

# DataBase programming course

## 2023

Milestone Two of the project (“ Indexing”)

Made by:  
**Mohamed Bassem 2003731**  
**Mina Ehab 2005830**

[9/12/2023]

---

## Query 1

### Index created on

TrackID	B-tree
---------	--------

### index type

### Edits Made

set enable\_seqscan = off

### Index implementation


```
CREATE INDEX btree_trackid ON "InvoiceLine" ("TrackId");
```

### Justification


B-tree indexes are well-suited for equality queries, and in this case, finding rows where "i.TrackId" is equal to t.TrackId is essentially an equality condition. .And index is created on table InvoiceLine on column TrackId to speed up the retrieval of rows that match the condition

	Before	After
Highest Cost	80.03	149
Slowest Run Time	0.486 ms	0.534 ms
Number of Highest Rows	3503	3500

## ScreenShot Before Applying Index

	QUERY PLAN	
	text	
1	Hash Join (cost=142.81..198.02 rows=2240 width=58) (actual time=0.903..2.147 rows=2240 loops=1)	
2	Hash Cond: (al."ArtistId" = ar."ArtistId")	
3	-> Hash Join (cost=134.63..183.86 rows=2240 width=41) (actual time=0.840..1.788 rows=2240 loops=1)	
4	Hash Cond: (t."AlbumId" = al."AlbumId")	
5	-> Hash Join (cost=123.82..167.11 rows=2240 width=41) (actual time=0.773..1.385 rows=2240 loops=1)	
6	Hash Cond: (i."TrackId" = t."TrackId")	
7	-> Seq Scan on "InvoiceLine" i (cost=0.00..37.40 rows=2240 width=21) (actual time=0.006..0.137 rows=2240 lo...	
8	-> Hash (cost=80.03..80.03 rows=3503 width=24) (actual time=0.762..0.762 rows=3503 loops=1)	
9	Buckets: 4096 Batches: 1 Memory Usage: 232kB	
10	-> Seq Scan on "Track" t (cost=0.00..80.03 rows=3503 width=24) (actual time=0.005..0.369 rows=3503 loop...	
11	-> Hash (cost=6.47..6.47 rows=347 width=8) (actual time=0.065..0.065 rows=347 loops=1)	
12	Buckets: 1024 Batches: 1 Memory Usage: 22kB	
13	-> Seq Scan on "Album" al (cost=0.00..6.47 rows=347 width=8) (actual time=0.007..0.032 rows=347 loops=1)	
14	-> Hash (cost=4.75..4.75 rows=275 width=25) (actual time=0.059..0.059 rows=275 loops=1)	
15	Buckets: 1024 Batches: 1 Memory Usage: 24kB	
16	-> Seq Scan on "Artist" ar (cost=0.00..4.75 rows=275 width=25) (actual time=0.011..0.030 rows=275 loops=1)	
17	Planning Time: 1.064 ms	
18	Execution Time: 2.270 ms	

## ScreenShot After Applying Index

	QUERY PLAN	
	text	
1	Hash Join (cost=44.96..365.97 rows=2240 width=58) (actual time=0.154..2.102 rows=2240 loops=1)	
2	Hash Cond: (al."ArtistId" = ar."ArtistId")	
3	-> Hash Join (cost=24.25..339.29 rows=2240 width=41) (actual time=0.083..1.748 rows=2240 loops=1)	
4	Hash Cond: (t."AlbumId" = al."AlbumId")	
5	-> Merge Join (cost=0.56..309.66 rows=2240 width=41) (actual time=0.009..1.377 rows=2240 loops=1)	
6	Merge Cond: (i."TrackId" = t."TrackId")	
7	-> Index Scan using btree_trackid on "InvoiceLine" i (cost=0.28..124.20 rows=2240 width=21) (actual time=0.005..0.465 rows=2240 loop...	
8	-> Index Scan using "PK_Track" on "Track" t (cost=0.28..148.83 rows=3503 width=24) (actual time=0.003..0.378 rows=3500 loops=1)	
9	-> Hash (cost=19.35..19.35 rows=347 width=8) (actual time=0.071..0.071 rows=347 loops=1)	
10	Buckets: 1024 Batches: 1 Memory Usage: 22kB	
11	-> Index Scan using "PK_Album" on "Album" al (cost=0.15..19.35 rows=347 width=8) (actual time=0.003..0.044 rows=347 loops=1)	
12	-> Hash (cost=17.27..17.27 rows=275 width=25) (actual time=0.068..0.068 rows=275 loops=1)	
13	Buckets: 1024 Batches: 1 Memory Usage: 24kB	
14	-> Index Scan using "PK_Artist" on "Artist" ar (cost=0.15..17.27 rows=275 width=25) (actual time=0.004..0.035 rows=275 loops=1)	
15	Planning Time: 2.409 ms	
16	Execution Time: 2.172 ms	

## Conclusion

By enforcing the use of the index, we noticed an increase in runtime (0.486 ms to 0.534ms). and the cost estimate increased from 80.03 to 149, signalling that the chosen index may not be the best fit and this is why I turned the seq\_scan off to make it use my index anyways. These changes highlight a trade-off between cost estimates and actual performance

## Query 2

**Index created on**

**index type**

"InvoiceId"

B+tree

### **Index implementation**

```
CREATE INDEX InvoiceDate_TrackId ON "InvoiceLine" ("InvoiceId")
```

### **Justification**

for Query 2, I added a special index, InvoiceDate\_TrackId, on the "InvoiceLine" table for the "InvoiceId" column. Making searching faster, especially when filtering by the year of the invoice date (2013). This helps improve the performance of the query when counting sales related to tracks in the specified year.

	<b>Before</b>	<b>After</b>
Highest Cost	<b>37.4</b>	<b>12.2</b>
Slowest Run Time	<b>0.844ms</b>	<b>0.442ms</b>
Number of Highest Rows	<b>2240</b>	<b>480</b>

## ScreenShot Before Applying Index

Data Output Notifications Messages

	QUERY PLAN	
	text	🔒
1	Aggregate (cost=59.08..59.09 rows=1 width=8) (actual time=2.940..2.942 rows=1 loops=1)	
2	-> Nested Loop (cost=12.48..59.05 rows=11 width=4) (actual time=1.164..2.825 rows=442 loops=1)	
3	-> Hash Join (cost=12.21..55.54 rows=11 width=4) (actual time=1.151..1.548 rows=442 loops=1)	
4	Hash Cond: (il."InvoiceId" = i."InvoiceId")	
5	-> Seq Scan on "InvoiceLine" il (cost=0.00..37.40 rows=2240 width=8) (actual time=0.026..0.382 rows=2240 loops=1)	
6	-> Hash (cost=12.18..12.18 rows=2 width=4) (actual time=0.401..0.402 rows=80 loops=1)	
7	Buckets: 1024 Batches: 1 Memory Usage: 11kB	
8	-> Seq Scan on "Invoice" i (cost=0.00..12.18 rows=2 width=4) (actual time=0.290..0.358 rows=80 loops=1)	
9	Filter: (EXTRACT(Year FROM "InvoiceDate") = '2013'::numeric)	
10	Rows Removed by Filter: 332	
11	-> Index Only Scan using "PK_Track" on "Track" t (cost=0.28..0.32 rows=1 width=4) (actual time=0.002..0.002 rows=1 loops=...)	
12	Index Cond: ("TrackId" = il."TrackId")	
13	Heap Fetches: 0	
14	Planning Time: 0.546 ms	
15	Execution Time: 2.994 ms	

## ScreenShot After Applying Index

Data Output Notifications Messages

	QUERY PLAN	
	text	🔒
1	Aggregate (cost=32.55..32.56 rows=1 width=8) (actual time=1.237..1.238 rows=1 loops=1)	
2	-> Nested Loop (cost=0.56..32.52 rows=11 width=4) (actual time=0.169..1.197 rows=442 loops=1)	
3	-> Nested Loop (cost=0.28..29.02 rows=11 width=4) (actual time=0.163..0.498 rows=442 loops=1)	
4	-> Seq Scan on "Invoice" i (cost=0.00..12.18 rows=2 width=4) (actual time=0.152..0.192 rows=80 loops=1)	
5	Filter: (EXTRACT(Year FROM "InvoiceDate") = '2013'::numeric)	
6	Rows Removed by Filter: 332	
7	-> Index Scan using "idx_InvoiceDate_TrackId" on "InvoiceLine" il (cost=0.28..8.37 rows=5 width=8) (actual time=0.002..0.003 rows=6 loops=...)	
8	Index Cond: ("InvoiceId" = i."InvoiceId")	
9	-> Index Only Scan using "PK_Track" on "Track" t (cost=0.28..0.32 rows=1 width=4) (actual time=0.001..0.001 rows=1 loops=442)	
10	Index Cond: ("TrackId" = il."TrackId")	
11	Heap Fetches: 0	
12	Planning Time: 0.466 ms	
13	Execution Time: 1.280 ms	

## Conclusion

These numbers collectively demonstrate a substantial enhancement in the query's efficiency and performance after adding the **InvoiceDate\_TrackId** index. The cost of the query went down a lot from 37.4 to 12.2, making it run more efficiently. The time it takes for the query to run also got better, going from 0.844 ms to 0.442 ms, so it's much quicker now. What's even more noticeable is that the number of rows the query has to deal with dropped from 2240 to 480, which means it's handling less data. Overall, the index has made a big difference, especially when I'm looking at sales for tracks in the year 2013.

### Query 3

**Index created on**

**index type**

TrackId

B-tree

### **Edits Made**

set enable\_seqscan = off

### **Index implementation**

```
CREATE INDEX btree_TrackId ON "InvoiceLine" ("TrackId")
```

### **Justification**

B-tree indexes are well-suited for equality queries, and in this case, finding rows where "i.TrackId" is equal to t.TrackId is essentially an equality condition. .And index is created on table InvoiceLine on column TrackId to speed up the retrieval of rows that match the condition

	<b>Before</b>	<b>After</b>
Highest Cost	<b>80.03</b>	<b>149</b>
Slowest Run Time	<b>0.398 ms</b>	<b>0.435 ms</b>
Number of Highest Rows	<b>3503</b>	<b>3501</b>

## ScreenShot Before Applying Index

QUERY PLAN	
	text
1	Limit (cost=215.52..215.53 rows=3 width=29) (actual time=2.217..2.218 rows=3 loops=1)
2	-> Sort (cost=215.52..216.21 rows=275 width=29) (actual time=2.215..2.217 rows=3 loops=1)
3	Sort Key: (count(i."TrackId")) DESC
4	Sort Method: top-N heapsort Memory: 25kB
5	-> HashAggregate (cost=209.22..211.97 rows=275 width=29) (actual time=2.177..2.193 rows=165 loops=1)
6	Group Key: ar."Name"
7	Batches: 1 Memory Usage: 45kB
8	-> Hash Join (cost=142.81..198.02 rows=2240 width=25) (actual time=0.810..1.845 rows=2240 loops=1)
9	Hash Cond: (a."ArtistId" = ar."ArtistId")
10	-> Hash Join (cost=134.63..183.86 rows=2240 width=8) (actual time=0.719..1.494 rows=2240 loops=1)
11	Hash Cond: (t."AlbumId" = a."AlbumId")
12	-> Hash Join (cost=123.82..167.11 rows=2240 width=8) (actual time=0.643..1.153 rows=2240 loops=1)
13	Hash Cond: (i."TrackId" = t."TrackId")
14	-> Seq Scan on "InvoiceLine" i (cost=0.00..37.40 rows=2240 width=4) (actual time=0.006..0.125 rows=2240 lo...
15	-> Hash (cost=80.03..80.03 rows=3503 width=8) (actual time=0.630..0.630 rows=3503 loops=1)
16	Buckets: 4096 Batches: 1 Memory Usage: 169kB
17	-> Seq Scan on "Track" t (cost=0.00..80.03 rows=3503 width=8) (actual time=0.006..0.335 rows=3503 loop...
18	-> Hash (cost=6.47..6.47 rows=347 width=8) (actual time=0.061..0.061 rows=347 loops=1)
19	Buckets: 1024 Batches: 1 Memory Usage: 22kB
20	-> Seq Scan on "Album" a (cost=0.00..6.47 rows=347 width=8) (actual time=0.009..0.034 rows=347 loops=1)
21	-> Hash (cost=4.75..4.75 rows=275 width=25) (actual time=0.065..0.066 rows=275 loops=1)
22	Buckets: 1024 Batches: 1 Memory Usage: 24kB
23	-> Seq Scan on "Artist" ar (cost=0.00..4.75 rows=275 width=25) (actual time=0.013..0.037 rows=275 loops=1)
24	Planning Time: 1.274 ms
25	Execution Time: 2.300 ms

## ScreenShot After Applying Index

	QUERY PLAN text	
1	Limit (cost=325.15..325.16 rows=3 width=29) (actual time=2.075..2.076 rows=3 loops=1)	
2	-> Sort (cost=325.15..325.84 rows=275 width=29) (actual time=2.074..2.075 rows=3 loops=1)	
3	Sort Key: (count(i."TrackId")) DESC	
4	Sort Method: top-N heapsort Memory: 25kB	
5	-> HashAggregate (cost=318.85..321.60 rows=275 width=29) (actual time=2.031..2.048 rows=165 loops=1)	
6	Group Key: ar."Name"	
7	Batches: 1 Memory Usage: 45kB	
8	-> Hash Join (cost=44.96..307.65 rows=2240 width=25) (actual time=0.176..1.699 rows=2240 loops=1)	
9	Hash Cond: (a."ArtistId" = ar."ArtistId")	
10	-> Hash Join (cost=24.25..280.97 rows=2240 width=8) (actual time=0.089..1.355 rows=2240 loops=1)	
11	Hash Cond: (t."AlbumId" = a."AlbumId")	
12	-> Merge Join (cost=0.56..251.33 rows=2240 width=8) (actual time=0.013..1.004 rows=2240 loops=1)	
13	Merge Cond: (t."TrackId" = i."TrackId")	
14	-> Index Scan using "PK_Track" on "Track" t (cost=0.28..148.83 rows=3503 width=8) (actual time=0.004..0.391 rows=3501 loops=1)	
15	-> Index Only Scan using btree_trackid on "InvoiceLine" i (cost=0.28..65.88 rows=2240 width=4) (actual time=0.007..0.178 rows=2240 loops=1)	
16	Heap Fetches: 0	
17	-> Hash (cost=19.35..19.35 rows=347 width=8) (actual time=0.072..0.072 rows=347 loops=1)	
18	Buckets: 1024 Batches: 1 Memory Usage: 22kB	
19	-> Index Scan using "PK_Album" on "Album" a (cost=0.15..19.35 rows=347 width=8) (actual time=0.003..0.044 rows=347 loops=1)	
20	-> Hash (cost=17.27..17.27 rows=275 width=25) (actual time=0.063..0.063 rows=275 loops=1)	
21	Buckets: 1024 Batches: 1 Memory Usage: 24kB	
22	-> Index Scan using "PK_Artist" on "Artist" ar (cost=0.15..17.27 rows=275 width=25) (actual time=0.006..0.036 rows=275 loops=1)	
23	Planning Time: 2.583 ms	
24	Execution Time: 2.162 ms	

## Conclusion

By enforcing the use of the index, we noticed an increase in runtime (0.398 ms to 0.435ms). and the cost estimate increased from 80.03 to 149, signalling that the chosen index may not be the best fit and this is why I turned the seq\_scan off to make it use my index anyways. These changes highlight a trade-off between cost estimates and actual performance



## Query 4

**Index created on**

**index type**

"AlbumId"

B+tree

### **Index implementation**

```
CREATE INDEX Track_AlbumId ON "Track" ("AlbumId");
```

### **Justification**

For Query 4, I added a B+ tree index called "Track\_AlbumId" on the "Track" table for the "AlbumId" column. This index improves the search efficiency, especially when filtering albums by title. It enhances the performance of the query, making it faster when retrieving tracks associated with a particular album title in the "Track," "Album," and "Artist" tables.

	<b>Before</b>	<b>After</b>
Highest Cost	<b>80</b>	<b>37.4</b>
Slowest Run Time	<b>0.358ms</b>	<b>0.4051ms</b>
Number of Highest Rows	<b>3503</b>	<b>2240</b>

## ScreenShot Before Applying Index

Data Output Notifications Messages



	QUERY PLAN	
	text	
1	Hash Join (cost=105.00..150.87 rows=6 width=60) (actual time=0.735..1.021 rows=5 loops=1)	
2	Hash Cond: (a."ArtistId" = ar."ArtistId")	
3	-> Hash Join (cost=96.81..142.67 rows=6 width=43) (actual time=0.656..0.940 rows=5 loops=1)	
4	Hash Cond: (i."TrackId" = t."TrackId")	
5	-> Seq Scan on "InvoiceLine" i (cost=0.00..37.40 rows=2240 width=4) (actual time=0.009..0.123 rows=2240 loops=1)	
6	-> Hash (cost=96.69..96.69 rows=10 width=47) (actual time=0.624..0.626 rows=10 loops=1)	
7	Buckets: 1024 Batches: 1 Memory Usage: 9kB	
8	-> Hash Join (cost=7.35..96.69 rows=10 width=47) (actual time=0.179..0.623 rows=10 loops=1)	
9	Hash Cond: (t."AlbumId" = a."AlbumId")	
10	-> Seq Scan on "Track" t (cost=0.00..80.03 rows=3503 width=24) (actual time=0.008..0.221 rows=3503 loop...	
11	-> Hash (cost=7.34..7.34 rows=1 width=31) (actual time=0.043..0.044 rows=1 loops=1)	
12	Buckets: 1024 Batches: 1 Memory Usage: 9kB	
13	-> Seq Scan on "Album" a (cost=0.00..7.34 rows=1 width=31) (actual time=0.015..0.043 rows=1 loops=1)	
14	Filter: (("Title")::text = 'The Battle Rages On'::text)	
15	Rows Removed by Filter: 346	
16	-> Hash (cost=4.75..4.75 rows=275 width=25) (actual time=0.076..0.076 rows=275 loops=1)	
17	Buckets: 1024 Batches: 1 Memory Usage: 24kB	
18	-> Seq Scan on "Artist" ar (cost=0.00..4.75 rows=275 width=25) (actual time=0.017..0.041 rows=275 loops=1)	
19	Planning Time: 0.472 ms	
20	Execution Time: 1.049 ms	

## ScreenShot After Applying Index

Data Output Messages Notifications



### QUERY PLAN

text

1	Hash Join (cost=22.19..68.05 rows=6 width=60) (actual time=0.276..0.860 rows=5 loops=1)
2	Hash Cond: (i."TrackId" = t."TrackId")
3	-> Seq Scan on "InvoiceLine" i (cost=0.00..37.40 rows=2240 width=4) (actual time=0.025..0.242 rows=2240 loops=1)
4	-> Hash (cost=22.07..22.07 rows=10 width=64) (actual time=0.211..0.213 rows=10 loops=1)
5	Buckets: 1024 Batches: 1 Memory Usage: 9kB
6	-> Nested Loop (cost=7.63..22.07 rows=10 width=64) (actual time=0.147..0.208 rows=10 loops=1)
7	-> Hash Join (cost=7.35..13.14 rows=1 width=48) (actual time=0.138..0.195 rows=1 loops=1)
8	Hash Cond: (ar."ArtistId" = a."ArtistId")
9	-> Seq Scan on "Artist" ar (cost=0.00..4.75 rows=275 width=25) (actual time=0.027..0.053 rows=275 loops=1)
10	-> Hash (cost=7.34..7.34 rows=1 width=31) (actual time=0.094..0.095 rows=1 loops=1)
11	Buckets: 1024 Batches: 1 Memory Usage: 9kB
12	-> Seq Scan on "Album" a (cost=0.00..7.34 rows=1 width=31) (actual time=0.029..0.090 rows=1 loops=1)
13	Filter: (("Title")::text = 'The Battle Rages On'::text)
14	Rows Removed by Filter: 346
15	-> Index Scan using "idx_Track_AlbumId" on "Track" t (cost=0.28..8.83 rows=10 width=24) (actual time=0.006..0.009 rows=10 loop...
16	Index Cond: ("AlbumId" = a."AlbumId")
17	Planning Time: 0.780 ms
18	Execution Time: 0.915 ms

## Conclusion

This index significantly improved overall query performance by decreasing the highest cost from 80 to 37.4, reducing the number of rows processed for the highest cost from 3503 to 2240, and slightly increasing the slowest run time from 0.358 ms to 0.4051 ms. Despite the marginal increase in run time, the substantial reduction in cost and processed rows indicates a favorable trade-off, making the query more resource-efficient after implementing the index.

## Query 5

**Index created on**

**index type**

GenreId

B-tree

### **Index implementation**


```
CREATE INDEX btree_GenreId ON "Track" ("GenreId");
```

### **Justification**

B-tree indexes are well-suited for equality queries, and in this case, finding rows where "t.GenreId" is equal to g.GenreId is essentially an equality condition. .And index is created on table track on column GenreId to speed up the retrieval of rows that match the condition

	<b>Before</b>	<b>After</b>
Highest Cost	<b>80.03</b>	<b>46.8</b>
Slowest Run Time	<b>0.505 ms</b>	<b>0.13 ms</b>
Number of Highest Rows	<b>3503</b>	<b>130</b>

## ScreenShot Before Applying Index

QUERY PLAN		
	text	
1	Nested Loop (cost=13.82..106.43 rows=13 width=555) (actual time=0.051..0.953 rows=130 loops=1)	
2	-> Nested Loop (cost=13.66..105.10 rows=13 width=301) (actual time=0.047..0.900 rows=130 loops=1)	
3	-> Hash Join (cost=13.51..102.88 rows=13 width=282) (actual time=0.042..0.770 rows=130 loops=1)	
4	Hash Cond: (t."GenreId" = g."GenreId")	
5	-> Seq Scan on "Track" t (cost=0.00..80.03 rows=3503 width=28) (actual time=0.010..0.244 rows=3503 loops=1)	
6	-> Hash (cost=13.50..13.50 rows=1 width=262) (actual time=0.020..0.021 rows=1 loops=1)	
7	Buckets: 1024 Batches: 1 Memory Usage: 9kB	
8	-> Seq Scan on "Genre" g (cost=0.00..13.50 rows=1 width=262) (actual time=0.017..0.018 rows=1 loops=1)	
9	Filter: (("Name")::text = 'Jazz')::text)	
10	Rows Removed by Filter: 24	
11	-> Index Scan using "PK_Album" on "Album" al (cost=0.15..0.17 rows=1 width=27) (actual time=0.001..0.001 rows=1 loops=130)	
12	Index Cond: ("AlbumId" = t."AlbumId")	
13	-> Memoize (cost=0.16..0.19 rows=1 width=262) (actual time=0.000..0.000 rows=1 loops=130)	
14	Cache Key: t."MediaTypeId"	
15	Cache Mode: logical	
16	Hits: 128 Misses: 2 Evictions: 0 Overflows: 0 Memory Usage: 1kB	
17	-> Index Scan using "PK_MediaType" on "MediaType" m (cost=0.15..0.18 rows=1 width=262) (actual time=0.002..0.002 rows=1 loop...	
18	Index Cond: ("MediaTypeId" = t."MediaTypeId")	
19	Planning Time: 0.411 ms	
20	Execution Time: 0.989 ms	

## ScreenShot After Applying Index

	QUERY PLAN text	
1	Nested Loop (cost=5.67..70.56 rows=13 width=555) (actual time=0.058..0.215 rows=130 loops=1)	
2	-> Nested Loop (cost=5.51..69.23 rows=13 width=301) (actual time=0.051..0.174 rows=130 loops=1)	
3	-> Nested Loop (cost=5.37..67.02 rows=13 width=282) (actual time=0.044..0.073 rows=130 loops=1)	
4	-> Seq Scan on "Genre" g (cost=0.00..13.50 rows=1 width=262) (actual time=0.009..0.010 rows=1 loops=1)	
5	Filter: ((*Name)::text = 'Jazz'::text)	
6	Rows Removed by Filter: 24	
7	-> Bitmap Heap Scan on "Track" t (cost=5.37..52.12 rows=140 width=28) (actual time=0.030..0.047 rows=130 loops=1)	
8	Recheck Cond: ("GenreId" = g."GenreId")	
9	Heap Blocks: exact=13	
10	-> Bitmap Index Scan on btree_genreid (cost=0.00..5.33 rows=140 width=0) (actual time=0.026..0.026 rows=130 loops=1)	
11	Index Cond: ("GenreId" = g."GenreId")	
12	-> Index Scan using "PK_Album" on "Album" al (cost=0.15..0.17 rows=1 width=27) (actual time=0.001..0.001 rows=1 loops=130)	
13	Index Cond: ("AlbumId" = t."AlbumId")	
14	-> Memoize (cost=0.16..0.19 rows=1 width=262) (actual time=0.000..0.000 rows=1 loops=130)	
15	Cache Key: t."MediaTypeId"	
16	Cache Mode: logical	
17	Hits: 128 Misses: 2 Evictions: 0 Overflows: 0 Memory Usage: 1kB	
18	-> Index Scan using "PK_MediaType" on "MediaType" m (cost=0.15..0.18 rows=1 width=262) (actual time=0.003..0.003 rows=1 loop...)	
19	Index Cond: ("MediaTypeId" = t."MediaTypeId")	
20	Planning Time: 1.548 ms	
21	Execution Time: 0.266 ms	

With the introduction of an index on "GenreId," the optimized query plan showcases substantial improvements. The cost was reduced from 80 to 46.3 , indicating a more efficient query execution strategy. The runtime time significantly decreased from 0.505 ms to 0.13 ms, demonstrating a notable enhancement in query performance after the index implementation. Overall, the indexed approach not only reduced the computational cost but also resulted in a more responsive and resource-efficient query execution

## Query 6

**Index created on**

**index type**

"AlbumId"

B+tree

### **Index implementation**

```
CREATE INDEX Track_AlbumId ON "Track" ("AlbumId");
```

### **Justification**

For Query 6, I implemented a B+ tree index, Track\_AlbumId, on the "Track" table for the "AlbumId" column. This index enhances the efficiency of the LEFT JOIN operation and filtering by artist's name, significantly improving the retrieval of tracks associated with a specific album and overall query performance.

	<b>Before</b>	<b>After</b>
Highest Cost	<b>80</b>	<b>6.47</b>
Slowest Run Time	<b>0.565ms</b>	<b>0.038ms</b>
Number of Highest Rows	<b>3503</b>	<b>347</b>

## ScreenShot Before Applying Index

Data Output Notifications Messages

	QUERY PLAN	
	text	🔒
1	Hash Right Join (cost=12.85..106.12 rows=13 width=47) (actual time=0.881..1.099 rows=9 loops=1)	
2	Hash Cond: (t."AlbumId" = a."AlbumId")	
3	-> Seq Scan on "Track" t (cost=0.00..80.03 rows=3503 width=24) (actual time=0.028..0.311 rows=3503 loops=1)	
4	-> Hash (cost=12.84..12.84 rows=1 width=31) (actual time=0.221..0.223 rows=1 loops=1)	
5	Buckets: 1024 Batches: 1 Memory Usage: 9kB	
6	-> Hash Join (cost=5.45..12.84 rows=1 width=31) (actual time=0.167..0.220 rows=1 loops=1)	
7	Hash Cond: (a."ArtistId" = ar."ArtistId")	
8	-> Seq Scan on "Album" a (cost=0.00..6.47 rows=347 width=31) (actual time=0.029..0.066 rows=347 loops=1)	
9	-> Hash (cost=5.44..5.44 rows=1 width=4) (actual time=0.083..0.084 rows=1 loops=1)	
10	Buckets: 1024 Batches: 1 Memory Usage: 9kB	
11	-> Seq Scan on "Artist" ar (cost=0.00..5.44 rows=1 width=4) (actual time=0.043..0.079 rows=1 loops=1)	
12	Filter: (("Name")::text = 'Pink Floyd')::text	
13	Rows Removed by Filter: 274	
14	Planning Time: 0.769 ms	
15	Execution Time: 1.134 ms	

## ScreenShot After Applying Index

Data Output Notifications Messages

	QUERY PLAN	
	text	🔒
1	Nested Loop Left Join (cost=5.73..13.98 rows=13 width=47) (actual time=0.088..0.114 rows=9 loops=1)	
2	-> Hash Join (cost=5.45..12.84 rows=1 width=31) (actual time=0.082..0.104 rows=1 loops=1)	
3	Hash Cond: (a."ArtistId" = ar."ArtistId")	
4	-> Seq Scan on "Album" a (cost=0.00..6.47 rows=347 width=31) (actual time=0.015..0.038 rows=347 loops=1)	
5	-> Hash (cost=5.44..5.44 rows=1 width=4) (actual time=0.035..0.035 rows=1 loops=1)	
6	Buckets: 1024 Batches: 1 Memory Usage: 9kB	
7	-> Seq Scan on "Artist" ar (cost=0.00..5.44 rows=1 width=4) (actual time=0.020..0.033 rows=1 loops=1)	
8	Filter: (("Name")::text = 'Pink Floyd')::text	
9	Rows Removed by Filter: 274	
10	-> Index Scan using track_albumid on "Track" t (cost=0.28..1.04 rows=10 width=24) (actual time=0.005..0.007 rows=9 loops=1)	
11	Index Cond: ("AlbumId" = a."AlbumId")	
12	Planning Time: 0.356 ms	
13	Execution Time: 0.136 ms	



### Conclusion

In conclusion Comparing the performance before and after adding the index, there were significant improvements: the highest cost decreased from 80 to 6.47, the slowest run time dropped from 0.565 ms to 0.038 ms, and the number of rows processed for the highest cost reduced from 3503 to 347. These results indicate a substantial enhancement in resource efficiency and query speed after the implementation of the index .

### Query 7

Index created on	index type
AlbumId	B-tree

### Index implementation


```
CREATE INDEX btree_AlbumId ON "Track" ("AlbumId");
```

### Justification


B-tree indexes are well-suited for equality queries, and in this case, finding rows where "AlbumId" is equal to (select "AlbumId" from "Album" where "Title" = 'Californication'); is essentially an equality condition. .And index is created on table track on column AlbumId to speed up the retrieval of rows that match the condition

	Before	After
Highest Cost	96.1	16.2
Slowest Run Time	0.368 ms	0.033 ms
Number of Highest Rows	3488	346

### ScreenShot Before Applying Index

	QUERY PLAN	
	text	
1	Seq Scan on "Track" (cost=7.34..96.13 rows=10 width=16) (actual time=0.294..0.368 rows=15 loops=...	
2	Filter: ("AlbumId" = \$0)	
3	Rows Removed by Filter: 3488	
4	InitPlan 1 (returns \$0)	
5	-> Seq Scan on "Album" (cost=0.00..7.34 rows=1 width=4) (actual time=0.020..0.046 rows=1 loops=...	
6	Filter: (("Title")::text = 'Californication'::text)	
7	Rows Removed by Filter: 346	
8	Planning Time: 0.120 ms	
9	Execution Time: 0.407 ms	

### ScreenShot After Applying Index

	QUERY PLAN	
	text	
1	Index Scan using btree_albumid on "Track" (cost=7.62..16.16 rows=10 width=16) (actual time=0.054..0.056 rows=15 loop...	
2	Index Cond: ("AlbumId" = \$0)	
3	InitPlan 1 (returns \$0)	
4	-> Seq Scan on "Album" (cost=0.00..7.34 rows=1 width=4) (actual time=0.024..0.033 rows=1 loops=1)	
5	Filter: (("Title")::text = 'Californication'::text)	
6	Rows Removed by Filter: 346	
7	Planning Time: 1.171 ms	
8	Execution Time: 0.071 ms	

### Conclusion

With the introduction of an index on "AlbumId," the optimized query plan showcases substantial improvements. The cost was reduced from 96.1 to 16.2 , indicating a more efficient query execution strategy. The runtime time significantly decreased from 0.368 ms to 0.033 ms, demonstrating a notable enhancement in query performance after the index implementation. Overall, the indexed approach not only reduced the computational cost but also resulted in a more responsive and resource-efficient query execution

## Query 8

**Index created on**

**index type**

"ArtistId"

B+tree

### **Edits Made**

set enable\_seqscan = off

### **Index implementation**

```
CREATE INDEX Album_ArtistId ON "Album" ("ArtistId");
```

### **Justification**

For Query 8, I used the B+ tree index "idx\_Album\_ArtistId" on the "Album" table. B+ tree index type is suitable as it aligns with my WHERE clause filtering artists who do not have albums. By optimizing the search for artists without associated albums, the chosen B+ tree index significantly contributes to the improved performance of my query.

	<b>Before</b>	<b>After</b>
Highest Cost	<b>6.47</b>	<b>1000000000</b>
Slowest Run Time	<b>0.309ms</b>	<b>0.264ms</b>
Number of Highest Rows	<b>347</b>	<b>347</b>

## ScreenShot Before Applying Index

Data Output Notifications Messages

	QUERY PLAN	
	text	
1	Seq Scan on "Artist" ar (cost=7.34..12.78 rows=138 width=25) (actual time=0.467..0.565 rows=71 loops=1)	
2	Filter: (NOT (hashed SubPlan 1))	
3	Rows Removed by Filter: 204	
4	SubPlan 1	
5	-> Seq Scan on "Album" (cost=0.00..6.47 rows=347 width=4) (actual time=0.015..0.256 rows=347 loops=1)	
6	Planning Time: 0.201 ms	
7	Execution Time: 0.603 ms	

## ScreenShot After Applying Index

Data Output Messages Notifications

	QUERY PLAN	
	text	
1	Seq Scan on "Artist" ar (cost=10000000021.08..10000000026.52 rows=138 width=25) (actual time=0.301..0.408 rows=71 loops=1)	
2	Filter: (NOT (hashed SubPlan 1))	
3	Rows Removed by Filter: 204	
4	SubPlan 1	
5	-> Index Only Scan using album_artistid on "Album" (cost=0.15..20.21 rows=347 width=4) (actual time=0.020..0.144 rows=347 loops=1)	
6	Heap Fetches: 347	
7	Planning Time: 0.151 ms	
8	Execution Time: 0.438 ms	

## Conclusion

By enforcing the use of the index, we achieved improvement in runtime (0.309 ms to 0.264 ms). However, the cost estimate increased from 6.47 to 1000000000, signalling that the chosen index may not be the best fit and this is why I turned the seq\_scan off to make it use my index anyways. These changes highlight a trade-off between cost estimates and actual performance. I assume the high cost is because this query makes postgres Scan more than 50% so index wouldn't be efficient.

## Query 9

### Index created on

### index type

PlaylistId

B-tree

### Edits Made

set enable\_seqscan = off

### Index implementation


CREATE INDEX btree\_PlaylistId ON "PlaylistTrack" ("PlaylistId")

### Justification


B-tree is suitable for equality conditions. Having an index on table PlaylistTrack on column PlaylistId can speed up the join operations.

	Before	After
Highest Cost	126	171.01
Slowest Run Time	1.21 ms	1.18 ms
Number of Highest Rows	8715	8715

### ScreenShot Before Applying Index

QUERY PLAN		
	text	
1	HashAggregate (cost=199.39..199.57 rows=18 width=266) (actual time=2.734..2.736 rows=12 loops=1)	
2	Group Key: p."Name"	
3	Batches: 1 Memory Usage: 24kB	
4	-> Hash Left Join (cost=1.41..155.82 rows=8715 width=262) (actual time=0.036..1.610 rows=8715 loops=1)	
5	Hash Cond: (pt."PlaylistId" = p."PlaylistId")	
6	-> Seq Scan on "PlaylistTrack" pt (cost=0.00..126.15 rows=8715 width=4) (actual time=0.013..0.382 rows=8715 loop...	
7	-> Hash (cost=1.18..1.18 rows=18 width=262) (actual time=0.014..0.015 rows=18 loops=1)	
8	Buckets: 1024 Batches: 1 Memory Usage: 9kB	
9	-> Seq Scan on "Playlist" p (cost=0.00..1.18 rows=18 width=262) (actual time=0.008..0.009 rows=18 loops=1)	
10	Planning Time: 0.151 ms	
11	Execution Time: 2.773 ms	

## ScreenShot After Applying Index

QUERY PLAN		
	text	
1	HashAggregate (cost=255.48..255.66 rows=18 width=266) (actual time=3.407..3.409 rows=12 loops=1)	
2	Group Key: p."Name"	
3	Batches: 1 Memory Usage: 24kB	
4	-> Hash Left Join (cost=12.92..211.90 rows=8715 width=262) (actual time=0.651..2.242 rows=8715 loops=1)	
5	Hash Cond: (pt."PlaylistId" = p."PlaylistId")	
6	-> Index Only Scan using btree_playlistid on "PlaylistTrack" pt (cost=0.29..171.01 rows=8715 width=4) (actual time=0.621..1.181 rows=8715 loop...	
7	Heap Fetches: 0	
8	-> Hash (cost=12.41..12.41 rows=18 width=262) (actual time=0.024..0.025 rows=18 loops=1)	
9	Buckets: 1024 Batches: 1 Memory Usage: 9kB	
10	-> Index Scan using "PK_Playlist" on "Playlist" p (cost=0.14..12.41 rows=18 width=262) (actual time=0.015..0.017 rows=18 loops=1)	
11	Planning Time: 0.160 ms	
12	Execution Time: 3.438 ms	

## Conclusion

By enforcing the use of the index, we achieved improvement in runtime (1.21 ms to 1.18 ms). However, the cost estimate increased from 126 to 171.01, signalling that the chosen index may not be the best fit and this is why I turned the seq\_scan off to make it use my index anyways. These changes highlight a trade-off between cost estimates and actual performance.

## Query 10

**Index created on**

**index type**

"TrackId"

B+tree

### **Edits Made**

set enable\_seqscan = off

### **Index implementation**

```
CREATE INDEX InvoiceLine_TrackId ON "InvoiceLine"("TrackId");
```

### **Justification**

I applied the B+ tree index InvoiceLine\_TrackId on the "InvoiceLine" table, focusing on the "TrackId" column, for Query 10. This type of B+ tree index is beneficial as it aligns with my JOIN operation on "TrackId" and the grouping and counting of tracks by name. The B+ tree index significantly enhances the efficiency of searching for tracks within invoice lines, contributing to the improved performance of my query.

	<b>Before</b>	<b>After</b>
Highest Cost	<b>130</b>	<b>149</b>
Slowest Run Time	<b>0.975ms</b>	<b>0.537ms</b>
Number of Highest Rows	<b>3503</b>	<b>3501</b>

## ScreenShot Before Applying Index

Data Output	Notifications	Messages
<div> </div>		
	<b>QUERY PLAN</b> text	
1	Sort (cost=325.36..330.96 rows=2240 width=24) (actual time=6.871..7.022 rows=1888 loops=1)	
2	Sort Key: (count(t."Name")) DESC	
3	Sort Method: quicksort Memory: 175kB	
4	-> HashAggregate (cost=178.31..200.71 rows=2240 width=24) (actual time=5.655..6.302 rows=1888 loops=1)	
5	Group Key: t."Name"	
6	Batches: 1 Memory Usage: 369kB	
7	-> Hash Join (cost=123.82..167.11 rows=2240 width=16) (actual time=2.035..3.847 rows=2240 loops=1)	
8	Hash Cond: (l."TrackId" = t."TrackId")	
9	-> Seq Scan on "InvoiceLine" l (cost=0.00..37.40 rows=2240 width=4) (actual time=0.018..0.331 rows=2240 loops=1)	
10	-> Hash (cost=80.03..80.03 rows=3503 width=20) (actual time=1.991..1.992 rows=3503 loops=1)	
11	Buckets: 4096 Batches: 1 Memory Usage: 219kB	
12	-> Seq Scan on "Track" t (cost=0.00..80.03 rows=3503 width=20) (actual time=0.020..0.975 rows=3503 loops=1)	
13	Planning Time: 0.403 ms	
14	Execution Time: 7.390 ms	

## ScreenShot After Applying Index

	<b>QUERY PLAN</b> text	
1	Sort (cost=409.58..415.18 rows=2240 width=24) (actual time=2.302..2.372 rows=1888 loops=1)	
2	Sort Key: (count(t."Name")) DESC	
3	Sort Method: quicksort Memory: 175kB	
4	-> HashAggregate (cost=262.53..284.93 rows=2240 width=24) (actual time=1.802..2.054 rows=1888 loops=1)	
5	Group Key: t."Name"	
6	Batches: 1 Memory Usage: 369kB	
7	-> Merge Join (cost=0.56..251.33 rows=2240 width=16) (actual time=0.016..1.248 rows=2240 loops=1)	
8	Merge Cond: (t."TrackId" = l."TrackId")	
9	-> Index Scan using "PK_Track" on "Track" t (cost=0.28..148.83 rows=3503 width=20) (actual time=0.007..0.516 rows=3501 loops=1)	
10	-> Index Only Scan using invoice_line_trackid on "InvoiceLine" l (cost=0.28..65.88 rows=2240 width=4) (actual time=0.007..0.195 rows=2240 loops=1)	
11	Heap Fetches: 0	
12	Planning Time: 0.246 ms	
13	Execution Time: 2.453 ms	

## Conclusion

By enforcing the use of the index, we achieved improvement in runtime (0.975 ms to 0.537 ms). However, the cost estimate increased from 130 to 149, signalling that the chosen index may not be the best fit and this is why I turned the seq\_scan off to make it use my index anyways. These changes highlight a trade-off between cost estimates and actual performance. I assume that the default index is the best index for this query.



## Query 11

### Index created on

### index type

TrackId

B-tree

### Edits Made

set enable\_seqscan = off

### Index implementation

```
CREATE INDEX btree_TrackId ON "InvoiceLine" ("TrackId")
```

### Justification


B-tree is suitable for equality conditions. Having an index on table InvoiceLine on column TrackId can speed up the join operations.

	Before	After
Highest Cost	80	134
Slowest Run Time	0.297 ms	0.472 ms
Number of Highest Rows	3503	2240

## ScreenShot Before Applying Index

	QUERY PLAN	
	text	
1	HashAggregate (cost=157.69..161.06 rows=337 width=24) (actual time=0.926..0.954 rows=330 loops=1)	
2	Group Key: t."Name"	
3	Batches: 1 Memory Usage: 61kB	
4	-> Hash Join (cost=59.47..156.01 rows=337 width=16) (actual time=0.416..0.857 rows=340 loops=1)	
5	Hash Cond: (t."TrackId" = i."TrackId")	
6	-> Seq Scan on "Track" t (cost=0.00..80.03 rows=3503 width=20) (actual time=0.008..0.155 rows=3503 loops=1)	
7	-> Hash (cost=55.26..55.26 rows=337 width=4) (actual time=0.404..0.405 rows=340 loops=1)	
8	Buckets: 1024 Batches: 1 Memory Usage: 20kB	
9	-> Hash Join (cost=11.93..55.26 rows=337 width=4) (actual time=0.087..0.377 rows=340 loops=1)	
10	Hash Cond: (i."InvoiceId" = i."InvoiceId")	
11	-> Seq Scan on "InvoiceLine" i (cost=0.00..37.40 rows=2240 width=8) (actual time=0.005..0.108 rows=2240 loop...	
12	-> Hash (cost=11.15..11.15 rows=62 width=4) (actual time=0.078..0.078 rows=62 loops=1)	
13	Buckets: 1024 Batches: 1 Memory Usage: 11kB	
14	-> Seq Scan on "Invoice" i (cost=0.00..11.15 rows=62 width=4) (actual time=0.007..0.069 rows=62 loops=1)	
15	Filter: (("BillingCity")::text ~~ 'B%':::text)	
16	Rows Removed by Filter: 350	
17	Planning Time: 0.350 ms	
18	Execution Time: 0.990 ms	

## ScreenShot After Applying Index

	QUERY PLAN	
	text	
1	HashAggregate (cost=299.64..303.01 rows=337 width=24) (actual time=1.169..1.199 rows=330 loops=1)	
2	Group Key: t."Name"	
3	Batches: 1 Memory Usage: 61kB	
4	-> Nested Loop (cost=33.82..297.96 rows=337 width=16) (actual time=0.098..1.092 rows=340 loops=1)	
5	-> Hash Join (cost=33.54..163.40 rows=337 width=4) (actual time=0.093..0.731 rows=340 loops=1)	
6	Hash Cond: (i."InvoiceId" = i."InvoiceId")	
7	-> Index Scan using btree_trackid on "InvoiceLine" i (cost=0.28..124.20 rows=2240 width=8) (actual time=0.011..0.472 rows=2240 loop...	
8	-> Hash (cost=32.48..32.48 rows=62 width=4) (actual time=0.076..0.076 rows=62 loops=1)	
9	Buckets: 1024 Batches: 1 Memory Usage: 11kB	
10	-> Index Scan using "PK_Invoice" on "Invoice" i (cost=0.27..32.48 rows=62 width=4) (actual time=0.005..0.070 rows=62 loops=1)	
11	Filter: (("BillingCity")::text ~~ 'B%':::text)	
12	Rows Removed by Filter: 350	
13	-> Index Scan using "PK_Track" on "Track" t (cost=0.28..0.40 rows=1 width=20) (actual time=0.001..0.001 rows=1 loops=340)	
14	Index Cond: (t."TrackId" = i."TrackId")	
15	Planning Time: 0.284 ms	
16	Execution Time: 1.239 ms	

## Conclusion

By enforcing the use of the index, we noticed a increase in runtime (0.297 ms to 0.472ms). and the cost estimate increased from 80 to 134, signalling that the chosen index may not be the best fit and this is why I turned the seq\_scan off to make it use my index anyways. These changes highlight a trade-off between cost estimates and actual performance

## Query 12

**Index created on**

**index type**

"AlbumId"

B+tree

### **Index implementation**

```
CREATE INDEX Track_AlbumId ON "Track" ("AlbumId");
```

### **Justification**

For Query 12, I added the B+ tree index "Track\_AlbumId" on the "Track" table for the "AlbumId" column. This helps speed up finding details like invoice lines, track info, album details, and artist info. The index makes searching based on "AlbumId" more efficient, improving the overall performance of my query.

	<b>Before</b>	<b>After</b>
Highest Cost	<b>80</b>	<b>37.4</b>
Slowest Run Time	<b>0.348ms</b>	<b>0.2ms</b>
Number of Highest Rows	<b>3503</b>	<b>347</b>

## ScreenShot Before Applying Index

Data Output Notifications Messages

<div> <div>≡+</div> <div>📄</div> <div>▼</div> <div>📋</div> <div>▼</div> <div>🗑️</div> <div>🗄️</div> <div>⬇️</div> <div>📈</div> </div>	
	<div> <div>QUERY PLAN</div> <div>text</div> <div>🔒</div> </div>
1	Hash Join (cost=106.28..152.16 rows=8 width=37) (actual time=0.732..1.042 rows=5 loops=1)
2	Hash Cond: (i."TrackId" = t."TrackId")
3	-> Seq Scan on "InvoiceLine" i (cost=0.00..37.40 rows=2240 width=21) (actual time=0.030..0.142 rows=2240 loops=...
4	-> Hash (cost=106.12..106.12 rows=13 width=20) (actual time=0.695..0.697 rows=4 loops=1)
5	Buckets: 1024 Batches: 1 Memory Usage: 9kB
6	-> Hash Join (cost=12.85..106.12 rows=13 width=20) (actual time=0.140..0.695 rows=4 loops=1)
7	Hash Cond: (t."AlbumId" = al."AlbumId")
8	-> Seq Scan on "Track" t (cost=0.00..80.03 rows=3503 width=24) (actual time=0.014..0.224 rows=3503 loops=...
9	-> Hash (cost=12.84..12.84 rows=1 width=4) (actual time=0.122..0.123 rows=2 loops=1)
10	Buckets: 1024 Batches: 1 Memory Usage: 9kB
11	-> Hash Join (cost=5.45..12.84 rows=1 width=4) (actual time=0.072..0.122 rows=2 loops=1)
12	Hash Cond: (al."ArtistId" = a."ArtistId")
13	-> Seq Scan on "Album" al (cost=0.00..6.47 rows=347 width=8) (actual time=0.017..0.040 rows=347 lo...
14	-> Hash (cost=5.44..5.44 rows=1 width=4) (actual time=0.051..0.052 rows=1 loops=1)
15	Buckets: 1024 Batches: 1 Memory Usage: 9kB
16	-> Seq Scan on "Artist" a (cost=0.00..5.44 rows=1 width=4) (actual time=0.023..0.048 rows=1 loop...
17	Filter: (("Name")::text = 'Accept'::text)
18	Rows Removed by Filter: 274
19	Planning Time: 1.076 ms
20	Execution Time: 1.079 ms

## ScreenShot After Applying Index

Data Output Notifications Messages



	QUERY PLAN	
	text	
1	Hash Join (cost=14.15..60.03 rows=8 width=37) (actual time=0.121..0.435 rows=5 loops=1)	
2	Hash Cond: (i."TrackId" = t."TrackId")	
3	-> Seq Scan on "InvoiceLine" i (cost=0.00..37.40 rows=2240 width=21) (actual time=0.015..0.132 rows=2240 loops=1)	
4	-> Hash (cost=13.98..13.98 rows=13 width=20) (actual time=0.102..0.103 rows=4 loops=1)	
5	Buckets: 1024 Batches: 1 Memory Usage: 9kB	
6	-> Nested Loop (cost=5.73..13.98 rows=13 width=20) (actual time=0.052..0.101 rows=4 loops=1)	
7	-> Hash Join (cost=5.45..12.84 rows=1 width=4) (actual time=0.048..0.093 rows=2 loops=1)	
8	Hash Cond: (al."ArtistId" = a."ArtistId")	
9	-> Seq Scan on "Album" al (cost=0.00..6.47 rows=347 width=8) (actual time=0.010..0.029 rows=347 loops=1)	
10	-> Hash (cost=5.44..5.44 rows=1 width=4) (actual time=0.034..0.034 rows=1 loops=1)	
11	Buckets: 1024 Batches: 1 Memory Usage: 9kB	
12	-> Seq Scan on "Artist" a (cost=0.00..5.44 rows=1 width=4) (actual time=0.012..0.032 rows=1 loops=1)	
13	Filter: (("Name")::text = 'Accept'::text)	
14	Rows Removed by Filter: 274	
15	-> Index Scan using track_albumid on "Track" t (cost=0.28..1.04 rows=10 width=24) (actual time=0.002..0.003 rows=2 loop...)	
16	Index Cond: ("AlbumId" = al."AlbumId")	
17	Planning Time: 0.476 ms	
18	Execution Time: 0.461 ms	

## Conclusion

In conclusion Comparing the metrics before and after index implementation, notable improvements were observed: the highest cost decreased from 80 to 37.4, the slowest run time dropped from 0.348 ms to 0.2 ms, and the number of rows processed for the highest cost reduced from 3503 to 347. These results underscore the positive impact of the index on resource efficiency and query speed for Query 12.