GERMAN INTERNATIONAL UNIVERSITY
الجامعة الألمانية الدولية

DataBase programming course

2023

Milestone one of the project (“ Indexing”)

Made by:
Mohamed Bassem 2003731
Mina Ehab 2005830

[14/11/2023]

Query 1

Index created on	index type
Composer	B+tree

Index Usage

Scan = 1	Read = 978	Fetch = 0

Index implementation

```
Create index composer_Btree on "Track"("Composer");
```

Justification

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

Query plan Time	Query execution Time
6996.934496999756	250157.12619600317

Latency	TPS
Before = 0.969 ms	Before = 1032.317338
After = 0.800 ms	After = 1249.397877

Conclusion

B-tree index on the "Composer" column has significantly improved the performance of the query by increasing the TPS and reducing the average latency

Query 2

Index created on	index type
Composer	B-tree

Index Usage

Scan = 0	Read = 0	Fetch = 0

Index implementation

```
CREATE INDEX Composer_btree ON "Track"("Composer");
```

Justification

The Index on the Composer column did not work. I tried many other columns and it did not work. The best index for this query that it always uses is its primary key and no other column works to be an index for this query.I also tried different indexing methods but still didn’t work.

Query plan Time	Query execution Time
6749.7116190000515	609971.7622899954

Latency	TPS
Before = 2.247 ms	Before = 445.038683
After = 2.202 ms	After = 440.389110

Conclusion

The values of Latency and Tps is not changed as the index I created is not used and in the after and before results the index that was used is the primary key of the table

Query 3

Index created on	index type
Composer	B+tree

Index Usage

Scan = 1	Read = 8	Fetch = 0

Index implementation

```
CREATE INDEX Composer_btree ON "Track"("Composer")
```

Justification

The B-tree index on the "Composer" column is effective for the query's filtering condition (WHERE t."Composer" = 'AC/DC'). The index allows for efficient lookups based on equality conditions.And index is created on table track on column Composer to speed up the retrieval of rows that match the condition

Query plan Time	Query execution Time
12291.082295000024	75586.67088399908

Latency	TPS
Before = 1.279 ms	Before = 781.711950
After = 0.987ms	After = 1013.041451

Conclusion

B-tree index on the "Composer" column has significantly improved the performance of the query by increasing the TPS and reducing the average latency

Query 4

Index created on	index type
Composer	Hash index

Index Usage

Scan = 1	Read = 8	Fetch = 0

Index implementation

```
CREATE INDEX Composer_hash ON "Track" using hash("Composer")
```

Justification

Hash index on the "Composer" column is effective for the query's filtering condition (WHERE t."Composer" = 'AC/DC'). The index allows for efficient lookups based on equality conditions..And index is created on table track on column Composer to speed up the retrieval of rows that match the condition

Query plan Time	Query execution Time
13347.085638999974	78512.6090140009

Latency	TPS
Before = 1.378	Before = 725.757526
After = 1.245	After = 803.320963

Conclusion

Hash index on the "Composer" column has significantly improved the performance of the query by increasing the TPS and reducing the average latency

Query 5

Index created on	index type
UnitPrice	Partial index

Index Usage

Scan = 1	Read = 213	Fetch = 213

Index implementation

```
CREATE INDEX UNitPrice_partial_index ON "Track" ("UnitPrice") WHERE "UnitPrice" != 0.99;
```

Justification

The partial index on "UnitPrice" is designed to cover the specific condition in the query (WHERE "UnitPrice" != 0.99). This allows the database to efficiently locate and retrieve rows where the "UnitPrice" is not equal to 0.99. And index is created on table track on column UnitPrice to speed up the retrieval of rows that match the condition

Query plan Time	Query execution Time
1712.0258639999865	57190.35292300116

Latency	TPS
Before = 0.940 ms	Before = 1064.128705
After = 0.653 ms	After = 1532.404431

Conclusion

The partial index on the "UnitPrice" column has significantly improved the performance of the query by increasing the TPS and reducing the average latency

Query 6

Index created on	index type
UnitPrice	Partial index

Index Usage

Scan = 0	Read = 0	Fetch = 0

Index implementation

```
CREATE INDEX UNitPrice_partial_index ON "Track" ("UnitPrice") WHERE "UnitPrice" = 0.99;
```

Justification

The Index on the UnitPrice column did not work. I tried many other columns and it did not work. The best index for this query that it always uses is its primary key and no other column works to be an index for this query.I also tried different indexing methods but still didn’t work.

Query plan Time	Query execution Time
131842.88444800072	326208.58733399806

Latency	TPS
Before = 2.694 ms	Before = 371.263929
After = 2.630 ms	After = 365.467290

Conclusion

The values of Latency and Tps is not changed as the index I created is not used and in the after and before results the index that was used is the primary key of the table

Query 7

Index created on	index type
AlbumId	B+tree

Index Usage

Scan = 4	Read = 4	Fetch = 0

Index implementation

```
CREATE INDEX Q7_AlbumId ON "Track" ("AlbumId");
```

Justification

The B+tree index type was chosen for the "AlbumId" index in the "Track" table. B+tree is well-suited for range queries and provides efficient traversal of data in a sorted order, which aligns with the typical access patterns for querying data by AlbumId.Having an index on table Track on column AlbumId can speed up the join operations.

Query plan Time	Query execution Time
132324.90745900074	327274.50591099774

Latency	TPS
Before =7.768 ms	Before = 128.727539
After = 7.326	After = 136.495015

Conclusion

The marginal decrease in latency and the increase TPS suggest that the index implementation improved the efficiency of queries involving the "AlbumId" column, contributing to a more responsive system.

Query 8

Index created on	index type
Artistid	B+ tree

Index Usage

Scan = 2	Read = 2	Fetch = 2

Index implementation

```
-- CREATE INDEX Q8_artistid ON "Album" ("ArtistId");
```

Justification

I went with a B+tree index for the "ArtistId" column in my "Album" table because it suits my query. It's perfect for sorting and quickly finding data within a range, which is crucial for joins and conditions on "ArtistId." This choice makes my query more efficient, speeding up data retrieval based on specific conditions.

Query plan Time	Query execution Time
442826.9468029906	1393653.2967069934

Latency	TPS
Before = 0.819 ms	Before = 1221.389305
After = 0.461 ms	After = 2170.206100

Conclusion

In summary, the utilisation of a B+tree index on the "ArtistId" column in the "Album" table resulted in a notable decrease in query execution latency and a substantial increase in TPS. This suggests that the index implementation significantly improved the efficiency of queries related to "ArtistId," leading to a more responsive and performant system.

Query 9

Index created on	index type
AlbumId	B+ tree

Index Usage

Scan = 6	Read = 8	Fetch = 4

Index implementation

```
CREATE INDEX  Q9_AlbumId  ON "Track" ("AlbumId");
```

Justification

I decided to use a B+tree index for the "AlbumId" column in my "Track" table to boost query performance. B+tree indexes are great for range queries, which fits well with how I usually access data when identifying albums. This choice is about optimizing the way I query data based on album identification, making it more efficient.

Query plan Time	Query execution Time
55329.78263099972	150704.62613399967

Latency	TPS
Before = 1.835 ms	Before = 545.049249
After = 1.552 ms	After = 700.267436

Conclusion

In summary, the implementation of the B+tree index on the "AlbumId" column in the "Track" table resulted in a reduced query execution latency and an increase in throughput. This suggests that the index optimization positively impacted the efficiency of queries involving the "AlbumId" column, contributing to overall improved system performance.

Query 10

Index created on	index type
ArtistId	B+tree

Index Usage

Scan = 2	Read = 2	Fetch = 2

Index implementation

```
-- CREATE INDEX  Q10_Artist  ON "Album" ("ArtistId")
```

Justification

I added a B+tree index to the "ArtistId" column in my "Album" table to improve query performance. B+tree indexes excel at handling range queries, which is exactly what I need for queries related to artist identification. Having an index on table Album on column ArtistId can speed up the join operations.

Query plan Time	Query execution Time
57166.80287899973	154677.46051399963

Latency	TPS
Before = 6.055 ms	Before = 165.154535
After = 5.573 ms	After = 179.443129

Conclusion

B-tree index on the "ArtistId" column has significantly improved the performance of the query by increasing the TPS and reducing the average latency.

Query 11

Index created on	index type
TrackId	B+tree

Index Usage

Scan = 52	Read = 149	Fetch = 145

Index implementation

```
CREATE INDEX PlayListTrack_btree  ON "PlaylistTrack"("TrackId")
```

Justification

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

Query plan Time	Query execution Time
58969.88314600196	294841.7196140023

Latency	TPS
Before = 1.608 ms	Before = 621.861520
After = 1.044 ms	After = 957.952080

Conclusion

B-tree index on the "TrackId" column has significantly improved the performance of the query by increasing the TPS and reducing the average latency

Query 12

Index created on	index type
BillingCountry	B+tree

Index Usage

Scan = 1	Read = 35	Fetch =0

Index implementation

```
CREATE INDEX  BillingCountry_btree ON "Invoice"("BillingCountry")
```

Justification

B-tree index on the " BillingCountry " column is effective for the query's filtering condition (where "BillingCountry" = 'Brazil'). The index allows for efficient lookups based on equality conditions. nd index is created on table Invoice on column BillingCountry to speed up the retrieval of rows that match the condition

Query plan Time	Query execution Time
5142.190808000033	6977.268794000108

Latency	TPS
Before = 0.424 ms	Before = 2358.554231
After = 0.303 ms	After = 3304.933925

Conclusion

B+tree index on the " BillingCountry " column has significantly improved the performance of the query by increasing the TPS and reducing the average latency

Query 13

Index created on	index type
Country	B-tree

Index Usage

Scan = 0	Read = 0	Fetch = 0

Index implementation

```
CREATE INDEX  Country_btree ON "Customer"("Country")
```

Justification

The Index on the Country column did not work. I tried many other columns and it did not work. The best index for this query that it always uses is its primary key and no other column works to be an index for this query.I also tried different indexing methods but still didn’t work.

Query plan Time	Query execution Time
1662.9098509999894	1459.2451340000346

Latency	TPS
Before = 0.530 ms	Before = 1886.852231
After = 0.510 ms	After = 1876.557902

Conclusion

The values of Latency and Tps is not changed as the index I created is not used and in the after and before results the index that was used is the primary key of the table

Query 14

Index created on	index type
SupportRepId	B-tree

Index Usage

Scan = 0	Read = 0	Fetch = 0

Index implementation

```
CREATE INDEX customerId_btree ON "Customer"  ("SupportRepId");
```

Justification

The Index on the SupportRepId column did not work. I tried many other columns and it did not work. The best index for this query that it always uses is its primary key and no other column works to be an index for this query.I also tried different indexing methods but still didn’t work.

Query plan Time	Query execution Time
9826.950984999976	6479.058575000022

Latency	TPS
Before = 0.308 ms	Before = 3241.903061
After = 0.315 ms	After = 3244.807140

Conclusion

The values of Latency and Tps is not changed as the index I created is not used and in the after and before results the index that was used is the primary key of the tables

Query 15

Index created on	index type
BillingCity	B-tree

Index Usage

Scan = 0	Read = 0	Fetch = 0

Index implementation

Create Index BillingCity_btree on “Invoice” (“BillingCity ”)

Justification

The Index on the BillingCity column did not work. I tried many other columns and it did not work. The best index for this query that it always uses is its primary key and no other column works to be an index for this query.I also tried different indexing methods but still didn’t work.

Query plan Time	Query execution Time
132999.68133400058	328611.7562689977

Latency	TPS
Before = 0.682 ms	Before = 1466.002395
After = 0.641	After = 1456.231735

Conclusion

The values of Latency and Tps is not changed as the index I created is not used and in the after and before results the index that was used is the primary key of the table

Query 16

Index created on	index type
Total	B+tree

Index Usage

Scan = 1	Read = 62	Fetch = 62

Index implementation

```
-- CREATE INDEX  Q16_total  ON "Invoice" ("Total");
```

Justification

I chose to use a B+tree index on the "Total" column in the "Invoice" table to enhance query performance, especially for operations involving total amounts. B+tree indexes are ideal for range queries and efficiently navigating sorted numerical data.And index is created on table Invoice on column Total to speed up the retrieval of rows that match the condition.

Query plan Time	Query execution Time
57693.12498899976	155905.4141099997

Latency	TPS
Before = 0.722 ms	Before = 1384.303326
After = 0.347 ms	After = 2885.865868

Conclusion

In summary, the implementation of the B+tree index on the "Total" column in the "Invoice" table resulted in a substantial reduction in query execution latency and a significant increase in TPS.

Query 17

Index created on	index type
("InvoiceId", "TrackId","UnitPrice", "Quantity")	B+tree

Index Usage

Scan = 2	Read =2	Fetch = 0

Index implementation

```
-- CREATE INDEX Q17_Track ON "InvoiceLine" ("InvoiceId", "TrackId") INCLUDE ("UnitPrice", "Quantity");
```

Justification

I created a composite index on ("InvoiceId", "TrackId") in the "InvoiceLine" table, including ("UnitPrice", "Quantity") as additional columns. This choice aims to optimize query performance. Composite indexes are valuable for addressing multiple columns in queries, and including extra columns improves the index's capacity to fulfill query needs without requiring additional lookups.

Query plan Time	Query execution Time
58476.09782999976	157562.39142599975

Latency	TPS
Before = 1.474 ms	Before = 678.271825
After = 1.127 ms	After = 887.199125

Conclusion

In summary, the creation of the composite index on ("InvoiceId", "TrackId") with included columns ("UnitPrice", "Quantity") in the "InvoiceLine" table resulted in a notable reduction in query execution latency and an increase in TPS.

Query 18

Index created on	index type
InvoiceId	B+tree

Index Usage

Scan = 9	Read = 40	Fetch = 38

Index implementation

```
-- CREATE INDEX Q18_InvoiceId ON "InvoiceLine"("InvoiceId");
```

Justification

I added a B+tree index to the "InvoiceId" column in the "InvoiceLine" table to boost query performance, especially when filtering or joining based on the invoice identifier.

Query plan Time	Query execution Time
59216.41521299976	159395.2489249999

Latency	TPS
Before = 0.591 ms	Before = 1693.470540
After = 0.466 ms	After = 2144.660304

Conclusion

In summary, the creation of the B+tree index on the "InvoiceId" column in the "InvoiceLine" table resulted in a notable reduction in query execution latency and an increase in TPS.