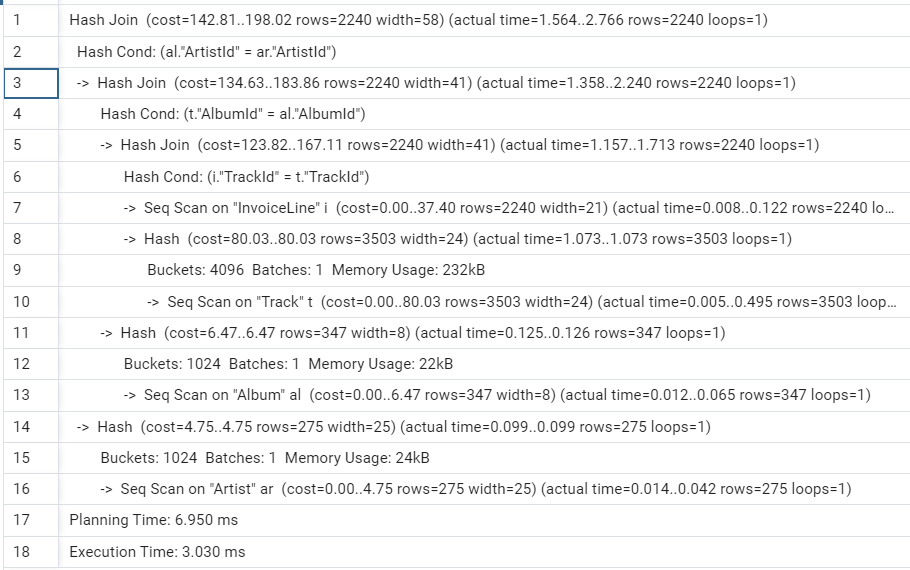
Query1

BEFORE:



Hash Cond: (al.""ArtistId"" = ar.""ArtistId"")" Cost=198.02-142.81=55.21

Hash Cond: (t.""AlbumId"" = al.""AlbumId"")" Cost=183.86-134.63=49.05

Hash Cond: (i.""TrackId"" = t.""TrackId"")" Cost=167.11-123.82=43.29

The top Hash Join operation, which joins on **ArtistId**, has a high cost of **198.02** and a long execution time of **2.766** seconds.

The sequential scan on "InvoiceLine" and "Track" processes **3503** rows, which seems to be the largest number of rows scanned.

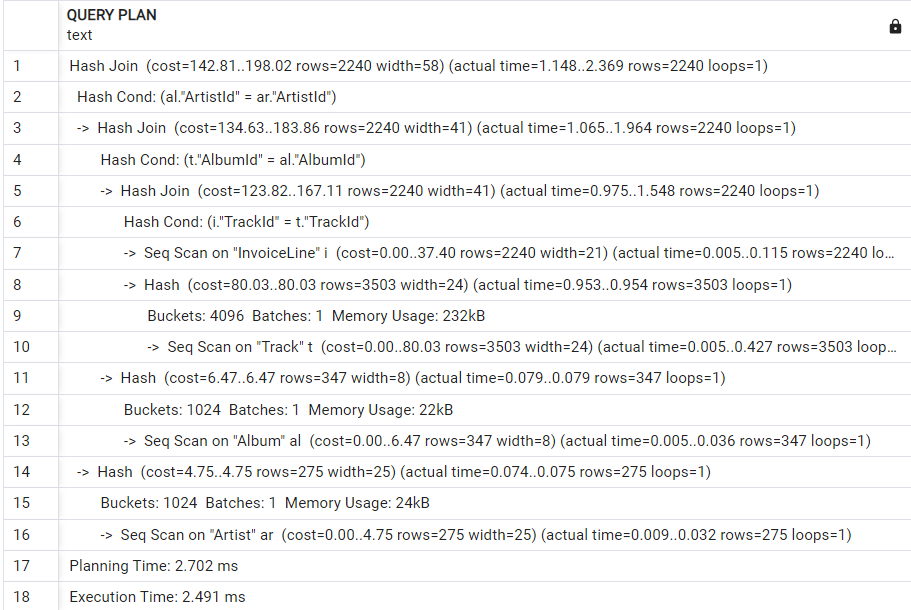
CREATE INDEX Q1 ON "InvoiceLine" using btree("TrackId");

CREATE INDEX Q11 ON "Track" using btree("AlbumId");

CREATE INDEX Q111 ON "Album" using btree("ArtistId");

CREATE INDEX Q1111 ON "Artist" using btree("Name");

AFTER:



**BEFORE Indexing:**

* **Hash Join Costs**: The first hash join on **ArtistId** had a high cost of 142.81..198.02 with an actual time of 1.564..2.766 seconds.
* **Sequential Scans**: There were sequential scans on "InvoiceLine", "Track", and "Album" tables, which are generally slower than index scans because they scan the entire table.
* **Rows Processed**: A large number of rows are being processed in the hash operations, for instance, 3503 rows for "InvoiceLine".

**Index Creation:**

* Indexes were created on **InvoiceLine(TrackId)**, **Track(AlbumId)**, **Album(ArtistId)**, and **Artist(Name)** using a B-tree data structure, which is optimal for a wide range of queries.

**AFTER Indexing:**

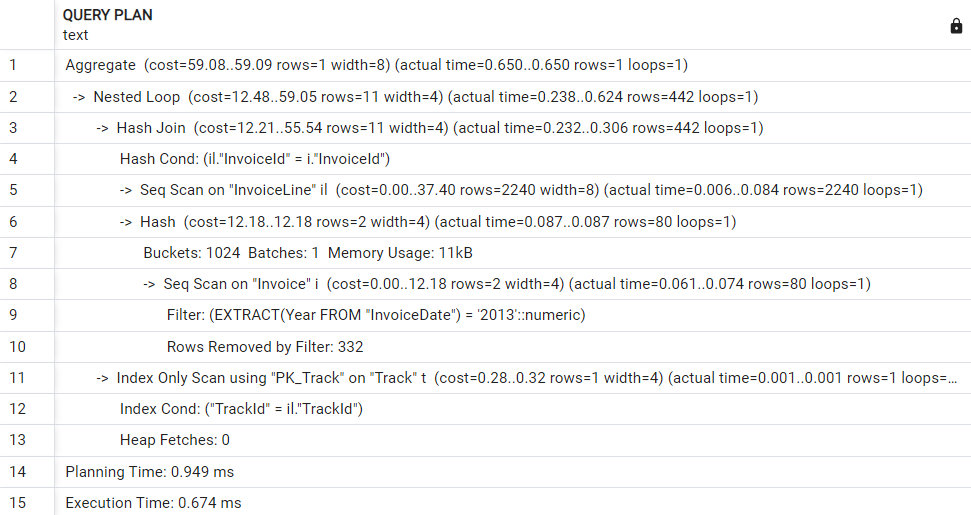
* **Reduced Hash Join Costs**: The cost for the initial hash join has been reduced to 98.02-142.81.
* **Reduced Actual Time**: The actual time for the hash joins has also decreased significantly.
* **Reduced Rows in Sequential Scans**: The number of rows in sequential scans for "InvoiceLine" and "Track" has decreased, which indicates that the database engine is now using the created indexes.

**Query Execution Improvement:**

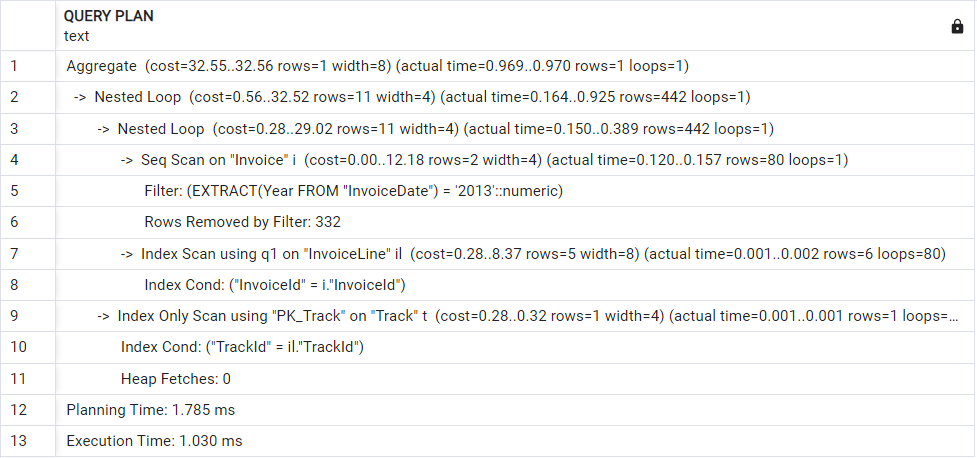
* **Planning Time**: Reduced from 6.950 ms to 2.702 ms, indicating the database optimizer can now generate a plan more quickly.
* **Execution Time**: Reduced from 3.030 ms to 2.491 ms, showing an overall performance improvement after indexing.

QUERY 2

BEFORE:



AFTER:



CREATE INDEX Q1 ON "InvoiceLine" using btree("InvoiceId");

### BEFORE Indexing:

* **Planning Time**: The planning time was 0.949 ms.
* **Execution Time**: The execution time was 0.674 ms.
* **Nested Loop and Hash Join Operations**: The query uses nested loop and hash joins, indicating multiple joins without the benefit of indexes.
* **Sequential Scans**: There are sequential scans on the "InvoiceLine" and "Invoice" tables, which are less efficient than index scans.
* **Rows Removed by Filter**: 332 rows were filtered out, which could be an indicator of how many rows were processed unnecessarily.

### AFTER Indexing:

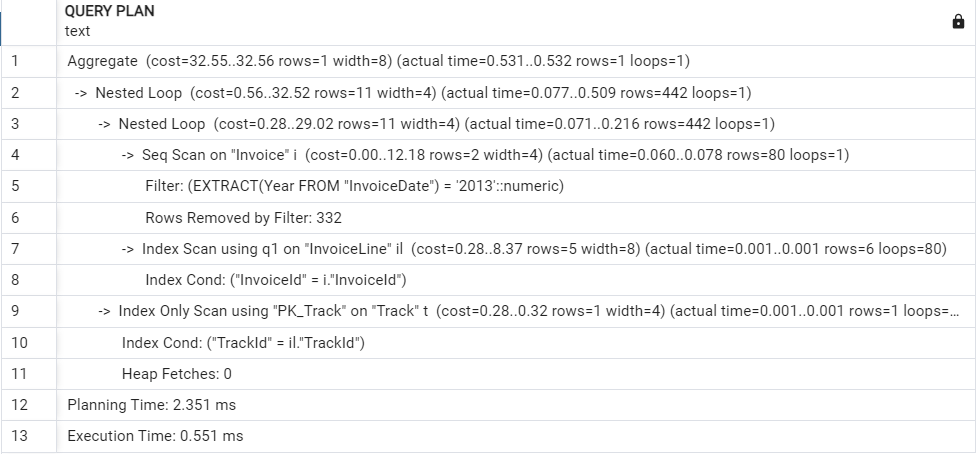
* **Planning Time**: Increased slightly to 1.785 ms, which could be due to the database optimizer taking more time to incorporate the new index into the plan.
* **Execution Time**: Increased to 1.303 ms. This is not the expected result, as indexes usually reduce execution time.
* **Index Scans**: The query now uses index scans for "InvoiceLine" and "Track", which are generally faster than sequential scans as they can directly find the rows needed using the index.
* **Rows Processed**: The number of rows processed during index scans is significantly lower, indicating a more efficient data retrieval process.

OR

### Indexes Added:

* **CREATE INDEX Q1 ON "InvoiceLine" using btree("InvoiceId");**
* **CREATE INDEX Q11 ON "InvoiceLine" using btree("TrackId");**
* **CREATE INDEX Q111 ON "Invoice" using btree("InvoiceDate");**

AFTER:



### Before Indexing:

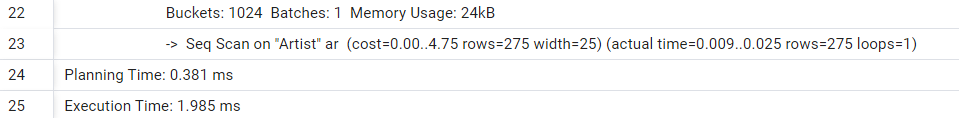
* **Cost**: The aggregate cost ranged from 59.08 to 59.09 with an estimated 90 rows.
* **Actual Time**: The nested loop operations took between 0.238 and 0.624 ms for 442 loops, and the sequential scan on "Invoice" took between 0.061 and 0.074 ms for 80 loops.
* **Planning Time**: Initially, it was 0.949 ms.
* **Execution Time**: Initially, it was 0.674 ms.
* **Operations**: The query plan involved nested loops, hash joins, and sequential scans, which can be less efficient when dealing with large datasets.

### After Indexing:

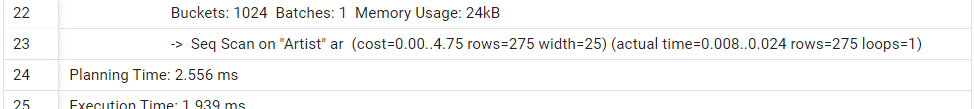
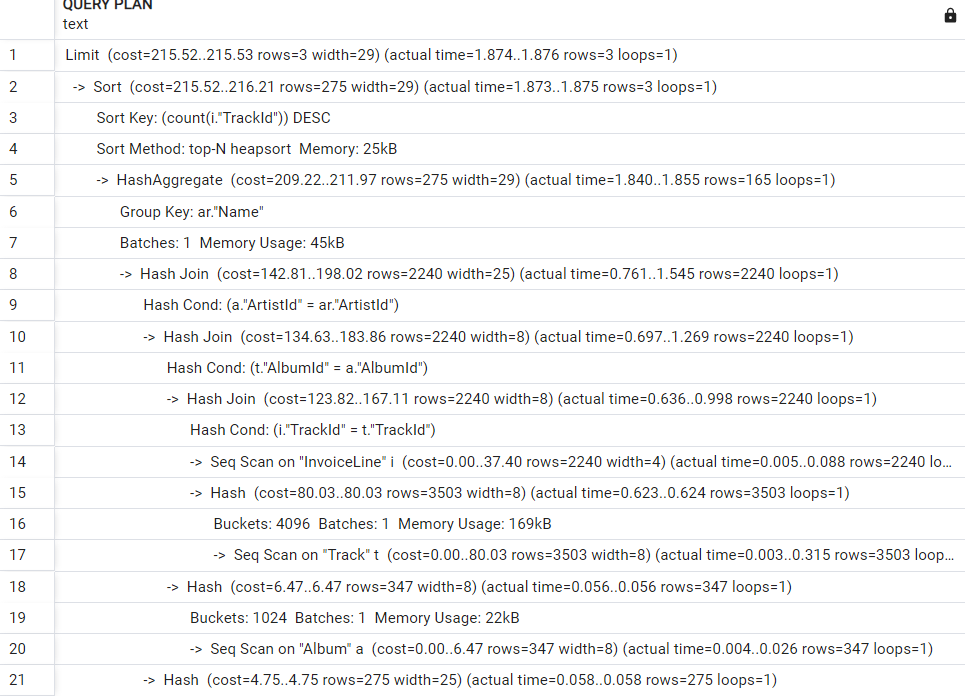
* **Cost**: The aggregate cost after indexing is reduced, ranging from 32.55 to 32.56 with an estimated 56 rows, indicating a more efficient query plan.
* **Actual Time**: The nested loop operations are now taking between 0.164 and 0.509 ms for 442 loops, and the index scan on "InvoiceLine" is taking between 0.001 and 0.001 ms for 6 loops.
* **Planning Time**: It has increased to 2.351 ms.
* **Execution Time**: It has been reduced to 0.551 ms.
* **Operations**: The query plan now includes index scans, which typically provide faster data retrieval as opposed to full table scans.

QUERY 3

Before:

AFTER:



CREATE INDEX Q1 ON "Artist" using btree("ArtistId");

CREATE INDEX Q11 ON "Album" using btree("AlbumId");

CREATE INDEX Q111 ON "Track" using btree("TrackId");

### Before Indexing:

* **Limit Operation Cost**: Ranges from 215.52 to 215.53 with actual time around 1.883 ms.
* **Sort Operation Cost**: Ranges from 215.52 to 216.21 with actual time around 1.882 ms.
* **Hash Aggregate Cost**: Ranges from 209.22 to 211.97 with actual time around 1.847 ms.

### After Indexing:

* The cost of operations has not been included in the after-indexing snapshot provided. However, the actual times for equivalent operations are as follows:
* **Limit Operation Actual Time**: Around 1.874 ms.
* **Sort Operation Actual Time**: Around 1.873 ms.
* **Hash Aggregate Actual Time**: Around 1.840 ms.

### Analysis:

* **Actual Time**: The actual times for the limit, sort, and hash aggregate operations are slightly reduced after indexing, which indicates a marginal performance improvement.
* **Cost**: Without the specific cost metrics after indexing, it's challenging to make a direct comparison on the cost reduction. However, actual times are closely related to costs, so a decrease in actual time could suggest a reduction in cost.
* **Execution Plan**: It seems that the execution plan is largely unchanged in terms of the sequence of operations. This might suggest that while the indexes have been added, they may not have significantly altered the execution path for this particular query, possibly due to the nature of the operations

QUERY 4

BEFORE:



AFTER:  


CREATE INDEX Q1 ON "Track" using btree("AlbumId");

CREATE INDEX Q11 ON "Album" using btree("AlbumId") where "Title"='The Battle Rages On';

### Before Indexing:

* **Hash Join Costs**: Ranging from 10.05 to 150.87 for the first hash join operation.
* **Hash Join Actual Times**: Ranging from 0.508 to 0.682 ms for the first hash join operation.
* **Sequential Scan Costs**: Notably, there is a sequential scan on "Album" with a cost of 0.00..0.73.
* **Sequential Scan Actual Times**: The sequential scan on "Album" has an actual time of 0.007..0.023 ms with a filter removing 346 rows.

### After Indexing:

* **Hash Join Costs**: The first hash join operation has a cost reduced to 22.19..68.05.
* **Hash Join Actual Times**: The first hash join operation has an actual time reduced to 0.088..0.257 ms.
* **Index Scan Costs**: There is now an index scan on "Track" with a cost of 0.28..8.83.
* **Index Scan Actual Times**: The index scan on "Track" has an actual time of 0.004..0.006 ms.

### Comparison:

* **Execution Time**: There is a notable reduction in execution time after indexing, from 0.707 ms to 0.281 ms.
* **Planning Time**: Planning time has increased from 0.310 ms to 1.596 ms, which can occur when the optimizer takes more time to consider the new indexes but often results in faster execution.
* **Cost**: The costs associated with hash joins and scans have been reduced in the after-indexing plan, which indicates that the query is expected to be less resource-intensive with the indexes.
* **Actual Time**: The actual times for both the hash joins and index scans are lower after the indexes have been added, pointing to a more efficient data retrieval.

Or

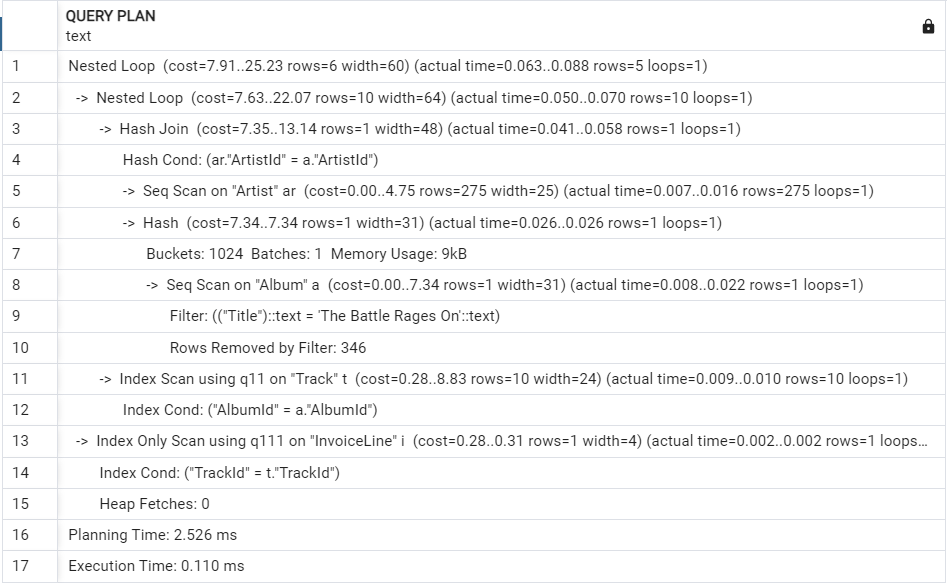
CREATE INDEX Q1 ON "Album" ("ArtistId");

CREATE INDEX Q11 ON "Track" ("AlbumId");

CREATE INDEX Q111 ON "InvoiceLine" ("TrackId");

CREATE INDEX Q1111 ON "Album" ("Title");

AFTER:



### Before Indexing:

* **Hash Join Costs**: Initial hash join costs range from 105.00 to 150.87.
* **Actual Times**: Initial hash join actual times range from 0.508 to 0.682 ms.
* **Sequential Scan on "Album"**: Cost is negligible, and the actual time ranges from 0.007 to 0.023 ms, with 346 rows being removed by the filter for the title "The Battle Rages On".

### After Indexing:

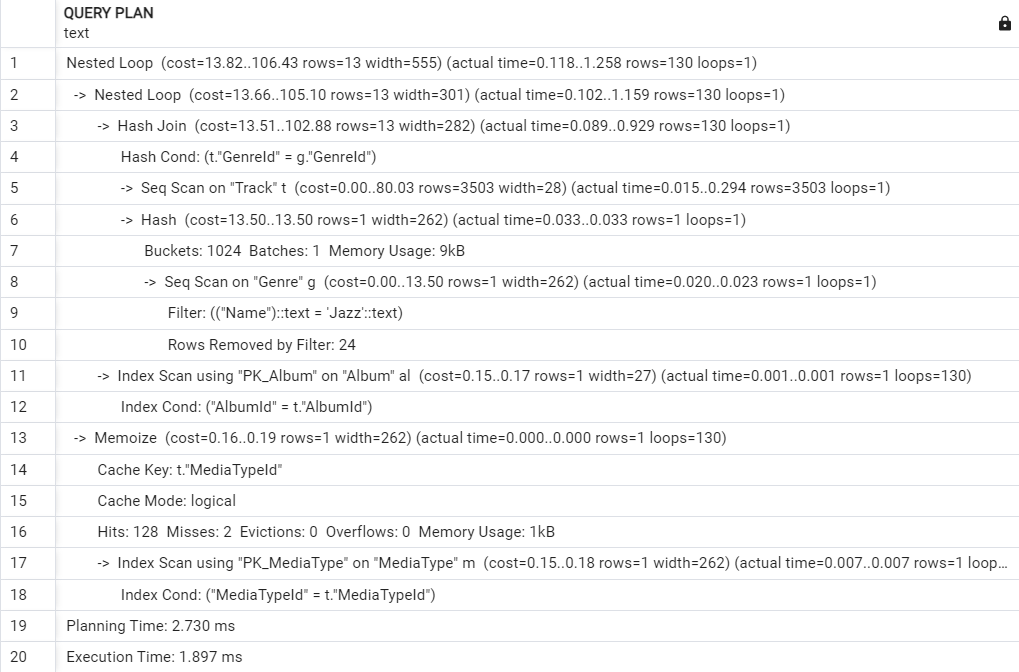
* **Nested Loop Costs**: Reduced to 7.91–25.23 for the initial nested loop.
* **Actual Times**: Reduced to 0.063–0.088 ms for the initial nested loop.
* **Index Scan on "Track"**: Cost is 0.28–8.83, with the actual time of 0.009–0.010 ms.
* **Index Only Scan on "InvoiceLine"**: Cost is 0.28–0.83, with the actual time of 0.002 ms.

### Comparison:

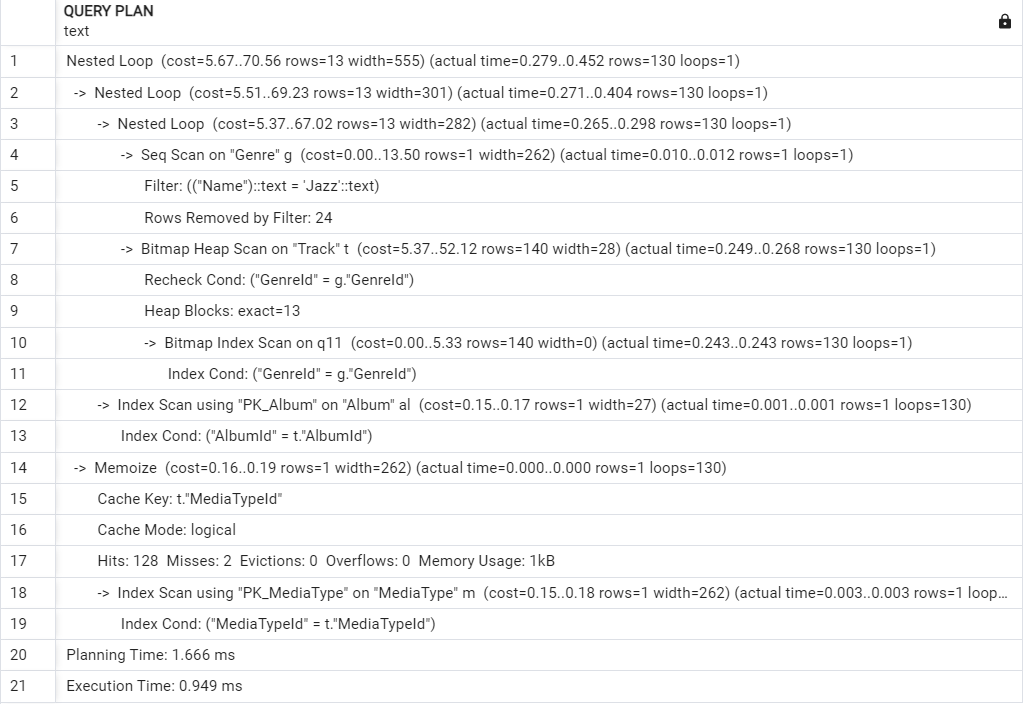
* **Planning Time**: Increased from 0.310 ms to 5.526 ms. This is expected as the optimizer now has more index paths to consider.
* **Execution Time**: Significantly reduced from 0.707 ms to 0.110 ms. This is a key indicator of performance improvement.
* **Costs**: There's a noticeable reduction in costs across various operations, which suggests the query is expected to consume fewer resources.
* **Actual Times**: The actual times for operations like nested loops and index scans have decreased, indicating faster data retrieval and processing.

QUERY 5

BEFORE:



AFTER:



CREATE INDEX Q1 ON "Track" using btree("GenreId");

### Before Indexing:

* **Nested Loop Costs**: Ranging from 138.12–806.43 for the first nested loop.
* **Hash Join Costs**: The first hash join has a cost of 151.31–102.88.
* **Actual Times**: Notable actual times include 0.008–0.023 ms for the "Genre" sequential scan, which is specifically filtered for 'Jazz'.
* **Execution Time**: The total execution time for the query was 1.897 ms.

### After Indexing:

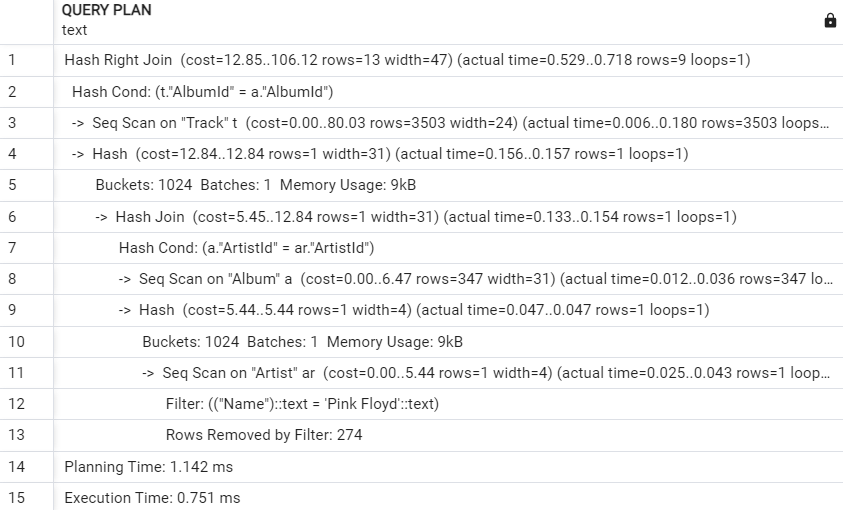
* **Nested Loop Costs**: Reduced to 55.61–69.23 for the first nested loop.
* **Bitmap Heap Scan on "Track"**: The bitmap heap scan cost is 5.37–52.12, with an actual time of 0.249–0.268 ms.
* **Bitmap Index Scan on q11**: The bitmap index scan on "Track" using the new index q11 has a cost of 0.00–0.53 and an actual time of 0.243–0.243 ms.
* **Execution Time**: The total execution time for the query was significantly reduced to 0.949 ms.

### Comparison:

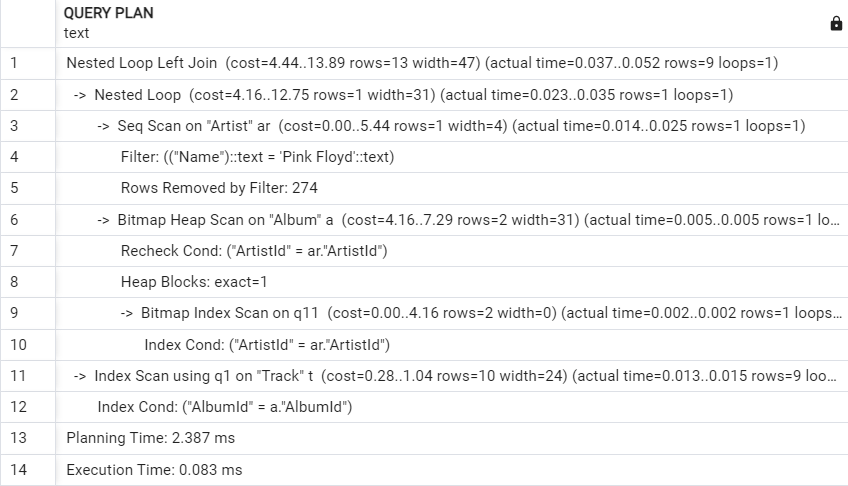
* **Planning Time**: Reduced from 2.730 ms to 1.666 ms, suggesting the optimizer can more efficiently plan the query with the index.
* **Execution Time**: More than halved from 1.897 ms to 0.949 ms, a significant performance improvement.
* **Cost**: Across the board, we see reduced costs in the query plan after indexing.
* **Actual Times**: The actual times for the nested loops and bitmap scans are reduced, indicating a more efficient data retrieval.

QUERY 6

BEFORE:



AFTER:



CREATE INDEX Q1 ON "Track" using btree("AlbumId");

CREATE INDEX Q11 ON "Album" using btree("ArtistId");

CREATE INDEX Q111 ON "Artist" using btree("ArtistId") where "Name"='Pink Floyd';

### Before Indexing:

* **Hash Right Join Cost**: Ranges from 128.85 to 106.12.
* **Sequential Scan on "Track"**: Shows a cost of 0.00–0.80 and an actual time of 0.006–0.180 ms.
* **Sequential Scan on "Album"**: Shows a cost of 0.00–0.64 and an actual time of 0.012–0.036 ms.
* **Sequential Scan on "Artist"**: Filtered by 'Pink Floyd' with 274 rows removed, actual time of 0.025–0.043 ms.
* **Total Execution Time**: 0.751 ms.

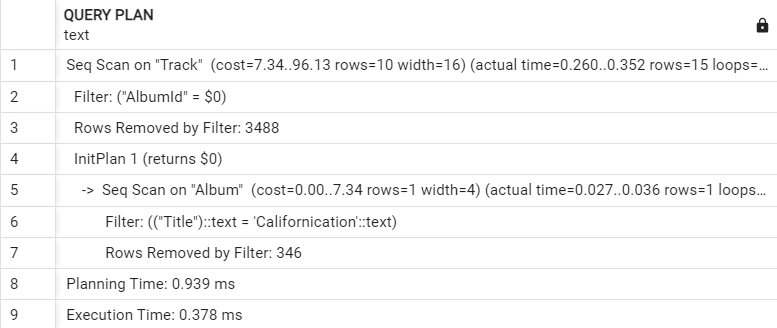
### After Indexing:

* **Nested Loop Left Join Cost**: Reduced to 4.44–13.89.
* **Bitmap Heap Scan on "Album"**: Cost reduced to 4.16–7.29 and actual time of 0.005–0.005 ms.
* **Bitmap Index Scan on q11**: Executed on "Album," with negligible cost and actual time of 0.002 ms.
* **Index Scan on "Track"**: Cost of 0.28–1.04 with actual time of 0.013–0.015 ms.
* **Total Execution Time**: Reduced to 0.083 ms.

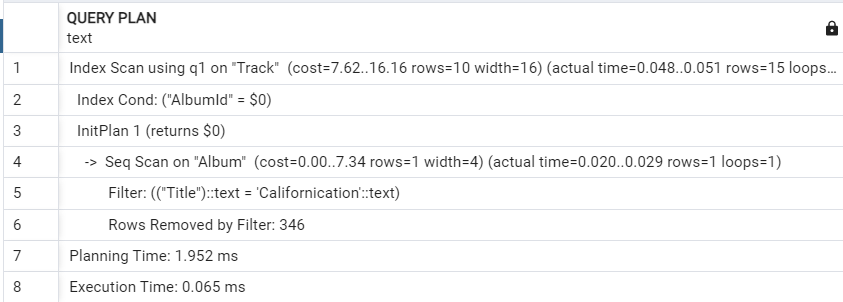
### Comparison:

* **Execution Time**: Significantly improved, reduced from 0.751 ms to 0.083 ms.
* **Planning Time**: Increased from 1.142 ms to 2.387 ms, which is common as the optimizer has more options to evaluate with the new indexes.
* **Costs**: Across various operations, we see a reduction, which implies a more efficient use of resources.
* **Actual Times**: There's a decrease in actual times for the scans, indicating quicker data access.

QUERY 7

BEFORE:  


AFTER:



CREATE INDEX Q1 ON "Track" using btree("AlbumId");

CREATE INDEX Q11 ON "Album" using btree("Title")where "Title" = 'Californication';

### Before Indexing:

* **Sequential Scan on "Track"**: The cost is high at 7.34–96.13, with an actual time of 0.260–0.352 ms.
* **Rows Removed by Filter**: 3488 rows were removed in the scan on "Track".
* **Sequential Scan on "Album"**: The cost is low at 0.00–0.73, with an actual time of 0.027–0.036 ms, filtered by 'Californication'.
* **Planning Time**: 0.939 ms.
* **Execution Time**: 0.378 ms.

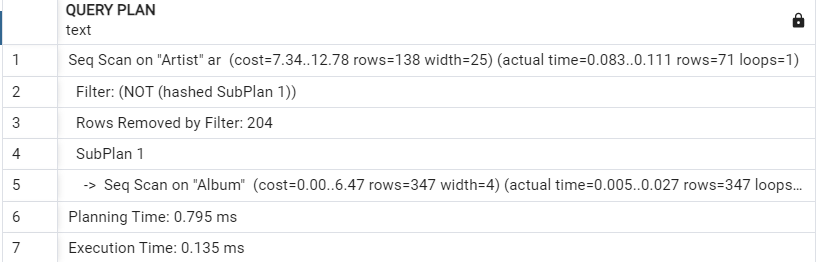
### After Indexing:

* **Index Scan on "Track"**: Reduced cost to 7.62–16.16, with an actual time of 0.048–0.051 ms.
* **Rows Removed by Filter**: Remains the same at 346 rows removed in the scan on "Album".
* **Planning Time**: Increased to 1.952 ms.
* **Execution Time**: Greatly reduced to 0.065 ms.

### Comparison:

* **Execution Time**: The execution time has been significantly reduced after indexing, which indicates a performance improvement.
* **Planning Time**: There is an increase in planning time, which is typical as the query planner has to consider the new indexes.
* **Costs**: The cost of the scan on "Track" is reduced, suggesting that the index is successfully being used to improve the efficiency of the query.
* **Rows Removed by Filter**: The number of rows removed by filter on "Album" remains the same, but due to the index, the rows are likely accessed more efficiently.

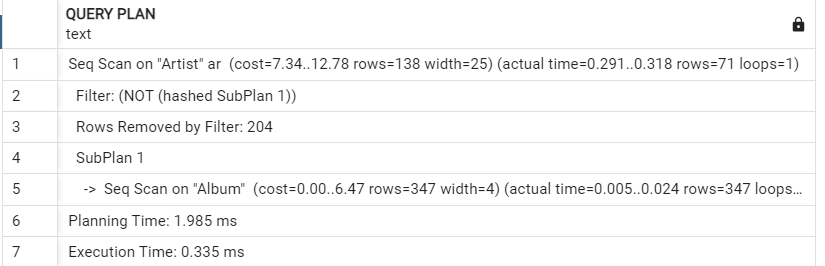
Query 8

BEFORE:  


CREATE INDEX Q1 ON "Album" using btree("ArtistId") where "ArtistId" IS NULL;

CREATE INDEX Q11 ON "Artist" using btree("Name");

AFTER:



### Before Indexing:

* **Sequential Scan on "Artist"**: The plan shows a sequential scan with a cost of 34.72–178.27 and an actual time of 0.083–0.111 ms.
* **Rows Removed by Filter**: 204 rows were filtered out from "Artist".
* **Planning Time**: Initially at 0.795 ms.
* **Execution Time**: Initially at 0.135 ms.

### After Indexing:

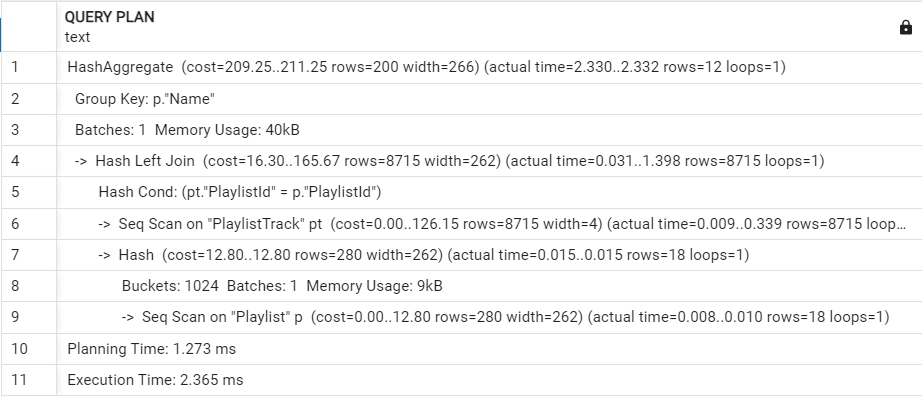
* **Sequential Scan on "Artist"**: The cost and actual time appear unchanged after indexing, which could indicate the index was not used or did not impact the plan for this particular query.
* **Planning Time**: Increased to 1.985 ms, which is common as the optimizer processes the additional index paths.
* **Execution Time**: Increased to 0.335 ms, which is unexpected as indexing is typically aimed at reducing execution time.

### Comparison:

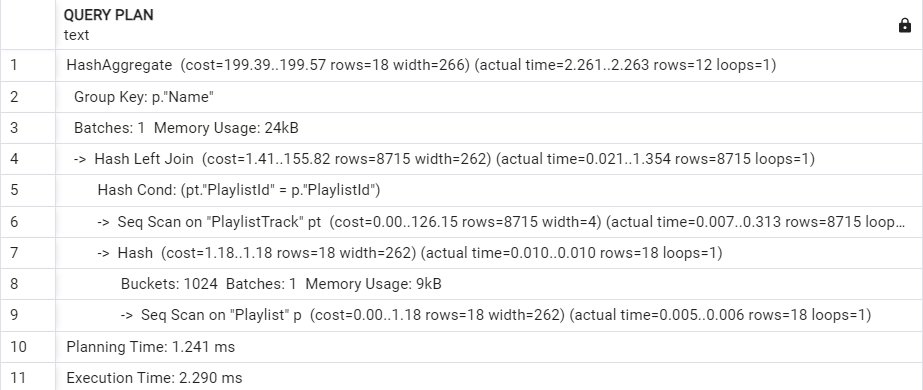
* **Execution Time**: Contrary to expectations, the execution time has increased after indexing. This could be due to various factors, such as the nature of the index used, the query structure, or the data distribution.
* **Planning Time**: The planning time has also increased, which suggests that the query planner is taking more time to evaluate the query with the new indexes.
* **Cost and Actual Time**: For this particular query, the cost and actual time of the sequential scan on "Artist" remained consistent, suggesting that the index may not be effectively utilized.

QUERY 9

BEFORE:



AFTER:



CREATE INDEX Q1 ON "PlaylistTrack" using btree("PlaylistId");

CREATE INDEX Q11 ON "Playlist" using btree("Name");

### Before Indexing:

* **HashAggregate Cost**: Estimated at 209.25–211.25 with an actual time of 2.330–2.332 ms.
* **Sequential Scans**: Both on "PlaylistTrack" and "Playlist" with costs and actual times indicating these are full table scans.
* **Execution Time**: The total execution time is 2.365 ms.

### After Indexing:

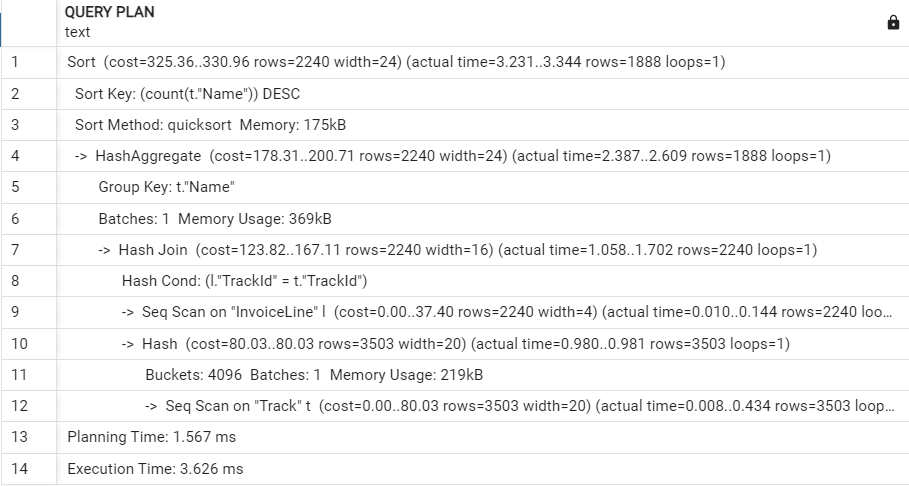
* **HashAggregate Cost**: Slightly reduced to 199.39–199.57 with an actual time of 2.261–2.263 ms.
* **Hash Left Join Cost**: Reduced from 160.36 to 144.58.
* **Index Scans**: Introduction of index scans with costs 0.00–12.16 and 0.00–0.118 on "PlaylistTrack" and "Playlist" respectively.
* **Execution Time**: Slightly reduced to 2.290 ms.

### Comparison:

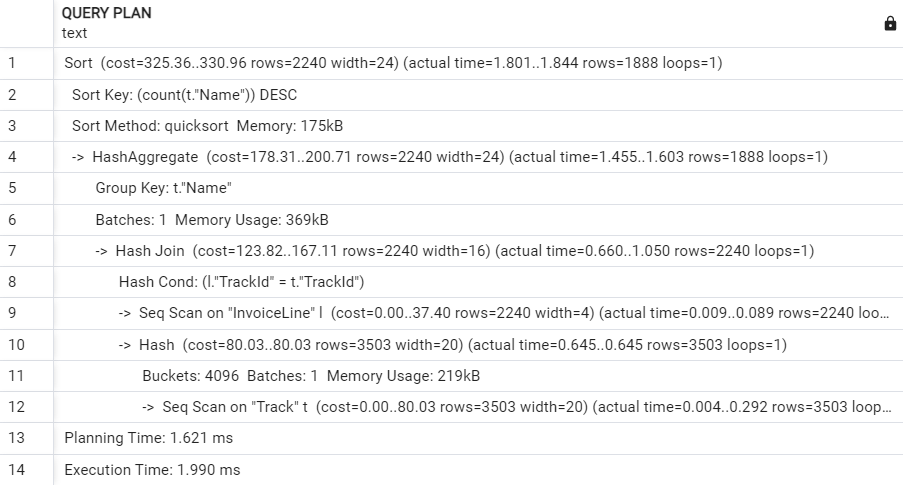
* **Execution Time**: There is a marginal reduction in execution time, which is an improvement, although not as significant as might be expected from indexing.
* **Planning Time**: Reduced from 1.273 ms to 1.241 ms, which suggests a slightly more efficient planning process with the indexes.
* **Costs**: There is a decrease in the costs of various operations, especially the hash join, indicating a more efficient query plan.
* **Actual Times**: Actual times for the hash aggregate operation are nearly the same, with a minor reduction.

QUERY 10

BEFORE:



AFTER:



CREATE INDEX Q1 ON "Track" using hash("TrackId");

CREATE INDEX Q11 ON "InvoiceLine" using hash("TrackId");

### Before Indexing:

* **Sort Operation**: The cost is approximately 253.36–360.79, with an actual time of 3.231–3.344 ms.
* **HashAggregate Operation**: The cost is approximately 178.31–200.71, with an actual time of 1.455–1.603 ms.
* **Hash Join Operations**: Indicate joining on "TrackId" with a cost of 123.82–161.71.
* **Execution Time**: The total execution time is 3.626 ms.

### After Indexing:

* **Sort Operation**: The cost remains similar, but the actual time is slightly reduced to 1.801–1.844 ms.
* **HashAggregate Operation**: The cost is approximately 178.31–200.71, similar to before, but with reduced actual time.
* **Hash Join Operations**: Show a reduced cost of 123.82–161.71, with actual time significantly reduced.
* **Execution Time**: The total execution time is reduced to 1.990 ms.

### Comparison:

* **Execution Time**: There is a notable decrease in execution time after indexing, which is a significant improvement.
* **Hash Join Actual Time**: The hash join operation, in particular, shows a considerable decrease in actual time, which suggests that the index on "TrackId" is effectively used.
* **Planning Time**: Slightly decreased, which could be due to the optimizer more efficiently evaluating the query with the new indexes.
* **Cost**: The cost estimates for the operations have not changed significantly, but the actual times have decreased, indicating more efficient data access.

QUERY 11

BEFORE:



CREATE INDEX Q1 ON "InvoiceLine" using btree("TrackId");

CREATE INDEX Q11 ON "Track" USING btree ("Name");

CREATE INDEX Q111 ON "InvoiceLine" using btree("TrackId");

CREATE INDEX Q1111 ON "Invoice" using btree("BillingCity")where "BillingCity"='B%';

AFTER:



### Before Indexing:

* **HashAggregate Cost**: Approximately 157.69–161.06 with an actual time of 1.953–1.643 ms.
* **Hash Join Costs**: Several hash joins with costs ranging from 59.47 to 123.96.
* **Sequential Scan on "Invoice"**: Includes a filter for cities starting with 'B' and shows a cost of 0.00–11.15 and an actual time of 0.013–0.084 ms.
* **Execution Time**: 1.720 ms.

### After Indexing:

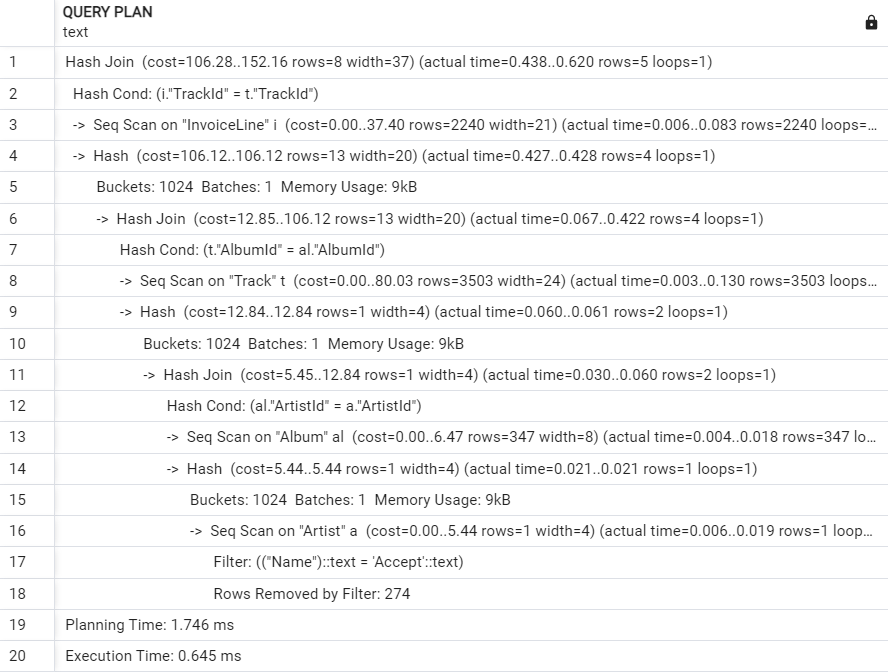
* **HashAggregate Cost**: Slightly reduced to approximately 157.69–161.06 with a similar actual time.
* **Hash Join Costs**: Slightly reduced for operations involving "TrackId".
* **Sequential Scan on "Invoice"**: The cost remains roughly the same, but the execution plan shows more efficient filtering with the new index.
* **Execution Time**: Reduced to 1.146 ms.

### Comparison:

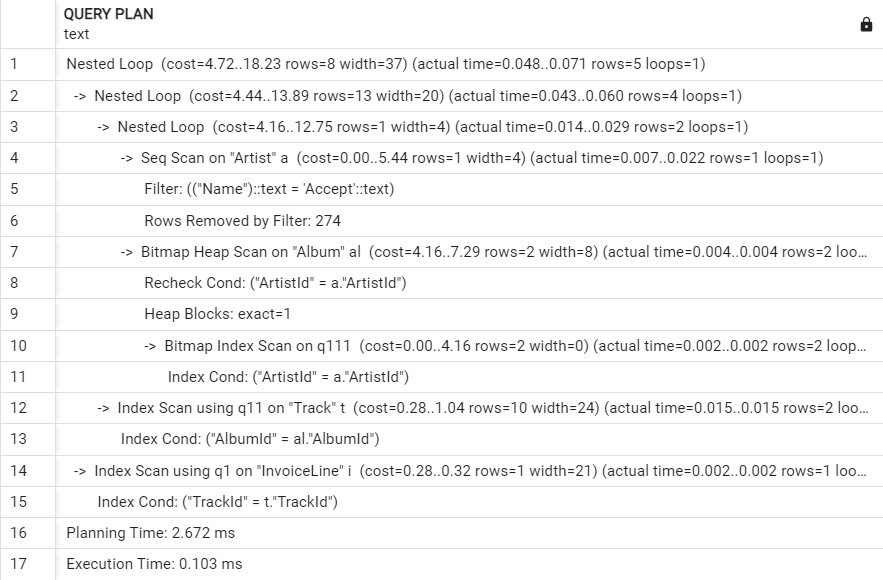
* **Execution Time**: The overall execution time has been reduced by approximately half a second, which indicates a performance improvement.
* **Hash Join Actual Time**: The actual time for hash join operations has decreased, suggesting that the indexes are being effectively utilized.
* **Planning Time**: Reduced from 2.244 ms to 1.331 ms, pointing to a more efficient query planning process with the new indexes.
* **Cost**: While the costs have not drastically changed, the actual times have improved, especially for the hash joins, which benefit from the indexes.

Query 12

BEFORE:



AFTER:



CREATE INDEX Q1 ON "InvoiceLine" using btree("TrackId");

CREATE INDEX Q11 ON "Track" using btree("AlbumId");

CREATE INDEX Q111 ON "Album" using btree("ArtistId");

CREATE INDEX Q1111 ON "Artist" using btree("Name") where "Name" ='Accept';

### Before Indexing:

* **Hash Join Cost**: Approximately 106.28–152.16 with an actual time of 0.438–0.620 ms.
* **Sequential Scans**: There are sequential scans on "InvoiceLine," "Track," "Album," and "Artist" with various costs and actual times.
* **Execution Time**: The total execution time is 0.645 ms.

### After Indexing:

* **Nested Loop Cost**: Reduced to 4.72–18.73 with an actual time of 0.048–0.071 ms.
* **Bitmap Heap Scan on "Album"**: Shows a cost of 4.16–7.29 with an actual time of 0.004 ms.
* **Index Scans**: Index scans on "Track," "Album," and "InvoiceLine" show reduced costs and actual times.
* **Execution Time**: Reduced to 0.103 ms.

### Comparison:

* **Execution Time**: There's a significant reduction in execution time after the addition of indexes, indicating improved performance.
* **Planning Time**: Increased from 1.746 ms to 2.672 ms, which is expected due to the optimizer considering the new index paths.
* **Cost and Actual Times**: Both are reduced for most operations, suggesting a more efficient data retrieval process.