Query Optimization

Katrina Dotzlaw, Simran Kaur, Aardarsh Patel, Smit Patel
Department of Computer Science
University of Manitoba
Winnipeg, MB, Canada

***Abstract*—** Queries are a vital part of database management systems, allowing users to retrieve and manipulate data. This paper examines the performance of various queries and optimizations. The process of query optimization involves selecting the most efficient plan for a query. However, with the increasing size and complexity of data and database management systems, efficient query optimization has become more vital than ever. In this paper, we explore various optimizations and analyse the results via comparing runtimes.

Keywords—database, database optimization, query optimization, indexing

I. Introduction

A. Summary of Data

Our dataset [1] consisted of parking and camera violations collected by the Department of Finance. This dataset is updated daily, so only a subset of data was used, specifically data uploaded from May 2016 to February 2023. This dataset includes multiple types of parking violations and associated fine data such as amount, penalties, and interest. It also includes license plate information, dates corresponding to violation occurrences, and judiciary hearings. This data could be used by the different levels of financial government (state, county, precinct) to track the different types of parking violations for their area and use the collected statistics to aid budget predictions and safety laws. This dataset could also be used by issuing agencies to track the distribution of parking violations compared to other agencies.

B. Queries

We developed 10 queries of varying complexity, collected the number of rows returned and the runtime.

Query 1 counts the total number of speed violations that occurred in a school zone and groups them by state. Knowing the total number of school zone speed violations would allow the government and police to determine if the severity of the violation needs to be increased. Query 1 returns 55 rows in 175.809 seconds, each row containing the total number of plates that have violations, and the states.

Query 2 lists all the license plate numbers with their relative fine amounts for anyone with a fine between $100.00 and $400.00. This could be helpful to create reports and analyze traffic plans and budgets by segregating fine amounts into ranges. Query 2 returns 29,422,339 rows in 199.743 seconds, each row containing plate numbers and the fine amount.

Query 3 provides the average of all fine amounts on a per state basis. This could be useful to compare the amount of fines between different states and allocate finances accordingly. This query returns 70 rows in 377.486 seconds, each row containing state and Avg_Fine_Amount.

Query 4 provides the sum of the total amount due for each type of violation, grouped by issuing agency. This can be helpful to assess which agencies have a high number of fine amounts and analyze agency performance. Query 4 returns 33 rows in 604.503 seconds, containing the issuing agency and the total fine amount for that agency.

Query 5 determines the number of violations that have bigger fines. These results can be used to determine and create monthly reports to analyze the changes in violation severity and allocate law enforcement accordingly. This query returns 10 rows in 156.6 seconds, with each row containing a fine amount and the number of plates that received a fine of that amount.

Query 6 counts the number of violations done by individuals with a non-PAS type license and groups results by violation and license type. This could track violations committed by commercial vehicles. Query 6 returns 3225 rows in 2248.581 seconds, each row containing the license type, violation, and the total number of plates.

Query 7 counts the total number of violations for each active violation in NY. This could identify common violations and inform law enforcement on how to prioritize the violations. Query 7 returns 98 rows in 536.355 seconds, each row containing the violation type and the number of violations for the state of NY.

Query 8 obtains all of the violations that are associated with each unique license plate and orders the results by the number of violations. Using these results, we can identify which cars can be trusted the least. This query returns 13,221,955 rows in 536.343 seconds, with each row containing a distinct license plate and the total number of violations for each plate.

Query 9 obtains the total number of violations for a specific date. This can help identify any associations between violations and days. Query 9 returns a single row in 143.653 seconds, containing the number of violations on the given date.

Query 10 obtains the total number of violations that took place on a yearly basis, beginning in 2000 and ending in the current year. These results could be used to identify any violation patterns differing per year. Query 10 returns 24 rows in 464.673 seconds, each row containing the number of violations and the year.

II. Data

A. Data Information

As previously mentioned, the Open Parking and Camera Violations dataset[1] is collected by the Department of Finance and is updated daily on the NYC OpenData portal. It contains data relating to multiple types of parking violations and associated fine data such as amount, penalties, and interest. Additionally, it contains data about associated license plate information, violation occurrence and information about judiciary hearings.
Each row in the dataset represents a separate violation and could contain any number of the 18 columns included. The 'plate' column tracks the license plates of cars that have associated parking violations or been caught by a camera. The 'license_type' column defines the class of license issued to the individual that received the violation. Columns 'state' and 'county' track where the fine was issued, while columns 'issuing_agency' and 'precinct' track the law enforcement agency issuing the fine and the precinct associated with the law enforcement officer. The time and date of the parking violation is recorded in the columns 'issue_date' and 'violation_time'. The type of violation that occurred and the associated status are stored in the 'violation' column and the 'violation_status' column respectively. Associated fine data is stored in columns 'fine_amount', 'penalty_amount','interest_amount', 'reduction_amount', 'payment_amount', and 'amount_due'. If a violation requires a jury hearing, the columns 'summons_number' and 'judgement_entry_date' would indicate that.

B. Technical Information

To manage and create our database, we used DB Browser for SQLite [2]. This DBMS allowed us to directly import the Parking Violations CSV file as a flat file, instead of manually importing millions of rows of data. Originally, the dataset[1] contained 19 columns, but we decided to exclude the column "SummonsImage" as it contained http links to images of jury summons and we did not think that anyone querying this database would find it useful.
After preprocessing, our database has a single table that has a file size of 25.520688GB, and 18 columns. Our database has approximately 93.9 million rows.

All queries will be run using DB Browser for SQLite [2] on a Windows 10 64-bit operating system. The device CPU is an AMD Ryzen 7 3700X which has 8 cores and 16 threads. The device RAM uses a 16.0GB Dual-Channel DDR4 at 1333MHz. It has a PRIME X570-PRO motherboard from ASUSTeK. The device uses a 2047MB NVIDIA GeForce GTX 1660 SUPER graphics card from ASUSTeK. It has 931GB of total storage on the hard drive, with 748GB currently used.

III. Experiments

A. Preliminary Investigation

Before conducting any experiments, we investigated the types of query optimizations allowed in SQLite. According to the SQLite documentation [3], all indexes created on a table that uses the default 'rowid' column are **non-clustered**. SQLite allowed us to analyze query execution using the EXPLAIN QUERY PLAN keyword, to identify and display the optimal execution of the query[4]. Using this functionality, we found that all indexes used in SQLite are **B+ tree** indexes and that SQLite automatically creates temporary **B+ tree** indexes for group by and order by functions. We also found that SQLite does not support bitmap indexes, so our original optimization plans for queries 2 and 5 would need to change. In terms of data, we found that our data was heavily skewed towards the state of NY, which would affect how we could optimize our queries.

We decided to use partial indexing in addition to the default type of indexing, which allowed us to create indexes that used a "where" clause to further filter the results. Our hypothesis was that further filtering of the data would greatly impact the optimization results, since our dataset[1] was extremely large.

Additionally, we found that some of the indexes created were covering indexes, which meant that all of the columns present in the query would be used in the index and would not need a table scan [5]. We decided to keep these indexes, as our hypothesis was that eliminating a table scan would allow for significant optimizations.

In addition to our query experiments, we also examined how individual pages and storage were used. We compared the amount and distribution of both primary pages and index pages between our indexed table and our original table. We display and discuss these results in the Analysis section of this paper.

B. Query Experimentation

As previously mentioned, the SQLite EXPLAIN QUERY PLAN functionality was used to examine how queries are being executed, including which indexes were used.

Query 1:

According to the query plan, we found that the unoptimized query would use a table scan and a temporary B+ tree index for the group by function, resulting in a 181.197 second runtime. Our first optimization attempt was to create an index on the **<State>** column, which we confirmed was being used in the table scan. This index resulted in a runtime of 468.512 seconds, which is significantly slower than the unoptimized query. Our second optimization attempt was to create an index on **<State, Plate, Violation>**. This optimization allowed the query to perform a table scan using the covering index we created and resulted in a significantly faster runtime of 16.223 seconds. Finally, we created a partial index on **<State, Plate, Violation> where Violation = PHTO SCHOOL ZN SPEED VIOLATION**. Again, this optimization allowed the query to perform a table scan using the covering index, which resulted in an improved runtime of 4.112 seconds.

Query 2:

According to the query plan, we found that the unoptimized query would use a table scan and a temporary B+ tree for the "order by" function, resulting in a 199.743 second runtime. Our first optimization was to create an index on the **<Plate>** column, which uses a temporary B+ tree index on the "order by" function. The index on this column did not work properly, as we ran it for literally 2 hours but we did not get any results. We then created an index on **<Plate, FineAmount>**, which scans the table using the covering index <**Plate, FineAmount>** and uses the temporary B+ tree on "order by" function. This index significantly improved the runtime of the query, resulting in a runtime of 79.965 seconds. Our last and best optimization was to create a partial index on **<Plate, FineAmount ASC>,** which scans the table using the covering index **<Plate, FineAmount ASC>**, and uses the temporary B+ tree on the "order by" function. This improved the query performance even more, resulting in a runtime of 47.781 seconds.

Query 3:

According to the query plan, we found that the unoptimized query would use a table scan, use a temporary B+ tree on "group by" function and a temporary B+ tree on the "order by" function, resulting in a 318.803 second runtime. Our first optimization was to create an index on the **<State>** column, which allowed the table scan to use the index created. The index on this column did work, but the runtime for this query was significantly worse than using an unoptimized query. It resulted in a runtime of 878.753 seconds.We then created an index on **<State, FineAmount>**, which scanned the table using the index on **<State, FineAmount>** and used the temporary B+ tree on "order by" function. This index resulted in a significant improvement in the run time of query, resulting in a runtime of 56.142 seconds.

Query 4:

According to the query plan, we found that the unoptimized query would use a table scan, use a temporary B+ tree on "group by" function and a temporary B+ tree on the "order by" function, resulting in a 537.909 second runtime. Our first optimization was to create an index on the **<IssuingAgency>** column, which scans the table on index **<IssuingAgency>** and uses a temporary B+ tree index on the "group by" function. The index on this column did work, and the runtime performance for this query was better than an unoptimized query. It resulted in a runtime of 453.228 seconds. We then created an index on **<IssuingAgency, AmountDue>**, which scans the table using the covering index **<IssuingAgency, AmountDue>**, and uses the temporary B+ tree on the "order by" function. This index significantly improved the runtime of the query, resulting in a runtime of 58.427 seconds.

Query 5:

According to the query plan, we found that the unoptimized query would use a table scan, use a temporary B+ tree index on "group by" function, resulting in a 99.989 second runtime. Our first optimization was to create an index on the **<Plate, FineAmount>** column, which scans the table using the covering index, and uses a temporary B+ tree index on the "group by" function. The index on this column did work and the runtime performance for this query was better than an unoptimized query. It resulted in a runtime of 44.219 seconds. Next we created an index on **<Plate, FineAmount ASC>**, which scans the table using the covering index, and uses the temporary B+ tree on the "group by" function. This index did improve the runtime compared to the unoptimized query, but did not improve the result significantly compared to the previous optimized query. The run time of the query was 44.306 seconds.

Query 6:

According to the query plan, we found that the unoptimized query would use a table scan and a temporary B+ tree on "group by" function, resulting in a 189.81 second runtime. Our first optimization was to create an index on the **<LicenseType, Violation>** column, which scans the table using the index on **<LicenseType, Violation>**. The index on this column did not give any improvement in the query performance, resulting in a runtime of 569.367 seconds . Then we created an index on **<LicenseType, Violation, Plate>**, which scans the table using the covering index **<LicenseType, Violation, Plate>**. This index resulted in a

significant improvement in the run time of query, resulting in a runtime of 15.564 seconds.

Query 7:

According to the query plan, we found that the unoptimized query would use a table scan and a temporary B+ tree on "order by" function and "group by" function, resulting in a 467.013 second runtime. Our optimization was to create an index on the **< Violation, State>** column, which scans the table using the index on **< Violation, State>** and a temporary B+ tree for the "order by" clause. The index on this column did improve the query performance, resulting in a runtime of 69.005 seconds .

Query 8:

According to the query plan, we found that the unoptimized query would use a table scan, use a temporary B+ tree for "group by", "distinct", and "order by" functions, resulting in a 473.568 second runtime. Our first optimization was to create an index on the **<Plate>** column, which scans the table and uses a temporary B+ tree index on the "order by" function and "distinct" function. The index on this column performed really well, as the query only took 117.003 seconds to complete. Next we created an index on **<Plate, Violations>**, which scans the table using the covering index **<Plate, Violations>**, and uses the temporary B+ tree on "order by" and "distinct" functions. This index did not do well compared to the previous optimization attempt, resulting in a runtime of 125.634 seconds, but did significantly improve runtime compared to the original unoptimized query.

Query 9:

According to the query plan, we found that the unoptimized query would scan the table, resulting in a runtime of 83.302 seconds. Our first and only optimization was to create an index on **<IssueDate>**, which searches the table using the covering index on **<IssueDate>**, and it performs significantly better than the unoptimized query. It results in a runtime of just 14 ms.

Query 10:

According to the query plan, we found that the unoptimized query would use a table scan, use a temporary B+ tree index on "group by" function, which resulted in a 398.887 second runtime. Our first optimization was to create an index on the **<Violation, IssueDate>** column, which scans the table using the covering index **<Violation, IssueDate>**, and uses a temporary B+ tree index on the "group by" function. The index on this column did work, and the runtime performance for this query was better than an unoptimized query. It resulted in a runtime of 305.835 seconds. Then we created a partial index on **<Violation, IssueDate>**, which scans the table using the covering index, and uses the temporary B+ tree on the "group by" function. The partial index included the **where** clause, which used string functions to determine if the year was between 2000 and 2023. This index did better than the previous optimization attempt. The run time of the query was 243.741 seconds.

IV. Analysis

A. Optimization Results

Using non-clustered B+ tree indexes in SQLite, we managed to achieve significant runtime optimizations of our original queries. After performing our query optimizations, we discovered that indexes created on multiple columns and partial indexes tended to have a significant improvement in runtime.

In query 1, we managed to improve the runtime from 181.197 seconds to 16.223 seconds using an index on multiple columns, and 4.112 seconds when using a partial index. In query 2, we optimized runtime by using a covering index, which reduced runtime to 79.965 seconds, and a partial index, which reduced runtime to 47.781 seconds. We managed to reduce the runtime of query 3 from 318.803 seconds to 56.142 seconds using an index on multiple columns. In query 4, we improved the runtime from 537.909 seconds to 453.228 seconds and 58.427 seconds, by creating an index on multiple columns and a covering index on multiple columns respectively. Query 5 had a comparatively better unoptimized runtime to the other queries, but we were able to optimize it using an index on multiple columns and a partial index, which both resulted in a runtime of approximately 44 seconds. In query 6, we optimized the runtime by indexing on multiple columns which resulted in a speed up of 174.246 seconds compared to the original. Query 7 also had an improved runtime using an index on multiple columns, which improved the runtime from 467.013 seconds to 69.005 seconds. For query 8, we achieved an optimization using an index on a single column, which resulted in a speed up of 356.565 seconds. We attempted to optimize query 8 further by indexing on multiple columns (which used a covering index), but the runtime was approximately 8 seconds worse than the previous optimization. Query 9 was optimized using a single index on a single column and had a significant improvement in runtime from 83.302 seconds to 14ms, likely because the query involves an exact value and a small subset of data. Finally, we optimized query 10 by creating 2 different indexes on multiple columns, resulting in improved runtimes of 305.835 seconds and 243.741 seconds compared to the original runtime of 398.887 seconds.

	Unoptimized Runtime	Index 1 Runtime	Index 2 Runtime	Index 3 Runtime
Query 1	181.197s	<State, Plate,Violations> partial: 4.112s	<State, Plate,Violations>: 16.223s	<State>: 468.512s
Query 2	199.743s	<Plate, FineAmount ASC>: 47.78s	<Plate, fineamount>: 79.965s	<Plate>: >2 hrs
Query 3	318.803s	<State, FineAmount>: 56.142s	N/A	<State>: 878.753s
Query 4	537.909s	<IssuingAgency, AmountDue>: 58.427s	<IssuingAgency>: 453.228s	N/A
Query 5	99.989s	<Plate, FineAmount>: 44.219s	<Plate,FineAmount ASC>: 44.306s	N/A
Query 6	189.81s	<LicenseType, Violation, Plate>: 15.564s	N/A	<LicenseType, Violation>: 569.367s
Query 7	467.013s	<Violation,State>: 69.005s	N/A	N/A
Query 8	473.568s	<Plate >: 117.003s	<Plate,Violation>: 125.634s	N/A
Query 9	83.302s	<IssueDate>: 14ms	N/A	N/A
Query 10	398.887s	<Violation, IssueDate> partial: 243.741s	<Violation, IssueDate>: 305.835s	N/A

Table 1: Comparison of unoptimized runtimes with created indexes, organized by speed

As shown in table 1, an index on a single column usually causes a significantly worse runtime and this is demonstrated in the <**state**> index of query 1 and 3, and <**Plate**> index of query 2. There were instances where an index on a single column performed well (as shown in query 8 and 9), presumably because the queries selected a significantly smaller subset of data. Additionally, queries that used a covering index consistently had a faster runtime. This is demonstrated in the indexes created for queries 1, 2, 4, 5, 8,9, and 10.

B. Additional Analysis

We used the SQLite3 Analyzer[6] to examine our database. This tool allows us to see how pages and bytes are being used, both for indexing and without indexing. Additionally, this tool also provides breakdowns of each index created, although that will not be discussed in this paper.

Using this software, we found that each page is 4096 bytes and that the entire database uses a total of 12862344 pages. We can also see a breakdown of how many pages each index uses as shown in figure 1. Based on these results, we can see that both indexes for query 10 and the second index for query 1 use the most pages, while the second index for query 2 uses the least.

```
PARKINGVIOLATIONS..... 3091951     24.0%
Q10A.................. 1026696      8.0%
Q10B.................. 1025887      8.0%
Q1B................... 1024069      8.0%
Q8B................... 955403       7.4%
Q6A................... 867518       6.7%
Q7.................... 842921       6.6%
Q4B................... 640919       5.0%
Q4A................... 528860       4.1%
Q2A................... 516056       4.0%
Q5.................... 516056       4.0%
Q9A................... 452269       3.5%
Q3B................... 404512       3.1%
Q1A................... 270554       2.1%
Q3A................... 270554       2.1%
Q1C................... 258236       2.0%
Q2B................... 169881       1.3%
SQLITE_SCHEMA......... 1            0.0%
```

Figure 1: Overview of Pages used per Index

We can also see additional metadata about storage space used per database entry, as shown in figure 2. Using these results, we can see that the database used a total of 52684152832 bytes of storage. Additionally, we can see that each entry has 0.26 bytes that were unused, which could potentially be optimized with more time and a deeper understanding of the internal layout of pages.

```
*** Table PARKINGVIOLATIONS and all its indices ******************************

Percentage of total database..... 100.000%
Number of entries..... 1439037963
Bytes of storage consumed..... 52684152832
Bytes of payload..... 47393406453   90.0%
Bytes of metadata..... 4910946943    9.3%
Average payload per entry..... 32.93
Average unused bytes per entry..... 0.26
Average metadata per entry..... 3.41
Average fanout..... 124.00
Maximum payload per entry..... 185
Entries that use overflow..... 0           0.0%
Index pages used..... 103596
Primary pages used..... 12758746
Overflow pages used..... 0
Total pages used..... 12862342
Unused bytes on index pages..... 6569464     1.5%
Unused bytes on primary pages..... 373229972   0.71%
Unused bytes on overflow pages..... 0
Unused bytes on all pages..... 379799436     0.72%
```

Figure 2: Bytes used for ParkingViolations database and associated indexes

Since all indexes created in SQLite are B+ tree indexes, we know that there should be no overflow pages in use and this is confirmed in figure 2. We can see that the indices used in our database use a total of 103596 index pages and 12758746 primary pages.

The SQLite Analyzer[6] also performs an analysis on the table without using any indexes, as shown in figure 3. Using this information, we can compare the results of indexing on a storage level.

```
*** Table PARKINGVIOLATIONS w/o any indices ***********************************

Percentage of total database.....  24.0%
Number of entries..... 92601266
Bytes of storage consumed..... 12664631296
Bytes of payload..... 11693815905  92.3%
Bytes of metadata..... 754392224    6.0%
B-tree depth..... 4
Average payload per entry..... 126.28
Average unused bytes per entry..... 2.34
Average metadata per entry..... 8.15
Average fanout..... 358.00
Non-sequential pages..... 0           0.0%
Maximum payload per entry..... 185
Entries that use overflow..... 0           0.0%
Index pages used..... 8619
Primary pages used..... 3083332
Overflow pages used..... 0
Total pages used..... 3091951
Unused bytes on index pages..... 4432753     12.6%
Unused bytes on primary pages..... 211990414    1.7%
Unused bytes on overflow pages..... 0
Unused bytes on all pages..... 216423167     1.7%
```

Figure 3: Analysis of Table without Indexes

As shown in figure 3, without using indexes the table uses 8619 index pages and 3083332 primary pages, which are less than the pages used with indexing. Additionally, we can see that without indexing, there is a higher percentage of unused bytes on pages (fig.3).

V. Discussion

During our experiment, we noticed that indexing on individual columns included in our queries was insufficient for optimization. Upon further investigation, we discovered that our dataset[1] was heavily skewed towards NY, which could have contributed to difficulty optimizing queries.

If we had more time, we could compare the runtimes of our queries using the same DBMS, but on different systems or OS'. This could allow us to identify an average optimization time on a diverse group of systems. Additionally, we could create our indexes on a different DBMS that uses a hash index or clustered index and compare runtimes to the results found using SQLite. This would allow us to investigate how the different types of indexes affect our specific dataset.

Another potential investigation would be to compare the optimization results of using SQLite's default non-clustered B+ index and using a covering non-clustered B+ index.

Acknowledgment

We would like to thank the NYC OpenData portal for allowing us to use their open-source parking violations dataset[1]. We would also like to thank Katrina Dotzlaw for creating and managing our SQLite database, running both the unoptimized and optimized queries, editing the report, and writing the introduction, analysis, and discussion sections of this report. We would also like to thank Smit Patel for developing queries and query optimizations, and writing the experiment section of this report. We would like to thank Simran Kaur for creating queries and its optimizations, editing the report, and writing the abstract section. We would also like to thank Aardarsh Patel for creating queries and optimizations.

VI. References

[1] NYC OpenData, "Open Parking and Camera Violations," Department of FInance, 29 03 2023. [Online]. Available: https://data.cityofnewyork.us/City-Government/Open-Parking-and-Camera-Violations/nc67-uf89 . [Accessed 16 02 2023].

[2] DigitalOcean, "DB Browser for SQLite," 25 03 2023. [Online]. Available: https://sqlitebrowser.org/ . [Accessed 16 02 2023].

[3] SQLite, "CREATE INDEX," 22 03 2023. [Online]. Available: https://www.sqlite.org/index.html . [Accessed 06 04 2023].

[4] SQLite Tutorial, "SQLite Expression-based Index," 2022. [Online]. Available: https://www.sqlitetutorial.net/sqlite-index-expression/ . [Accessed 06 04 2023].

[5] SQLite, "The SQLite Query Optimizer Overview," 22 03 2023. [Online]. Available: https://www.sqlite.org/optoverview.html . [Accessed 08 04 2023].

[6] SQLite, "SQLite Download Page -- Precompiled Binaries for Windows (sqlite-tools-win32)," 22 03 2023. [Online]. Available: https://www.sqlite.org/download.html . [Accessed 12 04 2023].