Lab 3

 Graded

Student
Shruti Jayavardhanan
 View or edit group
Total Points
100 / 100 pts
Question 1
Lab report

100 / 100 pts

− 0 pts Correct

− 5 pts Not an index-only plan

− 5 pts Station table is outer and trip is inner.

− 5 pts Query plan missing

− 20 pts Missing query plan explanation

CSCI 4707 - Fall 2023 Lab 3 Report
Group:
<Student 1> (<Student 1 x500>)
<Student 2> (<Student 2 x500>)
<Student 3> (<Student 3 x500>)

Part 1. Import and Index Data (10 pts)
1.1 Attach a screenshot of running “\d” and listing all the tables.

```
jayav004@csel-kh1250-32:/home/jayav004/install $ bin/psql bikedb
psql (16.0)
Type "help" for help.

bikedb=# \d
                List of relations
 Schema |          Name          | Type  |  Owner
--------+------------------------+-------+----------
 public | station                | table | jayav004
 public | trip                   | table | jayav004
 public | trip_btree_clustered   | table | jayav004
 public | trip_btree_unclustered | table | jayav004
 public | trip_hash              | table | jayav004
(5 rows)
```

1.2 Attach a screenshot of running “\d trip”.

```
bikedb=# \d trip
                          Table "public.trip"
     Column      |            Type             | Collation | Nullable | Default
-----------------+-----------------------------+-----------+----------+---------
 ride_id         | text                        |           |          |
 rideable_type   | text                        |           |          |
 started_at      | timestamp without time zone |           |          |
 ended_at        | timestamp without time zone |           |          |
 start_station_id | integer                    |           |          |
 end_station_id  | integer                     |           |          |
 member_casual   | text                        |           |          |
```

1.3 Attach a screenshot of running “\d trip_hash”.

```
bikedb=# \d trip_hash
                        Table "public.trip_hash"
     Column      |            Type             | Collation | Nullable | Default
-----------------+-----------------------------+-----------+----------+---------
 ride_id         | text                        |           |          |
 rideable_type   | text                        |           |          |
 started_at      | timestamp without time zone |           |          |
 ended_at        | timestamp without time zone |           |          |
 start_station_id | integer                    |           |          |
 end_station_id  | integer                     |           |          |
 member_casual   | text                        |           |          |
Indexes:
    "trip_hash_index" hash (start_station_id)
```

1.4 Attach a screenshot of running “\d trip_btree_unclustered”.

```
bikedb=# \d trip_btree_unclustered
                    Table "public.trip_btree_unclustered"
     Column      |            Type             | Collation | Nullable | Default
-----------------+-----------------------------+-----------+----------+---------
 ride_id         | text                        |           |          |
 rideable_type   | text                        |           |          |
 started_at      | timestamp without time zone |           |          |
 ended_at        | timestamp without time zone |           |          |
 start_station_id | integer                    |           |          |
 end_station_id  | integer                     |           |          |
 member_casual   | text                        |           |          |
Indexes:
    "trip_btree_index_uc" btree (start_station_id)
```

1.5 Attach a screenshot of running “\d trip_btree_clustered”.

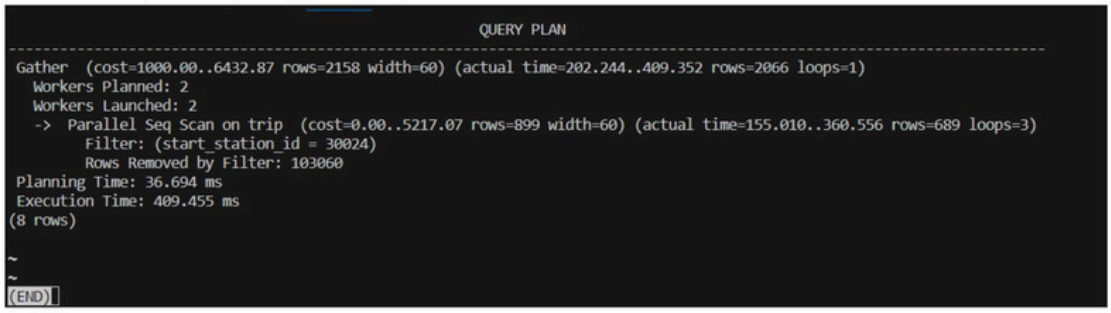
```
bikedb=# \d trip_btree_clustered
                    Table "public.trip_btree_clustered"
     Column      |            Type             | Collation | Nullable | Default
-----------------+-----------------------------+-----------+----------+---------
 ride_id         | text                        |           |          |
 rideable_type   | text                        |           |          |
 started_at      | timestamp without time zone |           |          |
 ended_at        | timestamp without time zone |           |          |
 start_station_id | integer                    |           |          |
 end_station_id  | integer                     |           |          |
 member_casual   | text                        |           |          |
Indexes:
    "trip_btree_index_c" btree (start_station_id) CLUSTER
```

Part 2. Single Table Lookup (50 pts)

For each query, what is the query plan and average execution time when using different trip tables (i.e., using different indexes)? To explain the query plan, take a screenshot of the query plan and try to explain it in a few sentences. To answer the average execution time, run a query three times and take the average.

2.1 Equal search: SELECT * FROM trip WHERE start_station_id = 30024; **(NOTE: I DID NOT USE THE FIRST EXECUTION TIME (EXECUTION TIMES SHOWN IN THE PICTURES) BECAUSE THEY ARE TOO DIFFERENT FROM THE SUBSEQUENT RESULTS)**

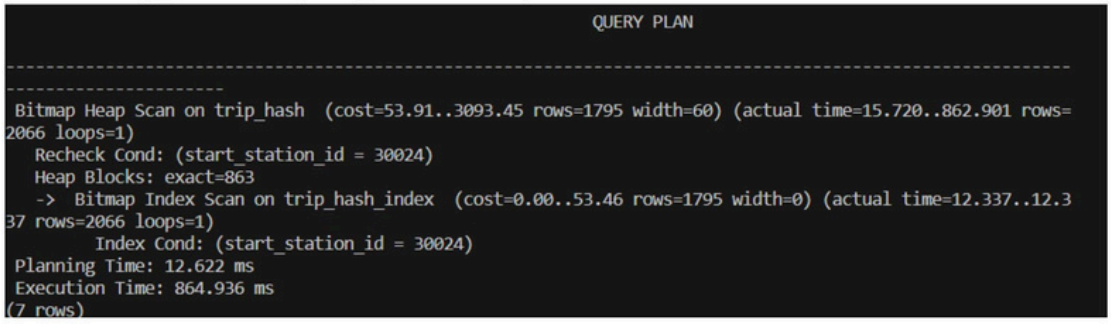
2.1-a Use trip table (using no index)



EXPLANATION: This query plan uses a parallel execution plan with worker processes. It starts by doing a parallel sequential scan on the 'trip' table. It filters rows based on the requirement that start_station_id = 30024. This sequential scan gives us 689 rows and filters out 103060 rows. The majority of the time is spent on the scanning part.

AVERAGE EXECUTION TIME (EXCLUDING THE EXECUTION TIME IN THE PICTURE): (22.452 ms + 19.971 ms + 20.977 ms) / 3 = 21.13 ms

2.1-b Use trip_hash table (using a hash index)



EXPLANATION:

This query plan involves a Bitmap Heap Scan followed by a Bitmap Index Scan on the 'trip_hash' table. At first, the Bitmap Index Scan filters rows efficiently based on the requirement. Later, he Bitmap Heap Scan retrieves the actual rows using the heap data, involving more detailed block operations, resulting in a higher execution time, with a higher portion of time spent on heap block operations

AVERAGE EXECUTION TIME: (3.105 ms + 3.027 ms + 3.046 ms) / 3= 3.05 ms

2.1-c Use trip_btree_unclustered (using unclustered b-tree index)

```
                                                                                    QUERY PLAN
-------------------------------------------------------------------------------------------------------------
--------------------------------------------
 Index Scan using trip_btree_index_uc on trip_btree_unclustered  (cost=0.42..2901.49 rows=2044 width=60) (a
ctual time=7.293..277.272 rows=2066 loops=1)
   Index Cond: (start_station_id = 30024)
 Planning Time: 12.172 ms
 Execution Time: 277.400 ms
(4 rows)
```

EXPLANATION: In this query plan, an Index Scan is performed using the 'trip_btree_index_uc' on the 'trip_btree_unclustered' table. The requirement is efficiently used to filter and get rows directly from the index. Most of the time is spent accessing the index and fetching the necessary rows that satisfy the condition. This method is efficient because it avoids scanning the entire table.

AVERAGE EXECUTION TIME: (1.730 ms + 1.623 ms + 1.704 ms) / 3 = 1.68 ms

2.1-d Use trip_btree_clustered (using clustered b-tree index)

```
                                                                                    QUERY PLAN
-------------------------------------------------------------------------------------------------------------
-------------------------------------------
 Index Scan using trip_btree_index_c on trip_btree_clustered  (cost=0.42..2678.34 rows=1878 width=60) (actu
al time=9.467..14.975 rows=2066 loops=1)
   Index Cond: (start_station_id = 30024)
 Planning Time: 6.470 ms
 Execution Time: 15.035 ms
(4 rows)
```

EXPLANATION: In this query plan, an Index Scan is executed using the 'trip_btree_index_c' on the 'trip_btree_clustered' table. The requirement is used to directly access rows from the index. The first execution time (in picture) is much lower compared to the other methods above. This indicates an efficient retrieval due to the index's clustered nature. The majority of time is spent accessing the index and fetching the necessary rows that satisfy the condition, resulting in a speedy execution of the query.

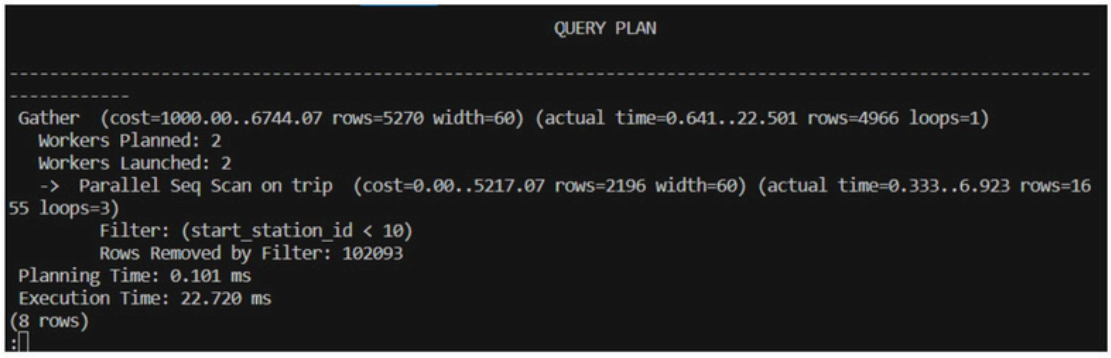
AVERAGE EXECUTION TIME: (0.866 ms + 0.825 ms + 0.864 ms) / 3 = 0.851 ms

2.1-e Rank the performance of different indexes on this query based on your observation.

A. No index B. Hash C. Unclustered b-tree D. Clustered b-tree

BEST TO WORST: D, C, B, A

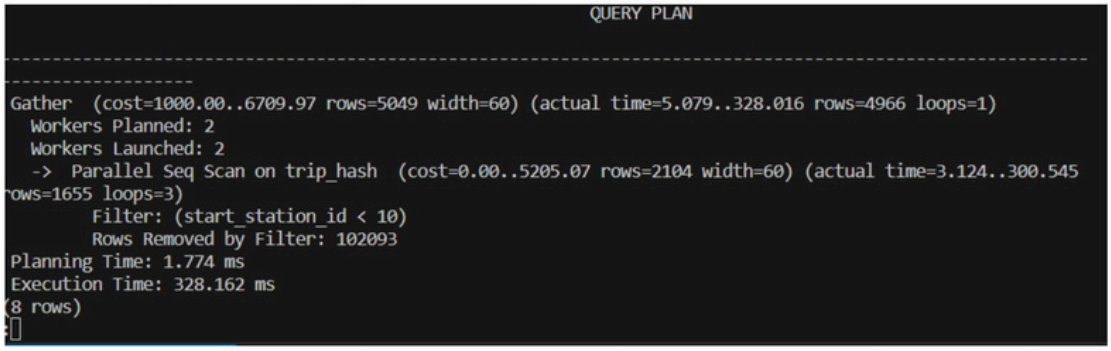
2.2 Range search: SELECT * FROM trip WHERE start_station_id < 10;
2.2-a Use trip table (using no index)



EXPLANATION: **In this query plan, a parallel sequential scan is conducted on the 'trip' table. The scan is executed by workers, where each worker performs a portion of the overall scan. The condition 'start_station_id < 10' is applied as a filter, which gives us a total of 4966 rows overall. However, 102,093 rows were excluded by this filter requirement.**

AVERAGE EXECUTION TIME: (22.175 + 20.443 ms + 20.234) / 3 = 20.95 ms

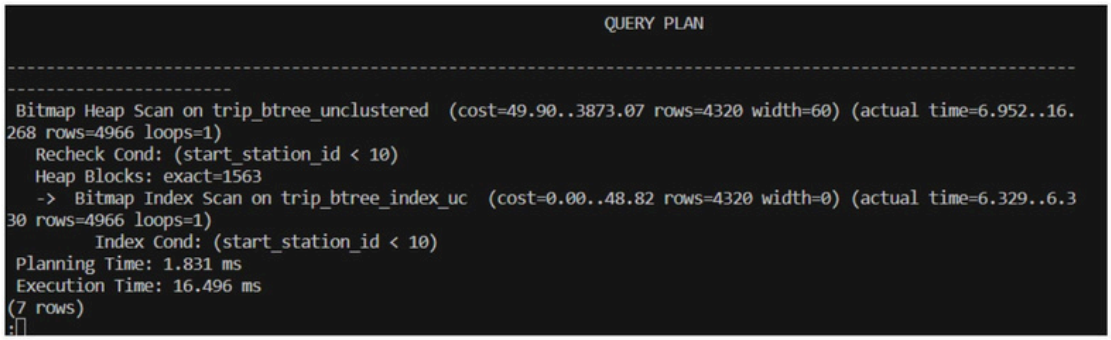
2.2-b Use trip_hash table (using a hash index)



EXPLANATION: **This query plan represents a parallel execution involving workers. It begins with a Gather operation coordinating these workers to execute a Parallel Sequential Scan on the table 'trip_hash'. During this scan, a filter condition is applied to retrieve rows where the 'start_station_id' is less than 10.**

AVERAGE EXECUTION TIME: (23.707 ms + 20.776 ms + 19.299 ms) = 21.26 ms

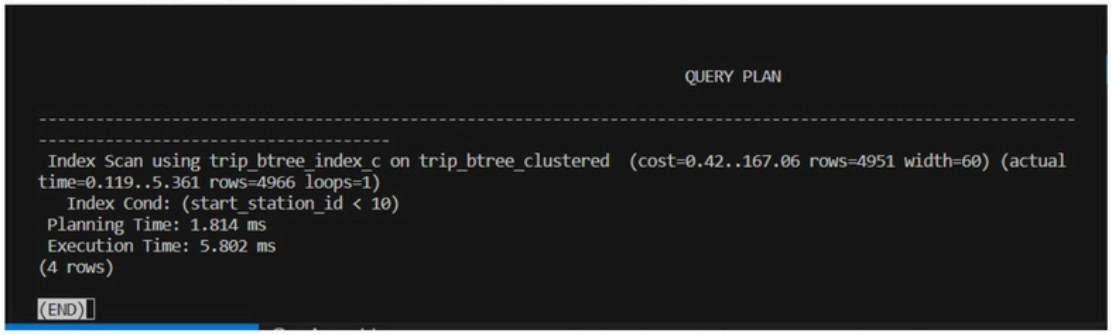
2.2-c Use trip_btree_unclustered (using unclustered b-tree index)



The provided query plan shows an execution involving a Bitmap Heap Scan and a Bitmap Index Scan on the table trip_btree_unclustered. The query aims to retrieve rows where the start_station_id is less than 10. The execution time for the first run is 16.496 ms.

AVERAGE EXECUTION TIME = (5.745+ 4.892 + 5.292) / 3 = 5.30 ms

2.2-d Use trip_btree_clustered (using clustered b-tree index)



This query plan utilizes an index scan, specifically the 'trip_btree_index_c', to retrieve data from the 'trip_btree_clustered' table. It applies a condition where the 'start_station_id' is less than 10 using the index. The execution time was 5.802 milliseconds, with the actual retrieval of 4966 rows meeting the specified condition.

AVERAGE EXECUTION TIME: (1.884 + 1.863 + 1.856)/ 3 = 1.86 ms

2.2-e Rank the performance of different indexes on this query based on your observation.
A. No index B. Hash C. Unclustered b-tree D. Clustered b-tree
D, C, A, B

Part 3. Join (25 pts)
For the query, what is the query plan and average execution time when using different trip tables? To answer the average execution time, run a query three times and take the average. To explain the query plan, take a screenshot of the query plan and answer:

1) Does it belong to any type of join algorithm we discussed in class, including simple/index nested loop join, sort-merge join, hash join, etc.
2) Is there an outer table for this join. If yes, which is the outer table?
3) Is there an inner table for this join? If yes, which is the inner table?
Join query: Find all trips starting from 'Coffman Union'
3-a Use trip table (using no index)

```
AND s.station_name = 'Coffman Union';
                                                                    QUERY PLAN
-----------------------------------------------------------------------------------------------------------------------------------------------------------
Nested Loop  (cost=0.42..2581.28 rows=1748 width=60) (actual time=0.045..4.186 rows=3625 loops=1)
  ->  Seq Scan on station s  (cost=0.00..7.59 rows=2 width=4) (actual time=0.023..0.090 rows=2 loops=1)
        Filter: (station_name = 'Coffman Union'::text)
        Rows Removed by Filter: 365
  ->  Index Scan using trip_start_station_id_idx on trip t  (cost=0.42..1278.22 rows=862 width=60) (actual time=0.012..1.413 rows=1812 loops=2)
        Index Cond: (start_station_id = s.station_id)
Planning Time: 0.348 ms
Execution Time: 4.533 ms
8 rows)
```

1. **Yes, it belongs to the hash join algorithm.**
2. **Yes, there is an outer table. The outer table is represented by the "trip" table alias 't'. It is evident based on this line (Parallel Seq Scan on trip t (cost=0.00..4892.85 rows=129685 width=60) (actual time=0.007..4.821 rows=103748 loops=3)**
3. **Yes, there is an inner table. The inner table is represented by the "s" / station table. (Hash Cond: (t.start_station_id = s.station_id)**

AVERAGE EXECUTION TIME: (41.331 ms + 24.244 ms + 47.699 ms) / 3 = 37.758 ms
THE QUERY PLAN: This plan indicates that the query is using parallel processing for scanning the 'trip' table, performing a hash join with the 'station' table based on a specific station name filter ('Coffman Union').

3-b Use trip_hash table (using a hash index)

```
-----------------------------------------------------------------------------------------------------------------------------------------------------------
Nested Loop  (cost=26.72..3839.86 rows=1754 width=60) (actual time=0.615..6.145 rows=3625 loops=1)
  ->  Seq Scan on station s  (cost=0.00..7.59 rows=2 width=4) (actual time=0.025..0.066 rows=2 loops=1)
        Filter: (station_name = 'Coffman Union'::text)
        Rows Removed by Filter: 365
  ->  Bitmap Heap Scan on trip_hash t  (cost=26.72..1907.47 rows=867 width=60) (actual time=0.586..2.641 rows=1812 loops=2)
        Recheck Cond: (start_station_id = s.station_id)
        Heap Blocks: exact=1729
        ->  Bitmap Index Scan on trip_hash_index  (cost=0.00..26.50 rows=867 width=0) (actual time=0.289..0.289 rows=1812 loops=2)
              Index Cond: (start_station_id = s.station_id)
Planning Time: 2.525 ms
Execution Time: 6.414 ms
11 rows)
```

1. **Yes, it belongs to the nested loop join algorithm.**
2. **The outer table is the station table (can be identified by Seq Scan on station s)**
3. **The inner table is the trip_hash table. In the plan, it's being accessed via a Bitmap Heap Scan on trip_hash t. This part of the plan is nested within the outer loop and is dependent on the results obtained from the outer table.**

AVERAGE EXECUTION TIME: (5.970 + 5.929 + 5.307) / 3 = 5.73 ms
QUERY PLAN: The query plan employs a Nested Loop join, first filtering two rows from the 'station' table based on the 'Coffman Union' station name. It then utilizes bitmap

scans and indexes on the 'trip_hash' table to retrieve 1812 rows where the 'start_station_id' matches the previously filtered station IDs.

3-c Use trip_btree_unclustered (using unclustered b-tree index)

```
------------------------------------------------------------------------------------------------
------
Nested Loop  (cost=0.42..2579.39 rows=1751 width=60) (actual time=0.080..7.986 rows=3625 loops=1)
  ->  Seq Scan on station s  (cost=0.00..7.59 rows=2 width=4) (actual time=0.022..0.039 rows=2 loops=1)
        Filter: (station_name = 'Coffman Union'::text)
        Rows Removed by Filter: 365
  ->  Index Scan using trip_btree_index_uc on trip_btree_unclustered t  (cost=0.42..1277.28 rows=862 width=60) (actual time=0.028..3.701 rows=1812 lo
ps=2)
        Index Cond: (start_station_id = s.station_id)
Planning Time: 5.469 ms
Execution Time: 8.149 ms
8 rows)
```

1. **Yes, the query plan has a Simple Nested Loop Join.**
2. **Yes, there is an outer table in this join, which is represented by the station table**
3. **The inner table is the trip_btree_unclustered table (Index Scan using trip_btree_index_uc on trip_btree_unclustered t (cost=0.42..1277.28 rows=862 width=60) (actual time=0.031..4.976 rows=1812 loops=2)**
   Index Cond: (start_station_id = s.station_id))

QUERY PLAN: The query plan begins with a Sequential Scan on the 'station' table to filter rows where the 'station_name' is 'Coffman Union', resulting in 2 rows after filtering out 365. Then, it performs an Index Scan on the 'trip_btree_unclustered' table using the 'trip_btree_index_uc', matching rows where 'start_station_id' equals 's.station_id' (derived from the outer table 's').
AVERAGE EXECUTION TIME: (4.460 + 4.522 + 4.538)/3 = 4.50 ms

3-d Use trip_btree_clustered (using clustered b-tree index)

```
                                                   QUERY PLAN
------------------------------------------------------------------------------------------------
--
 Nested Loop  (cost=0.42..143.84 rows=1746 width=60) (actual time=0.021..0.675 rows=3625 loops=1)
   ->  Seq Scan on station s  (cost=0.00..7.59 rows=2 width=4) (actual time=0.006..0.021 rows=2 loops=1)
         Filter: (station_name = 'Coffman Union'::text)
         Rows Removed by Filter: 365
   ->  Index Scan using trip_btree_index_c on trip_btree_clustered t  (cost=0.42..59.51 rows=862 width=60) (actual time=0.011..0.203 rows=1812 loops=2
)
         Index Cond: (start_station_id = s.station_id)
 Planning Time: 5.347 ms
 Execution Time: 0.753 ms
(8 rows)
```

1. **Yes, it uses Index Nested Loop Join.**
2. **Yes, there is an outer table involved, which is station s (-> Seq Scan on station s (cost=0.00..7.59 rows=2 width=4) (actual time=0.023..0.093 rows=2 loops=1))**
3. **The inner table is the trip_btree_clustered table (-> Index Scan using trip_btree_index_c on trip_btree_clustered t (cost=0.42..59.51 rows=862 width=60) (actual time=0.043..0.919 rows=1812 loops=2)**
   Index Cond: (start_station_id = s.station_id))

QUERY PLAN: : **The query plan begins by performing a Sequential Scan on the 'station' table, filtering rows based on the condition 'station_name = 'Coffman Union''. It retrieves 2 rows after filtering out 365. Afterwards, it conducts an Index Scan on the**

'trip_btree_clustered' table using the index 'trip_btree_index_c'. The requirement 'Index Cond: (start_station_id = s.station_id)' specifies the link between the 'start_station_id' from the inner table and 's.station_id' from the outer table. This filtering method fetches rows from the 'trip_btree_clustered' table that match the criteria defined by the outer table.
AVERAGE EXECUTION TIME: (2.912 + 1.492 + 2.967) / 3 = 2.457 ms

3-e Rank the performance of different indexes on joining based on your observation.
A. No index B. Hash C. Unclustered b-tree D. Clustered b-tree
Time-wise (best to worst) → D,C,B,A

Part 4. Index-only Plan (15 pts)
Write an SQL which is executed as an index-only plan. Attach a screenshot, which includes both your query and its query plan, and it must be index-only.

```
bikedb=# CREATE INDEX ON trip (start_station_id);
CREATE INDEX
bikedb=# EXPLAIN ANALYZE
SELECT start_station_id FROM trip WHERE start_station_id = 30024;
                                                                  QUERY PLAN
-----------------------------------------------------------------------------------------------------------------------------------------------
 Index Only Scan using trip_start_station_id_idx on trip  (cost=0.42..101.75 rows=2158 width=4) (actual time=1.282..1.805 rows=2066 loops=1)
   Index Cond: (start_station_id = 30024)
   Heap Fetches: 0
 Planning Time: 3.495 ms
 Execution Time: 2.071 ms
(5 rows)

bikedb=#
```