  geolocation_zip_code_prefix INT NOT NULL,  
  geolocation_lat DECIMAL(9,6),  
  geolocation_lng DECIMAL(9,6),  
  geolocation_city VARCHAR(100),  
  geolocation_state VARCHAR(2)  
);
```

CREATE TABLE olist_orders_dataset (

```
  order_id VARCHAR(255) NOT NULL PRIMARY KEY,  
  customer_id VARCHAR(255),  
  order_status VARCHAR(50),  
  order_purchase_timestamp TIMESTAMP,  
  order_approved_at TIMESTAMP,  
  order_delivered_carrier_date TIMESTAMP,  
  order_delivered_customer_date TIMESTAMP,  
  order_estimated_delivery_date TIMESTAMP  
);
```

Table olist_order_items_dataset

```
CREATE TABLE order_items (  
  order_id VARCHAR(255) NOT NULL,  
  order_item_id INT,  
  product_id VARCHAR(255),  
  seller_id VARCHAR(255),  
  shipping_limit_date DATETIME,  
  price FLOAT,  
  freight_value FLOAT,
```

PRIMARY KEY (order_id, order_item_id)

);

Table order_payments_dataset

```
CREATE TABLE order_payments (  
  order_id VARCHAR(255) NOT NULL,  
  payment_sequential INT NOT NULL,  
  payment_type VARCHAR,  
  payment_installments INT,  
  payment_value DECIMAL,  
  PRIMARY KEY (order_id, payment_sequential)
```

);

Table olist_order_reviews_dataset

```
CREATE TABLE order_reviews (  
  review_id VARCHAR(255) NOT NULL,  
  order_id VARCHAR(255) NOT NULL,  
  review_score INT,  
  review_comment_title VARCHAR,  
  review_comment_message VARCHAR,  
  review_creation_date TIMESTAMP,  
  review_answer_timestamp TIMESTAMP  
  PRIMARY KEY (review_id)
```

);

Table olist_products_dataset

```
CREATE TABLE products (  
  product_id VARCHAR(255) NOT NULL,  
  product_category_name VARCHAR(255),  
  seller_id VARCHAR(255),  
  product_name_length INT,  
  product_description_length INT,  
  product_photos_qty INT,  
  product_weight_g INT,  
  product_length_cm INT,  
  product_height_cm INT,  
  product_width_cm INT,  
  PRIMARY KEY (product_id)
```

);

Table olist_sellers_dataset

```
CREATE TABLE sellers (  
  seller_id VARCHAR(255) NOT NULL,  
  seller_city VARCHAR(255),  
  seller_state VARCHAR(255),
```

```
seller_state VARCHAR,  
PRIMARY KEY (seller_id)  
);
```

Table product_category_name_translation

```
CREATE TABLE product_category_name_translation (  
  product_category_name VARCHAR(255) NOT NULL,  
  product_category_name_english VARCHAR(255),  
  PRIMARY KEY (product_category_name)  
);
```

ROWS COUNT

table_name	row_count
olist_customers_dataset	99442
olist_geolocation_dataset	1000164
olist_order_items_dataset	102426
olist_order_payments_dataset	112650
olist_order_reviews_dataset	99224
olist_orders_dataset	99442
olist_products_dataset	32329
olist_sellers_dataset	3096
product_category_name_translation	72

9 rows in set (4.21 sec)

ADVANCED Query:

Query 1 – Average Freight and Price by Seller (Join + Group By)

```
SELECT
    s.seller_id,
    s.seller_city,
    COUNT(oi.order_id) AS total_orders,
    AVG(oi.price) AS avg_price,
    AVG(oi.freight_value) AS avg_freight
FROM olist_order_items_dataset oi
JOIN olist_products_dataset p ON oi.product_id = p.product_id
JOIN olist_sellers_dataset s ON p.seller_id = s.seller_id
GROUP BY s.seller_id, s.seller_city;
Limit15;
```

seller_id	seller_city	total_orders	avg_price	avg_freight
0015a82c2db000af6aaaf3ae2ecb0532	santo andre	3	895	21.020000457763672
001cca7ae9ae17fb1caed9dfb1094831	cariacica	240	105.1626248995463	36.99858363866806
001e6ad469a905060d959994f1b41e4f	sao goncalo	1	250	17.940000534057617
002100f778ceb8431b7a1020ff7ab48f	franca	55	22.445454198663885	14.43018214485862
003554e2dce176b5555353e4f3555ac8	goiania	1	120	19.3799991607666
004c9cd9d87a3c30c522c48c4fc07416	ibitinga	170	115.95711768655217	20.889588170893052
00720abe85ba0859807595bbf045a33b	guarulhos	26	38.750000073359566	12.153076777091393
00ab3eff1b5192e5f1a63bcecfee11c8	sao paulo	1	98	12.079999923706055
00d8b143d12632bad99c0ad66ad52825	belo horizonte	1	86	51.099998474121094
00ee68308b45bc5e2660cd833c3f81cc	sao paulo	319	110.4404068471496	17.52097165734043
00fc707aaaad2d31347cf883cd2dfe10	maringa	848	84.6035264483038	15.271910347325623
010543a62bd80aa422851e79a3bc7540	sao paulo	2	708	15.97499942779541
010da0602d7774602cd1b3f5fb7b709e	sao bernardo do campo	5	169.89999389648438	46.04999923706055
011b0eaba87386a2ae96a7d32bb531d1	pompeia	2	49.9900016784668	14.59000015258789
01266d4c46afa519678d16a8b683d325	curitiba	3	30.083333651224773	15.743333498636881

15 rows in set (2.14 sec)

Query 2: Top States by Average Review Score (JOINS, GROUP BY, ORDER BY, and aggregation) WORKING

```
SELECT
  c.customer_state,
  COUNT(r.review_id) AS total_reviews,
  ROUND(AVG(r.review_score), 2) AS avg_review_score
FROM olist_order_reviews_dataset r
JOIN olist_orders_dataset o ON r.order_id = o.order_id
JOIN olist_customers_dataset c ON o.customer_id = c.customer_id
GROUP BY c.customer_state
ORDER BY avg_review_score DESC
LIMIT 15;
```

customer_state	total_reviews	avg_review_score
AP	67	4.19
AM	147	4.18
PR	5038	4.18
SP	41689	4.17
MG	11625	4.14
RS	5483	4.13
MS	724	4.12
RN	482	4.11
TO	279	4.10
MT	903	4.10
SC	3623	4.07
DF	2148	4.06
AC	81	4.05
RO	252	4.05
ES	2016	4.04

15 rows in set (5.68 sec)

Query 3: Sellers with Low Freight Costs (Subquery)

This query finds sellers whose average freight value is greater than \$50, using a subquery in the WHERE clause:

```
SELECT
  s.seller_id,
  s.seller_city,
  s.seller_state
FROM olist_sellers_dataset s
WHERE s.seller_id IN (
  SELECT p.seller_id
  FROM olist_order_items_dataset oi
  JOIN olist_products_dataset p ON oi.product_id = p.product_id
  GROUP BY p.seller_id
  HAVING AVG(oi.freight_value) < 10
)
LIMIT 15;
```

```
-> LIMIT 15;
```

seller_id	seller_city	seller_state
3442f8959a84dea7ee197c632cb2df15	campinas	SP
e9e446d01bd10a97a8ffcf4a3a20cb2	sao paulo	SP
ec2e006556300a79a5a91e4876ab3a56	sao paulo	SP
05ca864204d09595ae591b93ea9cf93d	barueri	SP
0692610d8abe24f287e9fae90ea0bbee	sao paulo	SP
ea00f977a203ff88adf7057cb7806998	santos	SP
ffcfefa19b08742c5d315f2791395ee5	curitiba	PR
c1dde11f12d05c478f5de2d7319ad3b2	sao paulo	SP
bee36b4f9a2b9fdcaff6ec05df202ed0	sao paulo	SP
baf15155e37ef5492731459bdc05be8a	sao paulo	SP
71593c7413973a1e160057b80d4958f6	sao paulo / sao paulo	SP
f12d3c2a14729ae461b920c11fe20fdc	santo andre	SP
f0ec6a2adb05c62655a26dd347b8dede	sao paulo	SP
cc1f04647be106ba74e62b21f358af25	sao paulo	SP

15 rows in set (1.65 sec)

Query 4: Compare Products in High vs Low Freight Brackets (SET OPERATOR: UNION)

```
(
  SELECT
    product_id,
    price,
    freight_value,
    'High Freight' AS freight_bracket
  FROM olist_order_items_dataset
  WHERE freight_value > 100
)
UNION
(
  SELECT
    product_id,
    price,
    freight_value,
    'Low Freight' AS freight_bracket
  FROM olist_order_items_dataset
  WHERE freight_value < 5
)
```

```

+-----+-----+-----+-----+
| product_id | price | freight_value | freight_bracket |
+-----+-----+-----+-----+
| 43cc8e4d981bc04b9d78b12e8a908d41 | 1240 | 102.63 | High Freight |
| a233df9a388d27dbdfd31731d4236db0 | 2649.99 | 134.17 | High Freight |
| 8d4e92265a16e69a1e1d76e67e46d72f | 1350 | 294.76 | High Freight |
| 63c4a70e0a12b4bd9475fca9e9937e76 | 122.99 | 164.98 | High Freight |
| e303dfa61ada1f0823b4775f192606b3 | 148.5 | 165.32 | High Freight |
| 660422061e06da17ca6101e9d6b7aae8 | 649 | 169.12 | High Freight |
| ab495f166205a883ffe5ab0b5b55f867 | 629.9 | 117.35 | High Freight |
| 90c1b4e040d1d1c45897ec2dad4a809d | 839.99 | 174.49 | High Freight |
| ef854c7d98d5eba672287b0a9d37075b | 1990 | 125.05 | High Freight |
| 72d0a38fe43ba7087d71e245d1b76c9e | 229 | 106.31 | High Freight |
| 5640c59a8f6a08b3758272590693eec3 | 238.47 | 119.56 | High Freight |
| 67a7c7243c0585ccc44471b5a5c115e9 | 322 | 186.38 | High Freight |
| 65841ad29fc48cd40902e03da7511e05 | 849 | 174.95 | High Freight |
| 1945afae0c93166dce1f186d00125695 | 517.99 | 107.87 | High Freight |
| 8a443635fdf9759915c9be5be2e3b862 | 99.9 | 112.44 | High Freight |
+-----+-----+-----+-----+
15 rows in set (0.19 sec)
```


Query 5 : Products That Were Reviewed the Most with Highest Average Ratings (Multiple Joins, GROUP BY, HAVING with aggregation)

```
SELECT
    p.product_id,
    pct.product_category_name_english AS category,
    COUNT(r.review_id) AS total_reviews,
    ROUND(AVG(r.review_score), 2) AS avg_score
FROM olist_order_reviews_dataset r
JOIN olist_orders_dataset o ON r.order_id = o.order_id
JOIN olist_order_items_dataset oi ON o.order_id = oi.order_id
JOIN olist_products_dataset p ON oi.product_id = p.product_id
JOIN product_category_name_translation pct
    ON p.product_category_name = pct.product_category_name
GROUP BY p.product_id, pct.product_category_name_english
HAVING total_reviews > 5 AND avg_score >= 4.5
ORDER BY total_reviews DESC, avg_score DESC
```

```
***** 1. row *****
product_id: 629e019a
product_name: auto
total_reviews: 73
avg_score: 4.56
***** 2. row *****
product_id: f7a17d2c
product_name: housewares
total_reviews: 64
avg_score: 4.55
***** 3. row *****
product_id: 5f504b3a
product_name: cool_stuff
total_reviews: 63
avg_score: 4.56
***** 4. row *****
product_id: 6a8631b7
product_name: electronics
total_reviews: 62
avg_score: 4.71
***** 5. row *****
product_id: fbcl488c
product_name: perfumery
total_reviews: 58
avg_score: 4.62
***** 6. row *****
product_id: d696750e
product_name: housewares
total_reviews: 57
avg_score: 4.53
***** 7. row *****
product_id: 73326828
product_name: food
total_reviews: 54
avg_score: 4.83
***** 8. row *****
product_id: 130482ad
product_name: computers_accessories
total_reviews: 52
avg_score: 4.54
***** 9. row *****
product_id: ed2067a9
product_name: food
total_reviews: 52
avg_score: 4.54
***** 10. row *****
product_id: aa280035
product_name: toys
total_reviews: 46
avg_score: 4.63
***** 11. row *****
product_id: d5b703c2
product_name: toys
total_reviews: 44
avg_score: 4.50
***** 12. row *****
product_id: 57e089e3
product_name: baby
total_reviews: 43
avg_score: 4.63
***** 13. row *****
product_id: f8b624d4
product_name: housewares
total_reviews: 42
avg_score: 4.69
***** 14. row *****
product_id: aadff084
product_name: health_beauty
total_reviews: 40
avg_score: 4.65
***** 15. row *****
product_id: bdb4be6c
product_name: housewares
total_reviews: 40
avg_score: 4.55
15 rows in set (5.23 sec)
```

Part 2 Indexing:

QUERY 1

```
| -> Limit: 15 row(s) (actual time=3424..3424 rows=15 loops=1)
|   -> Table scan on <temporary> (actual time=3424..3424 rows=15 loops=1)
|     -> Aggregate using temporary table (actual time=3424..3424 rows=2996 loops=1)
|       -> Nested loop inner join (cost=222711 rows=111760) (actual time=150..2959 rows=111024 loops=1)
|         -> Nested loop inner join (cost=99775 rows=111760) (actual time=115..2569 rows=111024 loops=1)
|           -> Table scan on oi (cost=11702 rows=111760) (actual time=68.1..1246 rows=112651 loops=1)
|             -> Filter: (p.seller_id is not null) (cost=0.688 rows=1) (actual time=0.0115..0.0116 rows=0.986 loops=112651)
|               -> Single-row index lookup on p using PRIMARY (product_id=oi.product_id) (cost=0.688 rows=1) (actual time=0.0113..0.0113 rows=0.986 loops=112651)
|               -> Single-row index lookup on s using PRIMARY (seller_id=p.seller_id) (cost=1 rows=1) (actual time=0.00328..0.00331 rows=1 loops=111024)
|
```

Index Choice: 2

We chose Index Choice 2 because it improved query performance by speeding up joins and average calculations. The indexes reduced full table scans and enabled faster data access, especially on large datasets.

1. CREATE INDEX idx_oi_price_freight ON olist_order_items_dataset(price, freight_value);

```
| -> Limit: 15 row(s) (actual time=3282..3282 rows=15 loops=1)
|   -> Table scan on <temporary> (actual time=3282..3282 rows=15 loops=1)
|     -> Aggregate using temporary table (actual time=3282..3282 rows=2996 loops=1)
|       -> Nested loop inner join (cost=257539 rows=111760) (actual time=212..2739 rows=111024 loops=1)
|         -> Nested loop inner join (cost=134609 rows=111760) (actual time=141..2173 rows=111024 loops=1)
|           -> Table scan on oi (cost=11914 rows=111760) (actual time=92.5..764 rows=112651 loops=1)
|             -> Filter: (p.seller_id is not null) (cost=0.998 rows=1) (actual time=0.0122..0.0123 rows=0.986 loops=112651)
|               -> Single-row index lookup on p using PRIMARY (product_id=oi.product_id) (cost=0.998 rows=1) (actual time=0.012..0.012 rows=0.986 loops=112651)
|               -> Single-row index lookup on s using PRIMARY (seller_id=p.seller_id) (cost=1 rows=1) (actual time=0.00485..0.00489 rows=1 loops=111024)
|
```

This index is designed to optimize the aggregation operations in the query by allowing the database to quickly access the 'price' and 'freight_value' columns used in the AVG functions. By having these columns indexed together, the system can reduce full table scans and speed up the computation of averages, especially when working with a large dataset.

2. CREATE INDEX idx_p_seller_id ON olist_products_dataset(seller_id); JOINED ATTRIBUTE

```
| -> Limit: 15 row(s) (actual time=2282..2282 rows=15 loops=1)
|   -> Table scan on <temporary> (actual time=2282..2282 rows=15 loops=1)
|     -> Aggregate using temporary table (actual time=2282..2282 rows=2996 loops=1)
|       -> Nested loop inner join (cost=178840 rows=111760) (actual time=170..1857 rows=111024 loops=1)
|         -> Nested loop inner join (cost=50904 rows=111760) (actual time=109..1392 rows=111024 loops=1)
|           -> Table scan on oi (cost=11788 rows=111760) (actual time=109..888 rows=112651 loops=1)
|             -> Filter: (p.seller_id is not null) (cost=0.25 rows=1) (actual time=0.00418..0.00427 rows=0.986 loops=112651)
|               -> Single-row index lookup on p using PRIMARY (product_id=oi.product_id) (cost=0.25 rows=1) (actual time=0.00399..0.00402 rows=0.986 loops=112651)
|               -> Single-row index lookup on s using PRIMARY (seller_id=p.seller_id) (cost=1 rows=1) (actual time=0.00394..0.00397 rows=1 loops=111024)
|
```

Index on olist_products_dataset(seller_id): This index improves the efficiency of the join between the products and sellers datasets. By indexing the seller_id in the products table, the database engine can rapidly match products to their corresponding sellers, reducing lookup time during the join process and ultimately enhancing the overall query performance.

3. CREATE INDEX idx_oi_product_id ON olist_order_items_dataset(product_id);

```
| -> Limit: 15 row(s) (actual time=3604..3504 rows=15 loops=1)
|   -> Table scan on <temporary> (actual time=3504..3504 rows=15 loops=1)
|     -> Aggregate using temporary table (actual time=3504..3504 rows=2996 loops=1)
|       -> Nested loop inner join (cost=119315 rows=115078) (actual time=75.6..3197 rows=111024 loops=1)
|         -> Nested loop inner join (cost=12608 rows=32631) (actual time=71..785 rows=32329 loops=1)
|           -> Table scan on s (cost=327 rows=3090) (actual time=16.1..285 rows=3096 loops=1)
|             -> Covering index lookup on p using idx_p_seller_id (seller_id=s.seller_id) (cost=2.91 rows=10.6) (actual time=0.109..0.161 rows=10.4 loops=3096)
|             -> Index lookup on oi using idx_oi_product_id (product_id=p.product_id) (cost=2.91 rows=3.52) (actual time=0.05..0.0741 rows=3.43 loops=32329)
|
```

Index on olist_order_items_dataset(product_id): This index is established to speed up the join between order items and products. With the product_id column indexed in the order items

dataset, the database can quickly locate matching records for each product, which minimizes the data scanned during the join operation and improves the execution speed of the query.

Query2: Before Indexing Performance:

```
| -> Limit: 15 row(s) (actual time=3530..3530 rows=15 loops=1)
|   -> Sort: avg_review_score DESC, limit input to 15 row(s) per chunk (actual time=3530..3530 rows=15 loops=1)
|     -> Table scan on <temporary> (actual time=3530..3530 rows=28 loops=1)
|       -> Aggregate using temporary table (actual time=3530..3530 rows=28 loops=1)
|         -> Nested loop inner join (cost=104e+6 rows=94e+6) (actual time=1109.3448 rows=99224 loops=1)
|           -> Filter: (o.order_id = r.order_id) (cost=94.4e+6 rows=94e+6) (actual time=1059..1892 rows=99224 loops=1)
|             -> Inner hash join (<hash>(o.order_id)=<hash>(r.order_id)) (cost=94.4e+6 rows=94e+6) (actual time=1059..1832 rows=99224 loops=1)
|               -> Filter: (o.customer_id is not null) (cost=4.89 rows=9815) (actual time=49.6..655 rows=99442 loops=1)
|                 -> Table scan on o (cost=4.89 rows=98150) (actual time=49.6..641 rows=99442 loops=1)
|                   -> Hash
|                     -> Table scan on r (cost=10043 rows=95722) (actual time=80.916 rows=99224 loops=1)
|                       -> Single-row index lookup on c using PRIMARY (customer_id=o.customer_id) (cost=10.4e-6 rows=1) (actual time=0.0154..0.0154 rows=1 loops=99224)
|
```

Index Choice: 3

We chose Index Choice 3 because it significantly improved join performance between reviews, orders, and customers. Indexing `order_id` in both the reviews and orders datasets sped up matching operations, while indexing `customer_state` allowed quicker filtering. These indexes reduced full table scans and improved overall query efficiency, especially on large datasets.

1. CREATE INDEX idx_reviews_order_id ON olist_order_reviews_dataset(order_id);

```
| -> Limit: 15 row(s) (actual time=12547..12547 rows=15 loops=1)
|   -> Sort: avg_review_score DESC, limit input to 15 row(s) per chunk (actual time=12547..12547 rows=15 loops=1)
|     -> Table scan on <temporary> (actual time=12547..12547 rows=28 loops=1)
|       -> Aggregate using temporary table (actual time=12547..12547 rows=28 loops=1)
|         -> Nested loop inner join (cost=167171 rows=98150) (actual time=132.12333 rows=99224 loops=1)
|           -> Filter: ((o.order_id is not null) and (o.customer_id is not null)) (cost=10437 rows=98150) (actual time=79.955 rows=99442 loops=1)
|             -> Table scan on o (cost=10437 rows=98150) (actual time=79.930 rows=99442 loops=1)
|               -> Index lookup on r using idx_reviews_order_id (order_id=o.order_id) (cost=0.397 rows=1) (actual time=0.0888..0.0821 rows=0.998 loops=99442)
|                 -> Single-row index lookup on c using PRIMARY (customer_id=o.customer_id) (cost=1 rows=1) (actual time=0.0318..0.0318 rows=1 loops=99224)
|
```

This index is designed to speed up the join between reviews and orders by allowing the database to quickly locate matching `order_id` values in the reviews table. This minimizes the need for full table scans during the join operation, significantly enhancing performance when working with large datasets.

2. CREATE INDEX idx_orders_order_id ON olist_orders_dataset(order_id);

```
| -> Limit: 15 row(s) (actual time=11232..11232 rows=15 loops=1)
|   -> Sort: avg_review_score DESC, limit input to 15 row(s) per chunk (actual time=11232..11232 rows=15 loops=1)
|     -> Table scan on <temporary> (actual time=11231..11231 rows=28 loops=1)
|       -> Aggregate using temporary table (actual time=11231..11231 rows=28 loops=1)
|         -> Nested loop inner join (cost=168077 rows=95722) (actual time=155.11047 rows=99224 loops=1)
|           -> Filter: (r.order_id is not null) (cost=10043 rows=95722) (actual time=80.8..913 rows=99224 loops=1)
|             -> Table scan on r (cost=10043 rows=95722) (actual time=80.8..900 rows=99224 loops=1)
|               -> Filter: (o.customer_id is not null) (cost=0.451 rows=1) (actual time=0.075..0.0759 rows=1 loops=99224)
|                 -> Index lookup on o using idx_orders_order_id (order_id=r.order_id) (cost=0.451 rows=1) (actual time=0.0748..0.0756 rows=1 loops=99224)
|                   -> Single-row index lookup on c using PRIMARY (customer_id=o.customer_id) (cost=1 rows=1) (actual time=0.0257..0.0257 rows=1 loops=99224)
|
```

The index on `order_id` in the orders dataset optimizes the join process by efficiently retrieving corresponding order records for each review. By using this index, the database can quickly match `order_id` values between the reviews and orders tables, reducing lookup time and improving overall query performance.

3. CREATE INDEX idx_customers_state ON olist_customers_dataset(customer_state);

```
| -> Limit: 15 row(s) (actual time=3287..3287 rows=15 loops=1)
      -> Sort: avg_review_score DESC, limit input to 15 row(s) per chunk (actual time=3287..3287 rows=15 loops=1)
      -> Table scan on <temporary> (actual time=3286..3286 rows=28 loops=1)
      -> Aggregate using temporary table (actual time=3286..3286 rows=28 loops=1)
      -> Nested loop inner join (cost=104e+6 rows=94e+6) (actual time=1156..3197 rows=99224 loops=1)
            -> Filter: (o.order_id = r.order_id) (cost=94.4e+6 rows=94e+6) (actual time=1156..2125 rows=99224 loops=1)
            -> Inner hash join (hash=>(o.order_id)<=>(r.order_id)) (cost=94.4e+6 rows=94e+6) (actual time=1156..2058 rows=99224 loops=1)
                  -> Filter: (o.customer_id is not null) (cost=4.89 rows=9815) (actual time=47.6..756 rows=99442 loops=1)
                  -> Table scan on o (cost=4.89 rows=98150) (actual time=47.6..740 rows=99442 loops=1)
            -> Hash
                  -> Table scan on r (cost=10043 rows=95722) (actual time=63.7..980 rows=99224 loops=1)
      -> Single-row index lookup on c using PRIMARY (customer_id=o.customer_id) (cost=10.4e-6 rows=1) (actual time=0.0105..0.0105 rows=1 loops=99224)
|
-----
```

Indexing customer_state in the customers dataset is intended to improve the performance of the grouping and aggregation operations in the query. Since the query groups results by customer_state and computes aggregate functions like count and average, this index helps the database quickly sort and group rows by state, leading to faster execution even if the column has relatively few unique values.

Query 3 – Sellers with Low Freight Costs (Subquery)

```
| -> Limit: 15 row(s) (cost=314 rows=15) (actual time=5490..5495 rows=15 loops=1)
      -> Filter: <in optimizer>(s.seller_id,s.seller_id in (select #2)) (cost=314 rows=3086) (actual time=5490..5495 rows=15 loops=1)
      -> Table scan on s (cost=314 rows=3086) (actual time=77.4..81.1 rows=279 loops=1)
      -> Select #2 (subquery in condition; run only once)
            -> Filter: ((s.seller_id = 'materialized subquery'.seller_id)) (cost=0..0 rows=0) (actual time=19.6..19.6 rows=0.0543 loops=276)
            -> Limit: 1 row(s) (cost=0..0 rows=0) (actual time=19.6..19.6 rows=0.0543 loops=276)
            -> Index lookup on <materialized subquery> using <auto distinct key> (seller_id=s.seller_id) (actual time=19.6..19.6 rows=0.0543 loops=276)
            -> Materialize with deduplication (cost=0..0 rows=0) (actual time=5412..5412 rows=118 loops=1)
            -> Filter: (avg(o.freight_value) < 10) (actual time=5410..5412 rows=118 loops=1)
            -> Table scan on <temporary> (actual time=5409..5410 rows=3061 loops=1)
            -> Index lookup on <temporary> (actual time=5409..5409 rows=3061 loops=1)
            -> Nested loop inner join (cost=54139 rows=119945) (actual time=189..5152 rows=112651 loops=1)
                  -> Filter: (p.product_id is not null) (cost=3258 rows=31860) (actual time=89.6..954 rows=32952 loops=1)
                  -> Table scan on p (cost=3258 rows=31860) (actual time=89.6..947 rows=32952 loops=1)
                  -> Index lookup on oi using idx_oi_product_id (product_id=p.product_id) (cost=1.22 rows=3.76) (actual time=0.0692..0.127 rows=3.42 loops=32952)
|
-----
```

INDEX Choice: 4

We chose index 4 because it improved the performance of the outer query by speeding up the filtering process. It allowed the database to quickly match seller_ids from the subquery, reducing lookup time and making the overall query more efficient.

1. CREATE INDEX idx_oi_product_id ON olist_order_items_dataset(product_id);

```
| -> Limit: 15 row(s) (cost=327 rows=15) (actual time=4907..4940 rows=15 loops=1)
      -> Filter: <in optimizer>(s.seller_id,s.seller_id in (select #2)) (cost=327 rows=3096) (actual time=4907..4940 rows=15 loops=1)
      -> Table scan on s (cost=327 rows=3096) (actual time=9.59..42.9 rows=520 loops=1)
      -> Select #2 (subquery in condition; run only once)
            -> Filter: ((s.seller_id = 'materialized subquery'.seller_id)) (cost=99941..99941 rows=1) (actual time=9.31..9.31 rows=0.0288 loops=521)
            -> Limit: 1 row(s) (cost=99941..99941 rows=1) (actual time=9.3..9.3 rows=0.0288 loops=521)
            -> Index lookup on <materialized subquery> using <auto distinct key> (seller_id=s.seller_id) (actual time=9.3..9.3 rows=0.0288 loops=521)
            -> Materialize with deduplication (cost=99941..99941 rows=3038) (actual time=4847..4847 rows=116 loops=1)
            -> Filter: (avg(oi.freight_value) < 10) (cost=99637 rows=3038) (actual time=3375..4846 rows=116 loops=1)
            -> Group aggregate: avg(oi.freight_value) (cost=99637 rows=3038) (actual time=171..4844 rows=2996 loops=1)
            -> Nested loop inner join (cost=8345 rows=112922) (actual time=108..4768 rows=111024 loops=1)
                  -> Covering index scan on p using idx_p_seller_id (cost=3559 rows=32069) (actual time=30.3..376 rows=32329 loops=1)
                  -> Index lookup on oi using idx_oi_product_id (product_id=p.product_id) (cost=2.29 rows=3.52) (actual time=0.0847..0.135 rows=3.43 loops=32329)
|
-----
```

2. olist_order_items_dataset(product_id, freight_value)

CREATE INDEX idx_oi_product_freight ON olist_order_items_dataset(product_id, freight_value);

```
| -> Limit: 15 row(s) (cost=327 rows=15) (actual time=867..901 rows=15 loops=1)
      -> Filter: <in optimizer>(s.seller_id,s.seller_id in (select #2)) (cost=327 rows=3096) (actual time=867..901 rows=15 loops=1)
      -> Table scan on s (cost=327 rows=3096) (actual time=9.59..42.9 rows=520 loops=1)
      -> Select #2 (subquery in condition; run only once)
            -> Filter: ((s.seller_id = 'materialized subquery'.seller_id)) (cost=78505..78505 rows=1) (actual time=1.65..1.65 rows=0.0288 loops=521)
            -> Limit: 1 row(s) (cost=78505..78505 rows=1) (actual time=1.65..1.65 rows=0.0288 loops=521)
            -> Index lookup on <materialized subquery> using <auto distinct key> (seller_id=s.seller_id) (actual time=1.65..1.65 rows=0.0288 loops=521)
            -> Materialize with deduplication (cost=78505..78505 rows=3038) (actual time=858..858 rows=116 loops=1)
            -> Filter: (avg(oi.freight_value) < 10) (cost=78201 rows=3038) (actual time=253..857 rows=116 loops=1)
            -> Group aggregate: avg(oi.freight_value) (cost=78201 rows=3038) (actual time=122..856 rows=2996 loops=1)
            -> Nested loop inner join (cost=66959 rows=112423) (actual time=121..790 rows=111024 loops=1)
                  -> Covering index scan on p using idx_p_seller_id (cost=3559 rows=32069) (actual time=120..302 rows=32329 loops=1)
                  -> Covering index lookup on oi using idx_oi_product_freight (product_id=p.product_id) (cost=1.63 rows=3.51) (actual time=0.0122..0.0147 rows=3.43 loops=32329)
|
-----
```

This composite index helps the subquery perform both the join on `product_id` and the aggregation on `freight_value` efficiently. It allows the database to quickly find records based on `product_id` while also retrieving `freight_value` for computing averages, reducing unnecessary scans. Optimizes join to products + improves AVG computation in HAVING.

3. `olist_products_dataset(product_id, seller_id)`

```
| -> Limit: 15 row(s) (cost=327 rows=15) (actual time=1556..1604 rows=15 loops=1)
|   -> Filter: (<in optimizer>(s.seller_id,s.seller_id in (select #2))) (cost=327 rows=3096) (actual time=1556..1604 rows=15 loops=1)
|     -> Table scan on s (cost=327 rows=3096) (actual time=1.19..47.9 rows=520 loops=1)
|     -> Select #2 (subquery in condition): run only once
|       -> Filter: ((s.seller_id = <materialized subquery>'.seller_id)) (cost=0..0 rows=0) (actual time=2.98..2.98 rows=0.0288 loops=521)
|         -> Limit: 1 row(s) (cost=0..0 rows=0) (actual time=2.98..2.98 rows=0.0288 loops=521)
|           -> Index lookup on <materialized subquery> using <auto distinct key> (seller_id=s.seller_id) (actual time=2.98..2.98 rows=0.0288 loops=521)
|             -> Materialize with deduplication (cost=0..0 rows=0) (actual time=1555..1555 rows=116 loops=1)
|               -> Filter: (avg(oi.freight_value) < 10) (actual time=1553..1554 rows=116 loops=1)
|                 -> Table scan on <temporary> (actual time=1552..1552 rows=2996 loops=1)
|                   -> Aggregate using temporary table (actual time=1552..1552 rows=2996 loops=1)
|                     -> Nested loop inner join (cost=58523 rows=111760) (actual time=165..1294 rows=111024 loops=1)
|                       -> Table scan on oi (cost=12236 rows=111760) (actual time=110..365 rows=112631 loops=1)
|                       -> Single-row index lookup on p using PRIMARY (product_id=oi.product_id) (cost=0.314 rows=1) (actual time=0.00604..0.00607 rows=0.986 loops=112631)
```

This index speeds up the join between order items and products by enabling fast lookup of `seller_id` using `product_id`. It makes grouping by `seller_id` in the subquery more efficient, which is crucial for filtering based on the average freight value. Helps with the join and allows efficient grouping by `seller_id`.

4. `olist_sellers_dataset(seller_id)`

```
| -> Limit: 15 row(s) (cost=326 rows=15) (actual time=1590..1592 rows=15 loops=1)
|   -> Filter: (<in optimizer>(s.seller_id,s.seller_id in (select #2))) (cost=326 rows=3096) (actual time=1590..1592 rows=15 loops=1)
|     -> Table scan on s (cost=326 rows=3096) (actual time=0.0453..0.698 rows=520 loops=1)
|     -> Select #2 (subquery in condition): run only once
|       -> Filter: ((s.seller_id = <materialized subquery>'.seller_id)) (cost=0..0 rows=0) (actual time=3.05..3.05 rows=0.0288 loops=521)
|         -> Limit: 1 row(s) (cost=0..0 rows=0) (actual time=3.05..3.05 rows=0.0288 loops=521)
|           -> Index lookup on <materialized subquery> using <auto distinct key> (seller_id=s.seller_id) (actual time=3.05..3.05 rows=0.0288 loops=521)
|             -> Materialize with deduplication (cost=0..0 rows=0) (actual time=1590..1590 rows=116 loops=1)
|               -> Filter: (avg(oi.freight_value) < 10) (actual time=1588..1589 rows=116 loops=1)
|                 -> Table scan on <temporary> (actual time=1588..1588 rows=2996 loops=1)
|                   -> Aggregate using temporary table (actual time=1588..1588 rows=2996 loops=1)
|                     -> Nested loop inner join (cost=58105 rows=111760) (actual time=49.3..1427 rows=111024 loops=1)
|                       -> Table scan on oi (cost=11819 rows=111760) (actual time=49.2..726 rows=112631 loops=1)
|                       -> Single-row index lookup on p using PRIMARY (product_id=oi.product_id) (cost=0.314 rows=1) (actual time=0.00592..0.00595 rows=0.986 loops=112631)
```

This index ensures that the outer query can rapidly match seller records against the list of `seller_ids` produced by the subquery. It speeds up the filtering process in the main query by quickly locating the relevant sellers based on their `seller_id`. Helps with filtering in the outer query's `WHERE ... IN (...)`.

QUERY 4:

```
-----+
| -> Table scan on <union temporary> (cost=30915..31849 rows=74499) (actual time=2625..2625 rows=1270 loops=1)
| -> Union materialize with deduplication (cost=30915..30915 rows=74499) (actual time=2625..2625 rows=1270 loops=1)
|   -> Filter: (olist_order_items_dataset.freight_value > 100) (cost=11733 rows=37250) (actual time=15.2..2567 rows=671 loops=1)
|     -> Table scan on olist_order_items_dataset (cost=11733 rows=111760) (actual time=4.07..2558 rows=112651 loops=1)
|     -> Filter: (olist_order_items_dataset.freight_value < 5) (cost=11733 rows=37250) (actual time=0.434..48.3 rows=1228 loops=1)
|       -> Table scan on olist_order_items_dataset (cost=11733 rows=111760) (actual time=0.387..41 rows=112651 loops=1)
|
|-----+
```

Index Choice: 3

We chose index 3 because it optimizes both the filtering on `freight_value` and access to `product_id`, which improves overall query performance by reducing unnecessary lookups.

1. Single column index on `freight_value`

Best for optimizing the **WHERE** condition.

`olist_order_items_dataset(freight_value);`

```
-----+
| -> Table scan on union temporary: (cost=2250..2322 rows=1899) (actual time=1628..1628 rows=1270 loops=1)
| -> Union materialize with deduplication (cost=2250..2250 rows=1899) (actual time=1628..1628 rows=1270 loops=1)
|   -> Index range scan on olist_order_items_dataset using idx_freight_value over (100 < freight_value), with index condition: (olist_order_items_dataset.freight_value > 100) (cost=744 rows=671) (actual time=0.523..1242 rows=671 loops=1)
|   -> Index range scan on olist_order_items_dataset using idx_freight_value over (0.523 < freight_value < 5), with index condition: (olist_order_items_dataset.freight_value < 5) (cost=1161 rows=1228) (actual time=0.52..276 rows=1228 loops=1)
|
|-----+
```

This single-column index is designed to accelerate the filtering process in both parts of the UNION query, where `freight_value` is compared against 100 and 5. By indexing `freight_value`, the database engine can rapidly locate rows that meet the high or low freight conditions, thus minimizing full table scans. However, since the query also retrieves `product_id` and `price`, additional lookups may be necessary once the matching rows are identified.

2. Composite index on (`freight_value`, `price`)

Improves performance if you're analyzing price ranges or doing additional filtering/sorting by price.

olist_order_items_dataset(freight_value, price);

```
| -> Table scan on union temporary (cost=1864..2010 rows=1899) (actual time=1751..1751 rows=1270 loops=1)
|   -> Union materialize with deduplication (cost=1864..1984 rows=1899) (actual time=1751..1751 rows=1270 loops=1)
|     -> Index range scan on olist_order_items_dataset using idx_freight_price over (100 < freight_value), with index condition: (olist_order_items_dataset.freight_value > 100) (cost=434 rows=671) (actual time=0.174..1304 rows=671 loops=1)
|     -> Index range scan on olist_order_items_dataset using idx_freight_price over (NULL < freight_value < 5), with index condition: (olist_order_items_dataset.freight_value < 5) (cost=1160 rows=1228) (actual time=1.111..439 rows=1228 loops=1)
|
| 1 row in set (1.93 sec)
```

This composite index enhances performance by starting with the `freight_value` column, which is used for filtering, and then including the price column, which is part of the `SELECT` list. The combination allows the engine to not only quickly identify rows with the desired `freight_value` criteria but also to more efficiently access the price data directly from the index. While this can reduce the number of additional lookups compared to a single-column index, the `product_id` column is not included, so the engine may still need to fetch that value separately.

3. Composite index on (freight_value, product_id)

Useful if you later expand this query to group by or join on product_id.

olist_order_items_dataset(freight_value, product_id);

```
| -> Table scan on union temporary (cost=1876..1902 rows=1899) (actual time=803..803 rows=1270 loops=1)
|   -> Union materialize with deduplication (cost=1876..1876 rows=1899) (actual time=803..803 rows=1270 loops=1)
|     -> Index range scan on olist_order_items_dataset using idx_freight_product over (100 < freight_value), with index condition: (olist_order_items_dataset.freight_value > 100) (cost=596 rows=671) (actual time=0.0801..315 rows=671 loops=1)
|     -> Index range scan on olist_order_items_dataset using idx_freight_product over (NULL < freight_value < 5), with index condition: (olist_order_items_dataset.freight_value < 5) (cost=1090 rows=1228) (actual time=0.052..484 rows=1228 loops=1)
|
| 1 row in set (1.93 sec)
```

This composite index is optimized by pairing `freight_value` with `product_id`. It efficiently supports the `WHERE` clause by quickly filtering based on `freight_value`, while simultaneously speeding up access to `product_id`, which is crucial for the query's output. Although this index improves lookup performance for the filtering and one of the selected columns, the price column is omitted from the index, potentially necessitating extra lookups to retrieve its value.

QUERY5

```
| -> Sort: total_reviews DESC, avg_score DESC (actual time=456607.456607 rows=997 loops=1)
|   -> Filter: ((total_reviews > 3) and (avg_score >= 4.5)) (actual time=455951.456602 rows=997 loops=1)
|     -> Table scan on <temporary> (actual time=455047.456591 rows=32172 loops=1)
|       -> Aggregate using temporary table (actual time=455047.455047 rows=32172 loops=1)
|         -> Nested loop inner join (cost=417650 rows=112495) (actual time=253.428170 rows=110749 loops=1)
|           -> Nested loop inner join (cost=293908 rows=112495) (actual time=152.426430 rows=110749 loops=1)
|             -> Nested loop inner join (cost=170409 rows=112495) (actual time=145.372551 rows=112371 loops=1)
|               -> Nested loop inner join (cost=119929 rows=98842) (actual time=144.218868 rows=99224 loops=1)
|                 -> Table scan on r (cost=11203 rows=98842) (actual time=83.8569 rows=99224 loops=1)
|                   -> Single-row covering index lookup on o using PRIMARY (order_id=r.order_id) (cost=1 rows=1) (actual time=2.12..2.12 rows=1 loops=99224)
|                     -> Index lookup on oi using PRIMARY (order_id=r.order_id) (cost=0.397 rows=1.14) (actual time=1.54..1.55 rows=1.13 loops=99224)
|                 -> Filter: (p.product_category_name is not null) (cost=0.998 rows=1) (actual time=0.478..0.478 rows=0.986 loops=112371)
|                   -> Single-row index lookup on p using PRIMARY (product_id=oi.product_id) (cost=0.998 rows=1) (actual time=0.477..0.477 rows=0.986 loops=112371)
|                     -> Single-row index lookup on pct using PRIMARY (product_category_name=p.product_category_name) (cost=1 rows=1) (actual time=0.015..0.015 rows=1 loops=110749)
|
```

Index Choice: none

The reason why indexes 2 and 3, as well as the version without indexing, perform the same is likely because the dataset is already small enough to fit in memory, allowing the database engine to perform full table scans efficiently. Additionally, the joins involved may already use primary key lookups, which are optimized by default. As a result, adding secondary indexes doesn't provide a noticeable performance improvement in this specific query.

1. Index for joining reviews to orders:

```
olist_order_reviews_dataset(order_id);
```

Speeds up JOIN r ON r.order_id = o.order_id


```

| -> Sort: total_reviews DESC, avg_score DESC (actual time=514903..514903 rows=997 loops=1)
    -> Filter: ((total_reviews > 5) and (avg_score >= 4.5)) (actual time=513445..514896 rows=997 loops=1)
        -> Table scan on <temporary> (actual time=513440..514884 rows=32172 loops=1)
            -> Aggregate using temporary table (actual time=513436..513438 rows=32171 loops=1)
                -> Nested loop inner join (cost=347381 rows=113485) (actual time=284..498418 rows=110749 loops=1)
                    -> Nested loop inner join (cost=222548 rows=112922) (actual time=183..252264 rows=111024 loops=1)
                        -> Nested loop inner join (cost=98334 rows=112922) (actual time=217..40058 rows=111024 loops=1)
                            -> Nested loop inner join (cost=38802 rows=32069) (actual time=65..1.4529 rows=32329 loops=1)
                                -> Filter: (p.product_category_name is not null) (cost=3526 rows=32069) (actual time=40..3869 rows=32329 loops=1)
                                    -> Covering index scan on p using product_category_name (cost=3526 rows=32069) (actual time=40..3811 rows=32329 loops=1)
                                        -> Single-row index lookup on pct using PRIMARY (product_category_name=p.product_category_name) (cost=1 rows=1) (actual time=0.0191..0.0192 rows=1 loops=32329)
                                            -> Covering index lookup on oi using idx oi_product_id (product_id=p.product_id) (cost=1.5 rows=3.52) (actual time=1.06..1.1 rows=3.43 loops=32329)
                                                -> Single-row covering index lookup on o using PRIMARY (order_id=oi.order_id) (cost=1 rows=1) (actual time=1.91..1.91 rows=1 loops=111024)
                                                    -> Index lookup on r using idx_reviews_order_id (order_id=oi.order_id) (cost=1 rows=1) (actual time=2.21..2.22 rows=0.998 loops=111024)
|

```

This index is created to speed up the join between the reviews and orders datasets by allowing the database to quickly locate matching order_id values. Since the query starts by joining reviews to orders on order_id, this index reduces the amount of data scanned and accelerates the first step of the join chain.

2. Index for joining order items with products:

olist_order_items_dataset(product_id);

Optimizes JOIN oi ON oi.product_id = p.product_id

This index improves the join performance between order items and products by making product_id lookups more efficient. It allows the database to quickly find all order items for a given product, which is important for aggregating review scores and counts by product.

```

| -> Sort: total_reviews DESC, avg_score DESC (actual time=515085..515089 rows=997 loops=1)
    -> Filter: ((total_reviews > 5) and (avg_score >= 4.5)) (actual time=513127..515071 rows=997 loops=1)
        -> Table scan on <temporary> (actual time=513127..515071 rows=32172 loops=1)
            -> Aggregate using temporary table (actual time=513127..515127 rows=32172 loops=1)
                -> Nested loop inner join (cost=377189 rows=112495) (actual time=262..476953 rows=110749 loops=1)
                    -> Nested loop inner join (cost=220948 rows=112495) (actual time=197..415320 rows=112371 loops=1)
                        -> Nested loop inner join (cost=119929 rows=98842) (actual time=115..244500 rows=99224 loops=1)
                            -> Table scan on r (cost=11203 rows=98842) (actual time=64.2..9423 rows=99224 loops=1)
                                -> Single-row covering index lookup on o using PRIMARY (order_id=r.order_id) (cost=1 rows=1) (actual time=2.37..2.37 rows=1 loops=99224)
                                    -> Index lookup on oi using PRIMARY (order_id=r.order_id) (cost=0.999 rows=1.14) (actual time=1.75..1.75 rows=1.13 loops=99224)
                                        -> Filter: (p.product_category_name is not null) (cost=0.998 rows=1) (actual time=0.506..0.507 rows=0.986 loops=112371)
                                            -> Single-row index lookup on p using PRIMARY (product_id=oi.product_id) (cost=0.998 rows=1) (actual time=0.508..0.505 rows=0.986 loops=112371)
                                                -> Single-row index lookup on pct using PRIMARY (product_category_name=p.product_category_name) (cost=1 rows=1) (actual time=0.0133..0.0134 rows=1 loops=110749)
|

```

3. Index for joining products with category translation:

olist_products_dataset(product_category_name);

Speeds up JOIN pct ON p.product_category_name = pct.product_category_name

This index supports the final join in the query, where the products dataset is joined with the category translation table using product_category_name. By indexing this column, the database can more efficiently match each product with its English category name, streamlining the grouping and output formatting process.

```

| -> Sort: total_reviews DESC, avg_score DESC (actual time=465182..465182 rows=997 loops=1)
    -> Filter: ((total_reviews > 5) and (avg_score >= 4.5)) (actual time=463917..465175 rows=997 loops=1)
        -> Table scan on <temporary> (actual time=463912..465164 rows=32172 loops=1)
            -> Aggregate using temporary table (actual time=463912..463912 rows=32172 loops=1)
                -> Nested loop inner join (cost=420762 rows=112495) (actual time=341..435950 rows=110749 loops=1)
                    -> Nested loop inner join (cost=229510 rows=112495) (actual time=333..434418 rows=110749 loops=1)
                        -> Nested loop inner join (cost=119929 rows=98842) (actual time=213..220500 rows=99224 loops=1)
                            -> Table scan on r (cost=11203 rows=98842) (actual time=111..3032 rows=99224 loops=1)
                                -> Single-row covering index lookup on o using PRIMARY (order_id=r.order_id) (cost=1 rows=1) (actual time=2.13..2.13 rows=1 loops=99224)
                                    -> Index lookup on oi using PRIMARY (order_id=r.order_id) (cost=0.995 rows=1.14) (actual time=1.6..1.6 rows=1.13 loops=99224)
                                        -> Filter: (p.product_category_name is not null) (cost=0.5 rows=1) (actual time=0.489..0.49 rows=0.986 loops=112371)
                                            -> Single-row index lookup on p using PRIMARY (product_id=oi.product_id) (cost=0.5 rows=1) (actual time=0.488..0.488 rows=0.986 loops=112371)
                                                -> Single-row index lookup on pct using PRIMARY (product_category_name=p.product_category_name) (cost=1 rows=1) (actual time=0.0131..0.0131 rows=1 loops=110749)
|

```

