Project Phase 3 – Group 23

Group Members: Joseph Bernardin, Runyu Fang, Jainil Thakkar

I. INTRODUCTION

Our original dataset in phase one contained 4 tables. We dropped two of the tables because they were too small to be usable in our queries and unrelated to the two larger tables.

The data in these tables are very interesting as it lets us get new insights into how to Ethereum blockchain operates and the type of information you can get from it. With online blockchain explorers, it is not possible to see information like what we were able to find through our queries on our database.

We have come up with ten interesting queries regarding our data. We have queries that would be useful in a forensics situation, queries that give you insight into the massive amounts of money that are transferred around on the Ethereum blockchain on a regular basis and many others.

We put our queries through many optimizations in order to make them as fast as possible so that getting information that the user would want didn't take hours.

The queries we have are:

Query 1

The first query was to get the top 10 addresses who have sent the highest number of transactions.

Query 2

The second query was for addresses containing "eda7b6" that received deposits.

Query 3

The third query we had was for addresses who have sent the most Ethereum.

Query 4

The fourth query we had was for information about blocks between January 1st, 2020, and January 8th, 2020.

Query 5

The next transaction was for average gas per block.

Query 6

Our sixth query returned information about mining difficulty of blocks.

Query 7

Our seventh query was the top 10 highest transactions sent.

Query 8

This query was our favourite query we made on our database. It returns transaction records that have the same sender and recipient addresses. This is interesting because when the sender and recipient addresses are the same, usually this is to cancel a previous pending transaction.

Query 9

Our second to last query was to find the month and year that had the highest average transaction fees.

Query 10

Our final query was to get the address of the person who paid the most in transaction fees.

II. DATA

Our biggest table, transactions, is 22.17GB and contains 43,473,476 rows of data and has twelve columns. The other table, blocks, is 486.37MB and contains 439,334 rows of data and has 18 columns.

The computer the database was hosted on has a Ryzen 5825U (8 core, 16 thread) processor, 16GB of 4000MHz DDR4 RAM. The data was stored on a PCIe 3.0 NVMe Drive and the operating system is Fedora 36. The database management system used for our queries was SQLite3 through the command line.

III. EXPERIMENTS

Query 1

This query took an average of 101.429 seconds to return our results. To optimize the query, we created an index on the from_address attribute and ran the query again to see if there were any improvements.

Query 2

The average time for this query to run was 58.844 seconds. Putting an index on the to_address attribute was the only optimization we could find for this query.

Query 3

The query took 109.496 seconds on average. Our first attempt at optimizing this query was to put an index on from_address. Then we added value to the index for our second attempt at optimizing it.

Query 4

The query took 57.480 seconds to complete before our optimizations. This query we had many theories of ways to optimize it. First, we put an index on the WHERE part of the query. Next, we tried to get the number of transactions from joining the transaction and block tables. Then we realized that all the information being returned in this query was available in the blocks table, so we rewrote the query to be used purely on the blocks table, and then put an index on it to improve it further.

Query 5

This query took about 62 seconds to run before we attempted to optimize it. Our optimization attempt was to join transactions with blocks and get the number of transactions from the blocks table instead of counting.

Query 6

This was a fast query to start with only taking 1.9 seconds, but we wanted to try to improve it. Seeing as difficulty was used in all the aggregate functions, we started by putting an index on only difficulty. Our second optimization we attempted was to put the other two attributes, miner and number into the index.

Query 7

Our first optimization was to decrease the number of attributes that were being returned. So, we removed block_number and the hash. The other proposed optimization we had for this query was to put an index on the ORDER BY attribute.

Query 8

With the query taking over a minute to run each time, we wanted to optimize our favourite query a little bit. So, we decided to put two indices on the transactions table, one on from_address and one on to_address.

Query 9

This query took a long time considering how simple it seemed, so we believed that we could decrease the run time down from over 98 seconds. Our idea was to put an index on the block_timestamp.

Query 10

This query originally took over 100 seconds on average so we believed that we could decrease this significantly. Our attempt at optimization was to put an index on from_address and see if that would be better.

IV. ANALYSIS

Query 1

After creating an index on the from-address attribute, the average time went down to 6.408 seconds to return the results. The index on from-address reduced our query's run time by almost 94%.

Query 2

The index on the to-address attribute resulted in an 88% decrease in the time to return our results. The query only takes 6.657 seconds to run now.

Query 3

Our first attempt optimizing this query did not work out as we thought it would. We assumed since the query was grouped by from_address, an index on from_address would speed up the query. Instead, it added on average 40 seconds to each run time. So, we added the other attribute that the query returned, value, to the index and ran it again. This time we saw

the improvement that we expected, bringing the run time down to 7.154 seconds.

Query 4

The first optimization we attempted was to put an index on block_timestamp. This resulted in an 86% decrease in running time. But we knew we could improve the query even more. So, we tried to get the number of transactions per block by joining the two tables. In retrospect, we should've known this would've resulted in a larger number of tuples for our query to look through. This optimization backfired and ended up taking over a minute longer. Finally, realizing that all the information could be gathered from the blocks table alone. After changing the query to be only on the blocks table, the query now ran in just under 3 seconds. Knowing we could improve this further, we put an index on the timestamp of the block table. Bringing the run time down to a staggering 0.316 seconds. This is over a 99% improvement in run time.

Query 5

Attempting to optimize the query by joining transactions with blocks ended up not working as well as we thought. Joining the two tables together increased the number of tuples the query had to check, which lead to an increase of 10 seconds in run time.

Query 6

The first optimization of an index on difficulty didn't work out, its improvement was less than 1% which is not noticeable. We chalked this improvement up to being related to the conditions of the computer the queries were running on. The next optimization we tried was to add miner and number onto the index, this resulted in a improvement to 33% of the original time. Bringing the query down to 0.629 seconds.

Query 7

By removing block_number and hash from the selection part of our query, it improved the runtime by about 5%, but this was within our margin of error on original run times, so we discarded this improvement. We then went to the index optimization. By placing an index on the order by attribute, we were able to make the query basically instant. Taking only 10ms to return our information. This was by far the biggest improvement we saw from a query. This type of run time for a query was amazing to us. Being able to get information out of a 22.17GB table and have it output to us within 10ms, an unnoticeable amount of time, was amazing.

Query 8

Our optimizations of this query were underwhelming. We put an index on from_address and to_address thinking the WHERE condition would be sped up significantly, but it's run time was

only brought down by about 50% to the 30 second range.

Query 9

Our attempt at optimizing this query did not work. Putting an index on block_timestamp increased the time the query took to almost two minutes. This was unexpected which led us to analyze the query further. Looking at the query, it is ordered by an aggregate function. So, we think that is why the index did not improve the time since, like we learnt in class, aggregate functions generally require a table scan and maintaining running info.

Since the index didn't work to optimize the query, we did some analysis on how the query was structured.

It takes 3 attributes and does a table scan. For each row of data, the query converts the block_timestamp into a year and month.

The group by clause then groups the rows together if they have the same year and month. Since the time span is 20 months, there are 20 groups that need to be stored in memory. Assuming month and year are 2 bytes each, and the cumulative sum of transaction fees is 8 bytes (floating point), and the number of transactions to be 2 bytes. Each group will take up (2 + 2 + 8 + 2) = 14 bytes. For 20 groups, 20 x 14 = 280 bytes. Finally, within each group, the average transaction cost is calculated.

The order by clause then sorts these groups in descending order based on highest average transaction cost.

Query 10

To optimize our last query, we put an index on from_address. The optimization was slightly underwhelming, since it decreased the run time to 70 seconds, which is still unreasonably long for someone to wait for information. We think this because, like query 9, query 10 is ordered by an aggregate function causing the query to require a table scan.

V. DISCUSSION

One thing we wondered about was in our optimization for query 8. Since SQLite uses B+ trees for its indices and our WHERE condition was an equality. We think if we could do a hash index on the same two attributes would've resulted in a bigger improvement in runtime. If using the figures provided to us in class of an average of 1.2 I/Os for finding records using a hash index and 3 I/Os for a B+ tree, we think this could've brought the runtime of this query down further to about 12 seconds.

Since none of our queries had complex WHERE conditions on them, one thing we wondered about was trying to convert a query into conjunctive normal form and seeing what kind of improvement we could see like that. So, we ran an extra query.

The extra query we ran returned all the columns from transactions where the block number was in a

range, or the gas was in a range. We ran this query five times in the form of (A and B) or (C and D) and the average time was around 13.5 seconds. Then we converted the query into conjunctive normal form and ran it again. The average time decreased to around 13 seconds. Normally, I would assume that the difference could be disregarded and assume that with more trials the numbers would converge. But knowing that the DBMS converts the query into conjunctive normal form to begin with, this may be a real improvement that we saw here.

It is interesting to see how just changing how the query is written changes how fast the query takes to return results.

While analyzing the queries, we wondered how our system used buffer pages. Since our test computer has 16GB of RAM, and if 70% of the memory can be used as buffer space for the queries, we say that 11.2GB of memory is available to use as buffer pages. If each page is 4KB, then the amount of buffer pages we have is 16GB/4KB = 2,800,000 buffer pages. This is significantly larger than the number of buffer pages we used for examples in class.

Overall, we optimized seven out of our ten queries, with query 7 being the best example of our optimizations.

VI. REFERENCES

"Conjunctive normal form," Wikipedia, 23-Dec-2022. Available: https://en.wikipedia.org/wiki/Conjunctive_normal_form.

"Insight into the SQL server buffer cache," SQL Shack - articles about database auditing, server performance, data recovery, and more, 29-Oct-2020. Available: https://www.sqlshack.com/insight-into-the-sql-server-buffer-cache/.

M. A.I., "Ethereum blockchain," Kaggle, 03-Jul-2021. Available: https://www.kaggle.com/datasets/buryhuang/ethereum-blockchain.

SQLite, "Sqlite/sqlite: Official git mirror of the sqlite source tree," GitHub. Available: https://github.com/sqlite/sqlit