COMP4380 Project Phase 3

Xiaoran Meng[1], Taeho Choi[2], Tanisha Turner[3], Sukhmandeep Kaur[4], Jasmine Tabuzo[5]

Department of Computer Science
University of Manitoba
Winnipeg, MB, R3T 2N2, Canada

[1]mengx5@myumanitoba.ca

[2]choit1@myumanitoba.ca

[3]turnert1@myumanitoba.ca

[4]kaurs45@myumanitoba.ca

[5]tabuzoj@myumanitoba.ca

***Abstract*— In this paper, we present the analysis we completed on several queries that use the City of Chicago's taxi trip data. We used optimization strategies discussed in the course materials as well as some from our previous knowledge. The following paper will illustrate the results of our experiments and include comparisons of the query times before and after our optimizations as well as an analysis of the optimizations that we applied. Furthermore, we discuss the optimization techniques that did not significantly affect our results, and consider potential improvements to our queries for the future.**

***Keywords*— City of Chicago, Optimization, Taxi Trips, Hash Index, B+ Tree Index, Conjunctive Normal Form, Database, Queries**

I. INTRODUCTION

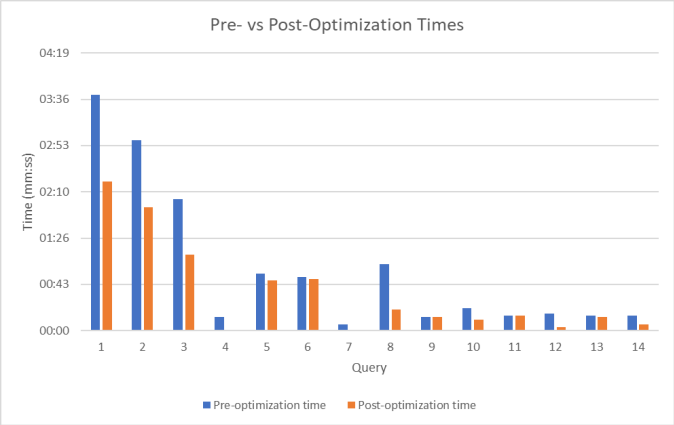
The dataset that was used for this project falls under the category of transportation in the city of Chicago [1]. It has records of taxi trips reported by the city, where each record is a taxi trip with a unique Trip ID. This data could be useful for transportation agencies, especially taxis and cab companies, to better inform their systems by thoroughly analyzing the patterns in this data. The entire dataset published on the city of Chicago's Open Data Portal, has a size of around 49GB, but for our project, we will only be considering records from *'Trip Start Timestamp*' after 01/01/2018 12:00:00 AM to '*Trip End Timestamp*' till 02/12/2023 12:00:00 AM which has a size of 21GB. However, we will only be analysing 10GB of the above data because we use MS SQL Server as our data platform which allows up to 10GB of data to be loaded and worked upon.

For the third phase of our project, we had to re-import the data in order to alter the data types of some of the attributes. This caused an increase in the number of rows in the table as well as a decrease in some of the costs/times for the queries compared to those that we reported during the second phase of our project. Moreover, the syntax of the queries also changed from Phase 2 because they didn't require to be converted to appropriate types anymore. New syntax for each query is provided in the following sections.

To run our queries, we used different systems, denoted by S1-S3, details of which are provided in the *Data* section.

The queries we used throughout our experiment cover different types and aspects such as selection, projection, and aggregate operations. However, joins are not used as we only have one table and the data is not complex enough to join on itself in order to form interesting queries.

The final results of our optimization attempts varied greatly. Some queries resulted in an improved cost, while other queries had no or a minimal change in cost. We noticed that some optimization techniques work better than others depending on the query complexity. The detailed results will be provided in the *Analysis* section. However, we have included a preview of our final results in the image displayed below:



II. DATA

The dataset contains 1 table with 15,000,000 rows each of which represents a taxi trip record. There are 23 columns in the dataset as shown in the Table 1 below:

Attribute	**Type**
Trip ID	varchar(50)
Taxi ID	varchar(128)
Trip Start Timestamp	datetime
Trip End Timestamp	datetime
Trip Seconds	int
Trip Miles	float
Pickup Census Tract	varchar(50)
Dropoff Census Tract	varchar(50)
Pickup Community Area	int
Dropoff Community Area	int
Fare	float
Tips	float
Tolls	float
Extras	float
Trip Total	float
Payment Type	varchar(50)
Company	varchar(50)
Pickup Centroid Latitude	varchar(50)
Pickup Centroid Longitude	varchar(50)
Pickup Centroid Location	varchar(50)
Dropoff Centroid Latitude	varchar(50)
Dropoff Centroid Longitude	varchar(50)
Dropoff Centroid Location	varchar(50)

Table 1: City of Chicago Taxi Trips

The table has a size of 10,428,416 KB (~10GB). This data information differs from the one reported in Phase 1 and 2 because we had to reformat and re-import the dataset which led to some changes in these numbers. This data is a subset of the actual data reported by the city and takes into consideration the records from '*Trip Start Timestamp*' after 01/01/2018 12:00:00 AM to '*Trip End Timestamp*' till 02/12/2023 12:00:00 AM. The data was available in a csv file in its raw form. In order to make the best use of the data and the optimization strategies, we converted the data types into appropriate format before importing the dataset.

We used Microsoft SQL Server to manage our data and MS SQL Management Studio to run and evaluate our queries on the given data. We used three systems overall (S1, S2, S3). We divided the queries to be run on different systems and the times of optimizations (i.e. using indexes, converting to CNF, etc.), were captured using that particular system. For some queries where we didn't notice any significant improvements, we tried a slower system to verify the results. Further information will be included in the *Analysis* section of the report.

The system and SQL server specifications are listed below along with the specific queries that were run on them.

S1:

System Specs:

Installed Physical Memory (RAM)	8.00 GB
Total Physical Memory	7.87 GB
Available Physical Memory	849 MB
Total Virtual Memory	14.6 GB
Available Virtual Memory	4.71 GB
Page File Space	6.75 GB
Page File	C:\pagefile.sys

SQL Server Specs:

Name	DESKTOP-V71KVLQ\SQLEXPRESS
Product	Microsoft SQL Server Express (64-bit)
Operating System	Windows 10 Home (10.0)
Platform	Windows
Version	16.0.1000.6
Language	English (United States)
Memory	8058 MB
Processors	4
Root Directory	C:\Program Files\Microsoft SQL Server\MSSQL16.S
Server Collation	SQL_Latin1_General_CP1_CI_AS
Is Clustered	False
Is XTP Supported	True
Is HADR Enabled	False

Queries run: Q1, Q2, Q4, Q9, Q11, Q12, Q13, Q14

S2:

System Specs:

Installed Physical Memory (RAM)	16.0 GB
Total Physical Memory	15.8 GB
Available Physical Memory	6.88 GB
Total Virtual Memory	29.2 GB
Available Virtual Memory	11.5 GB
Page File Space	13.4 GB
Page File	C:\pagefile.sys

SQL Server Specs:

Name	DESKTOP-UE954NU\SQLEXPRESS
Product	Microsoft SQL Server Express (64-bit)
Operating System	Windows 10 Pro (10.0)
Platform	Windows
Version	16.0.1050.5
Language	English (United States)
Memory	16194 MB
Processors	24
Root Directory	D:\Program Files\Microsoft SQL Server\MSSQL16.
Server Collation	SQL_Latin1_General_CP1_CI_AS
Is Clustered	False
Is XTP Supported	True
Is HADR Enabled	False

Queries run: Q3, Q5, Q6

S3:

System Specs:

Installed Physical Memory (RAM)	16.0 GB
Total Physical Memory	15.4 GB
Available Physical Memory	5.18 GB
Total Virtual Memory	19.9 GB
Available Virtual Memory	3.74 GB
Page File Space	4.50 GB
Page File	C:\pagefile.sys

SQL Server Specs:

Name	LAPTOP-75F25OUN\SQLEXPRESS
Product	Microsoft SQL Server Express (64-bit)
Operating System	Windows 10 Home (10.0)
Platform	Windows
Version	16.0.1050.5
Language	English (United States)
Memory	15776 MB
Processors	16
Root Directory	C:\Program Files\Microsoft SQL Server\MSSQL16.SQ
Server Collation	SQL_Latin1_General_CP1_CI_AS
Is Clustered	False
Is XTP Supported	True
Is HADR Enabled	False

Queries run: Q7, Q8, Q10

III. Experiments

In an attempt to optimize these queries, we utilized several methods and techniques. These techniques include: rewriting the queries to be more efficient (i.e. changing "SELECT * …" queries, pre-converting queries to CNF, using less costly functions, etc.), and using many different indexes on appropriate columns. We have decided to use these optimization strategies for the following reasons:

Firstly, we decided to rewrite inefficient queries to reduce unnecessary processing time. For instance, we changed "SELECT * …" queries to not select all attributes because retrieving the additional columns was unnecessary for the purposes of our queries. Additionally, we changed some queries by pre-converting them to a conjunctive normal form in an attempt to bypass any time it would take for the system to convert it for us. We also tried different functions, such as YEAR(), to determine if the use of these functions is less costly.

Secondly, we used many different indexes throughout our experiment. The indexes we created include: <Tips>, <Payment Type>, <Trip Start Timestamp, Trip End Timestamp>, <Company>, <Pickup Centroid Latitude, Pickup Centroid Longitude, Tips>, <Trip Seconds, Trip Miles>, and <Trip Total>. Both clustered and non-clustered indexes

were used throughout the experiment to determine a difference between the two. These indexes were created in an attempt to decrease the search and retrieval costs of records, instead of scanning the entire table each time.

For creating indexes, the following syntax was used:

CREATE INDEX name ON dbo.taxiTrips (column [ASC/DESC]);

In SQL Server, this creates a non-unique, non-clustered (by default) B+ tree index on the specified column(s), which is sorted in ascending (ASC) or descending (DESC) order on the column, if applicable.

For creating clustered indexes, the following syntax was used:

CREATE CLUSTERED INDEX name ON dbo.taxiTrips (column [asc/desc]);

The syntax for each query is provided in separate files. However, we will describe the purpose and discuss the specific optimization techniques used on each query below:

Query 1:

Description: This query aims to determine how many taxi trips received no tips at all.

To optimize this query, the most significant optimization strategy we used was to create an index (B+ tree, non-unique, non-clustered) on 'Tips'. Furthermore, we restructured the query to only include attributes of interest, i.e., [*Trip ID], [Taxi ID], [Tips]* instead of all (*). Finally, we used a subquery to analyze if the query can be further optimized.

The SQL syntax with the subquery method is shown as follows:

```
SELECT [Trip ID], [Taxi ID], [Tips]
FROM dbo.taxiTrips WHERE [Trip ID]
IN (SELECT [Trip ID] FROM dbo.taxiTrips WHERE
[Tips] = 0);
```

Query 2:

Description: This query aims to determine the taxi trips that received the payment in cash.

To optimize this query, the most significant optimization strategy we used was to create an index (B+ tree, non-unique, non-clustered) on 'Payment Type'. Furthermore, we restructured the query to only include attributes of interest, i.e., [*Trip ID], [Taxi ID]* instead of all attributes. Finally, we used a subquery to analyze if the query can be further optimized.

The SQL syntax with the subquery method is shown as follows:

```
SELECT [Trip ID], [Taxi ID]
FROM dbo.taxiTrips
WHERE [Trip ID]
IN (SELECT [Trip ID] FROM dbo.taxiTrips WHERE
[Payment Type] = 'Cash' );
```

Query 3:

Description: This query aims to select all taxi trips that occurred in the year 2019.

We create an index (B+ tree, non-unique, non-clustered) on columns 'Trip Start Timestamp' and 'Trip End Timestamp' together. Furthermore, we restructured the query to only include attributes of interest, i.e., [*Trip ID], [Taxi ID]* instead of all attributes.

The SQL syntax with the optimization is shown as follows:

```
SELECT [Trip ID], [Taxi ID]
FROM dbo.taxiTrips
WHERE [Trip Start Timestamp] >= '2019-01-01
00:00:00.000' AND [Trip End Timestamp] < '2019-
12-31 23:59:59.999';
```

Query 4:

Description: This query displays how many trips cost over $1000 per taxi company, from greatest to least.

For this query, we created an index (B+ tree, non-unique, non-clustered) on 'Trip Total' using the index-creation syntax shown at the start of this section.

Query 5:

Description: This query calculates the total amount of trip earnings per taxi company and sorts by greatest to least.

To optimize this query, we created a B+ tree index on <Company> (non-unique, non-clustered) that is sorted by Company in ascending order.

Query 6:

Description: This query determines how trip earnings are split between payment types.

To optimize this query, we created a B+ tree index on <Payment Type> (non-unique, non-clustered) that is sorted by PaymentType in ascending order.

Query 7:

Description: This query displays which month has the most trips for each company in 2019, from greatest to least.

To optimize this query, the most significant optimization strategy we used was to create an index on <Trip Start Timestamp> (non-unique, clustered). For further optimization, we used YEAR() instead of comparing the datetime. Finally, we used a subquery to analyze if the query can be further optimized.

The SQL syntax with the subquery method is shown as follows:

```
SELECT tmp.year, tmp.month, tmp.count,
tmp.Company
FROM (
SELECT YEAR([Trip Start Timestamp]) AS year,
MONTH([Trip Start Timestamp]) AS month,
COUNT(1) as COUNT, [Company]
FROM dbo.taxiTrips
GROUP BY MONTH([Trip Start Timestamp]),
YEAR([Trip Start Timestamp]), [Company] ) tmp
WHERE tmp.year = 2019
ORDER BY COUNT DESC;
```

Query 8:

Description: This query determines which pickup areas had the best average tips. This query was sorted by the average tip amount descending.

To optimize this query, the most significant optimization strategy we used was to create an index on <Pickup Centroid Latitude, Pickup Centroid Longitude, Tips> using a B+ tree index in SQL Server Manager.

Query 9:

Description: This query determines trips for company 'City Services' or 'Sun Taxi' which had payments made through mobile and lasted for more than 30 mins.

To optimize this query, the most significant optimization strategy we used was to convert the query into a conjunctive normal form, which is shown below:

```
SELECT * FROM dbo.taxiTrips where ([Company] =
'City Service' or [Company] = 'Sun Taxi') and
([Payment Type] = 'Mobile') and ([Trip Seconds] >
1800);
```

Query 10:

Description: This query determines which trips lasted for more than 1 hour and had distance greater than 10 km (6.214 miles). And, sorted by the time (trip seconds) in an increasing order.

To optimize this query, the most significant optimization strategy we used was to create an index on columns Trip Seconds and Trip Miles together, using a clustered B+ tree index in SQL Server Manager. Additionally, we also changed the selection attributes.

The SQL syntax with the changed selected attributes shown as follows:

```
SELECT [Trip ID], [Trip Seconds], [Trip Miles]
FROM dbo.taxiTrips WHERE [Trip Seconds] > 3600
AND [Trip Miles] > 6.214 ORDER BY [Trip
Seconds] ASC;
```

Query 11:

Description: This query determines how many trips have more tips than fares by each company, sorted by the number of trips in decreasing order.

To optimize this query, we created an index on the 'Company' column, using a clustered B+ tree index in SQL Server Manager.

Query 12:

Description: This query determines which payment method is preferred.

To optimize this query, we created an index on the 'Payment Type' column, using a clustered B+ tree index in SQL Server Manager.

Query 13:

Description: This query was run to determine which trips had fare and tips equal to 0 or had a trip total and trip miles as 0.

To optimize this query, the most significant optimization strategy we used was to convert the query into a conjunctive normal form, which is shown below:

*SELECT * FROM dbo.taxiTrips WHERE (Fare = 0 or [Trip Total] = 0) and (Fare = 0 or [Trip Miles] = 0) and (Tips = 0 or [Trip Total] = 0) and (Tips = 0 or [Trip Miles] = 0);*

Query 14:

Description: This query aims to determine the month with the highest fares.

To optimize this query, the most significant optimization strategy we used was to create an index on <Trip Start Timestamp, Fare> using a B+ tree index in SQL Server Manager.

IV. ANALYSIS

In this section, we aim to discuss the results of the experiments conducted above. We will list a breakdown of each query, including the system(s) it was run on, its cost pre-optimization and post-optimization, a detailed analysis of the results, and potential improvements (if applicable).

Note: Some of the queries were run multiple times before and after optimization to verify the validity of the results (i.e., to confirm if it's optimized or just a system delay).

Query 1:

System: S1

Pre-optimization time: 3 minutes 40 seconds

Post-optimization time:

With all (*) attributes, using index: 3 minutes 27 seconds

With attributes of interest only, using index: 1 minute 39 seconds

Using subqueries, with attributes of interest only: 2 min 15 sec - 2 min 19 sec

Analysis:

After conducting the experiment on query 1, we didn't observe a significant improvement using index on column - 'Tips'. The query was run multiple times, and the time using an index seemed a few seconds faster than without using an index. This could also be a minor change in query times due to system delay and not actually an optimization. This also implies that SQL Server might already make use of some relevant optimization strategies prior to executing the query. Also since the table isn't sorted, the index on tips is non-clustered, which might not be as beneficial as a clustered one.

But, changing the query to only print the attributes of interest instead of * reduced the time significantly.

Similar optimizations were observed using the subquery method. However, the subquery method didn't reduce the time as compared to our query with only selected attributes, rather, it increased it by a few seconds. This implies that running a query in a subquery/nested form doesn't really help in this case because of the non-complex nature of the query.

Potential improvements:

SQL Server allows for B+ tree indexes (in our case, each page of the index was 100% full with a total of 40961 pages), but creating hash indexes could be even faster because they have less index scanning time. Furthermore, having a clustered index could prove to be more beneficial in this scenario. This can be achieved by sorting (using efficient sorting techniques discussed in class) the table on the index's search key (i.e., the attribute(s) of interest).

Query 2:

System: S1

Pre-optimization time: 2 min 58 sec - 3 min 7 seconds

Post-optimization time:

With all (*) attributes, using index: 2 min 57 sec - 3 min

With attributes of interest only, using index: 1 min 35 seconds

Using subqueries, with attributes of interest only: 1 min 38 seconds — 1 min 55 seconds

Analysis:

Similar to the observations made in query 1, we noticed that index on column - 'Payment Type' wasn't a significant improvement in this case. The query was run multiple times, and the time using index seemed just a few seconds faster than without using an index. This could also be a minor change in query times due to system delay and not actually an optimization. This also implies that SQL Server might already make use of some optimization strategies beforehand.

Also since the table isn't sorted, the index on payment types is non-clustered, which might not be as beneficial as expected.

Similar to the observations noted in query 1's analysis, slight optimizations were observed using the subquery method. However, the subquery method didn't reduce the time as compared to our query with only selected attributes, rather, it increased it by a few seconds. This implies that running a query in a subquery/nested form doesn't really help in this case because of the non-complex nature of the query.

Potential improvements:

Similar potential improvements can be suggested for this query. Since SQL Server only allows for B+ tree indexes (in our case, each page of the index was 99.96% full with a total of 46977 pages), creating hash indexes could be even faster because they have less index scanning time. Furthermore, having a clustered index could prove to be more beneficial in this scenario. This can be achieved by sorting (using efficient sorting techniques discussed in class) the table on the index's search key (i.e., the attribute(s) of interest).

Query 3:

System: S2

Pre-optimization time: 2 minutes and 3 seconds

Post-optimization time:

With all (*) attributes, using index: 1 min 51 sec

With attributes of interest only, using index: 1 min 11 sec

Analysis:

Using an index on `<Trip Start Timestamp, Trip End Timestamp>` has noticeable improvements in the query time. Changing the query to only print the attributes of interests further reduces the time by a few seconds. It can be observed that having an index on those relevant columns helped to retrieve records faster as opposed to sequentially scanning the entire file. This might not have been as advantageous if an index was created only on 1 attribute.

Potential improvements:

Having a clustered index could prove to be more beneficial in this scenario. This can be achieved by sorting the table on the index's search key (i.e., the attribute(s) of interest) using general multiway mergesort algorithm with appropriate number of buffer pages.

Query 4:

System: S1

Pre-optimization time: 13 seconds

Post-optimization time: 0 seconds

Analysis:

Using index on column - 'Trip Total' optimized the query by a significant amount. From running our previous set of experiments, we noticed that indexing is more helpful in case of slightly complex queries as opposed to very simple ones. Typically, the ones including the range searches get optimized better using indexes.

Query 5:

System: S2

Pre-optimization time: 53 seconds

Post-optimization time: 47 seconds

Analysis:

A slight improvement was made using the index on column - 'Company'. We suspect this is because the query requires a scan of each tuple for each company

anyway to get the sum of the trip totals per company. However, the slight improvement may come from the "group by" statement, as the index is already sorted on Company, which may make it easier to group the calculations. On the other hand, this could possibly not be an optimization at all and could be due to system delay.

Query 6:

System: S2

Pre-optimization time: 50 seconds

Post-optimization time: 48 seconds

Analysis:

A very slight improvement was made using the index on column - 'Payment Type'. We suspect this is because the query requires a scan of each tuple for each payment type anyway to get the sum of the trip totals per payment type. However, the slight improvement may come from the "group by" statement, as the index is already sorted on Payment Type, which may make it easier to group the calculations. On the other hand, this could possibly not be an optimization at all and could be due to system delay.

Query 7:

System: S3

Pre-optimization time: 6 seconds

Post-optimization time:

Using subqueries, without index: 6 seconds

With attributes of interest only, using index: 1 second

Analysis:

We found that using the clustered index on `Trip Start Timestamp` has significant improvements. Creating the clustered index on `Trip Start Timestamp` will change the physical order of the data in the table so that it is now more efficient for range scans of the data. For further improvement, we tried using the YEAR() instead of comparing the datetime. However, we noticed that using YEAR() increased the running time.

Also, we didn't observe any significant improvement after restructuring the query to include subquery in it.

Query 8:

System: S3

Pre-optimization time: 1 minute and 2 seconds

Post-optimization time: 20 seconds

Analysis:

There was a significant improvement in the time the query took to run after creating an index on <Pickup Centroid Latitude, Pickup Centroid Longitude, Tips>. It can be observed that having an index on those relevant columns helped to retrieve records faster as opposed to sequentially scanning the entire file. This might not have been as advantageous if an index was created only on 1 attribute.

Query 9:

System: S1

Pre-optimization time: 12-14 seconds

Post-optimization time: 12-13 seconds

Analysis:

After re-writing the query in a conjunctive normal form, no significant improvements were observed. It implies that SQL server already optimizes the query by converting it into a CNF prior to running the query which doesn't take that long for a short or non-complex query.

To analyse this optimization strategy again, another query **Query 13** was run on System S1 and similar results were observed. It took 13-14 seconds prior to converting to CNF and took 12-13 seconds after the conversion. The slight difference could be generated due to system delay and hence, no significant improvement can be reported.

However, converting to conjunctive normal forms could be helpful in case of complex queries involving a lot of conjuncts with different attributes and different operations of those attributes.

Potential Improvements:

On top of using CNF, the indexes on appropriate columns could be added to optimize the query cost.

Queries with similar results/analyses: Query 13

Query 10:

System: S3/S1

Pre-optimization time: 8 seconds/20-22 seconds

Post-optimization time: 0 seconds/10 seconds

Analysis:

This query was run on multiple systems, namely S1 and S3, and similar observations were made. After creating an index on <Trip Seconds, Trip Miles>, the time taken by the query to fetch required results was significantly reduced.

Potential Improvements:

Having a clustered index on the same attributes could prove to be even better on the slower systems (e.g., S1 in this case).

Query 11:

System: S1

Pre-optimization time: 13-14 seconds

Post-optimization time: 13-14 seconds

Analysis:

After creating an index on column - 'Company', we didn't observe any significant improvements. This could be due to the observations made earlier while experimenting on query 1 and query 2, where the index is non-clustered and doesn't add any betterment. We suspected that the index might not be on a relevant column, therefore, we created indexes on other columns <Fare, Tips> but we didn't achieve any differences in time.

This could potentially be due to the nature of the query which might involve scanning individual tuples anyway and hence, having an index doesn't add any value in this scenario.

Query 12:

System: S1

Pre-optimization time: 14-18 seconds

Post-optimization time: 2-4 seconds

Analysis:

Query time using B+ tree index on 'Payment Type' was much faster than without using an index. Query was run multiple times before and after optimization and the same results were observed each time, which implies that the index helped to retrieve appropriate records faster as opposed to scanning through the file sequentially.

Potential Improvements:

Further optimizations can be made by having a clustered index on the same attributes.

Query 13:

Note: Due to the similar results being observed for query 13 using the CNF optimization strategy, its analysis has been provided along with query 9 for a better comparison.

Query 14:

System: S1

Pre-optimization time: 13-15 seconds

Post-optimization time: 5-8 seconds

Analysis:

After running the experiment using an index on column - 'Trip Start Timestamp' and 'Fare', we observed that the query time was optimized by a significant amount. This allowed the system to fetch the records for appropriate columns faster as opposed to sequentially scanning through the file.

Potential Improvements:

Further optimizations, especially on slower systems, can be made by having a clustered index on the same attributes.

V. Discussion

Additional Insights:

When we conducted one of the experiments, we accidentally ran the query after creating a clustered index on the wrong attribute. This caused the table to be re-sorted on something irrelevant to the query. As a result, running the query took a substantial amount of time, compared to running the query without the clustered index. Thus, we were able to observe the effects of using a clustered index and the negative effects of using the wrong one.

Further, on researching SQL server's optimization strategies, we learnt that it automatically creates a non-clustered index on a column containing primary or unique constraints [3]. However, we couldn't analyse this aspect of it because our table didn't have a primary or a candidate key attribute.

Reflection:

We noticed that many of our results showed very minimal or no change at all. Some of the pre-optimization times, after reimporting the data with the new data types, were also very close to 0. If we had more time and resources, we would have liked to run these experiments again on a larger file size to observe more change. Additionally, we would run the optimizations multiple times on multiple systems to determine if the change was due to the optimization or system delay. This would have allowed us to see if the optimization proves to be helpful on a slower system as opposed to a faster system.

We would have also liked to experiment more with clustered indexes. However, that requires the table to be sorted on the index's search key, which could further add to the time complexity of the query. This could potentially be improved by using a general multiway mergesort using the appropriate number of buffer pages.

Also, since almost every index in our experiment was a B+ tree index, which was created by SQL Server by default, we didn't get a chance to explore the other types of indexes such as different hash indexes, bitmap indexes, etc. Since their overall I/O cost of retrieving records is typically lower than the B+ trees, they would have yielded better optimization results for some queries. Hash indexes would have been better for queries performing equality searches.

Furthermore, using a different dataset with multiple tables would have also been interesting, so that we could observe the changes to optimization times using joins. Also, since we could only analyze around 10GB of the data due to SQL Server's limited data capacity, we might not have seen significant optimization results for some of our queries due to that reason. As discussed above, having a larger database with multiple tables could have allowed us to make more interesting queries and achieve more interesting results.

References

[1] City of Chicago, "Taxi trips", City of Chicago Open Data Portal. [Online]. Available: https://data.cityofchicago.org/Transportation/Taxi-Trips/wrvz-psew/data

[2] "SQL server management studio (SSMS)," *SQL Server Management Studio (SSMS) | Microsoft Learn*. [Online]. Available: https://learn.microsoft.com/en-us/sql/ssms/download-sql-server-management-studio-ssms?view=sql-server-ver16

[3] MikeRayMSFT, "Clustered and nonclustered indexes described - SQL server," *SQL Server | Microsoft Learn*. [Online]. Available: https://learn.microsoft.com/en-us/sql/relational-databases/indexes/clustered-and-nonclustered-indexes-described?view=sql-server-ver16. [Accessed: 12-Apr-2023].