ADVANCED DB FINAL

Ahmet Furkan Kavraz - 150190024

This is the final exam report of the Advanced Database course. The question was about improving the query execution times of a given query set for a given database. The database contains lots of tables, however, our queries are related to only one table which is “Variants”. There are 4 different query types in the query set.

The first one is:
SELECT * FROM "Variants" WHERE "Chromosome" = 1 AND "Type" = 'SNP';
We observe that there are some equalities in the queries and we can use B+ Tree or Hash Indexes for this type of query.

The second one is:
SELECT * FROM "Variants" WHERE "Chromosome" = 1 AND "Type" = 'SNP' AND "GeneName" LIKE '%RRC%';
Among the previous issues we have some extra ‘like’ queries which are in type ‘%w%’. Solving the issues with these queries is a little bit hard. With research, I learned that there is a special index called the ‘Trigram’ index. I have heard these indexes during my term project research but that was ‘n-gram’ indexes. In PostgreSQL, they implemented the ‘Trigram’ index version of this.

The third one is:
The extension of a ‘Range’ selection. We can also use B+ Tree Indexes to overcome this problem.

The fourth and last one is:
The addition of another ‘like’ query with the type ‘%w%’. We can not use B+ Tree or Hash Indexes for this type of query. But if the query is in the ‘w%’ type, we might use B+ Tree, which is not valid for our case.

Apart from all of these, I also tried to use a combination of these indexes, transferring the dataset to MongoDB which is a NoSQL database, partitioning in PostgreSQL, and changing the resource of the PostgreSQL. I will mention all of these one by one.

Note: All the graphics are in terms of milliseconds.

Experiment Settings

I used my personal computer Huawei 14 which has an AMD Ryzen 7 CPU and 16 GB Ram during the experiments. I measured the execution times of the queries using 'EXPLAIN ANALYZE' for PostgreSQL and also the 'Explain Plan' section for MongoDB when measuring the query execution times.

Resource Controlling

It is suggested to set the 'work_mem' and 'maintanenance_work_mem' variables. 'work_mem' is used for controlling the resource for operations during query execution, and "maintanenance_work_mem' is used for controlling index-related things such as memory used when building indexes, etc. Neither of these two didn't change anything when I conduct my experiments. They might increase the execution times of other operations, however, nothing is changed in our case. The main reason is we don't perform join operations or our optimizations are not related to index building etc.

Transferring MongoDB

MongoDB is a schema-free key-value database which means that you can insert any data to a collection. In our case, we transferred our data using the below code. (The script is written by me)

```
import psycopg2
from pymongo import MongoClient

def connect_postgres():
    connection = psycopg2.connect(
        host="localhost",
        port="5432",
        database="adv_db",
        user="postgres",
        password="1234"
    )
    return connection

def connect_mongodb():
    client = MongoClient("mongodb://localhost:27017")
    return client

def insert_into_mongodb(connection, client):
    cursor = connection.cursor()
    cursor.execute("SELECT * FROM \"Variants\";")
    variants = cursor.fetchall()

    db = client["adv_db"]
    collection = db["Variants"]
    inserted_collection = []

    print("Inserting data into MongoDB...")
    count = 0

    for variant in variants:
        variant_data = {
            "Id": variant[0],
            "PositionStart": variant[1],
            "Length": variant[2],
            "Reference": variant[3],
            "Alternative": variant[4],
            "Type": variant[5],
            "GeneName": variant[6],
            "ClinicalSignificance": variant[7],
            "Disease": variant[8],
            "Synonimity": variant[9],
            "GeneRegion": variant[10],
            "Chromosome": variant[11]
        }
        inserted_collection.append(variant_data)
        count += 1
        if count % 10000 == 0:
            print(f"{count} records inserted into MongoDB.")

    collection.insert_many(inserted_collection)

    print("Data inserted into MongoDB successfully.")

# Connect to PostgreSQL
postgres_connection = connect_postgres()

# Connect to MongoDB
mongo_client = connect_mongodb()
```

```
# Insert data into MongoDB
insert_into_mongodb(postgres_connection, mongo_client)

# Close the connections
postgres_connection.close()
mongo_client.close()
```

After the transfer step, we also changed our queries to the proper type for MongoDB and perform our measurements. The script that is used for measurements in PostgreSQL will be mentioned in the next section. The below script is used for measurements in MongoDB (The script is written by me):

```
import json
from typing import List
from pymongo import MongoClient

def read_file(file_path: str) -> List[str]:
    with open(file_path, "r") as file:
        lines = file.readlines()
    return [line.strip("\n").strip() for line in lines]

def connect_mongodb() -> MongoClient:
    client = MongoClient("mongodb://localhost:27017")
    return client

def run_query(client: MongoClient, query: str) -> float:

    result =
client["adv_db"]["Variants"].find(json.loads(query)).explain()

    execution_time = result["executionStats"]["executionTimeMillis"]
    return execution_time

# get the query execution statistics from MongoDB
def get_stats(client: MongoClient, queries: List[str]) -> dict:
    query_multiplier = 1
    execution_times = []

    for q_index in range(len(queries)):
        if q_index % 20 == 0:
           print(f"\nStarting Query group {q_index // 20 + 1}\n")
        query = queries[q_index]
        query = query.replace("Variants", "Variants_partition")
        # print(f"Running query: {query}")
        for _ in range(query_multiplier):
            execution_time = run_query(client, query)
            execution_times.append(execution_time)

        if q_index % 20 == 19:
            # Print the average, minimum, and maximum execution times
            print("\n", end="")
            print(f"Minimum query execution time: {min(execution_times)}
milliseconds.")
            print(f"Median query execution time:
{sorted(execution_times)[len(execution_times) // 2]} milliseconds.")
            print(f"Average query execution time: {sum(execution_times) /
len(execution_times)} milliseconds.")
            print(f"Maximum query execution time: {max(execution_times)}
milliseconds.")
            print("\n", end="")

            execution_times = []


# Connect to MongoDB
mongo_client = connect_mongodb()

# Read the query statements
queries = read_file("query_mongo.txt")

# Get the query execution statistics
print("Running the query without indexes")
get_stats(mongo_client, queries)

mongo_client.close()
```

The experiment results for pure PostgreSQL table and pure MongoDB collection are represented in Figure 1 (pure means without any other modifying such as using an index):

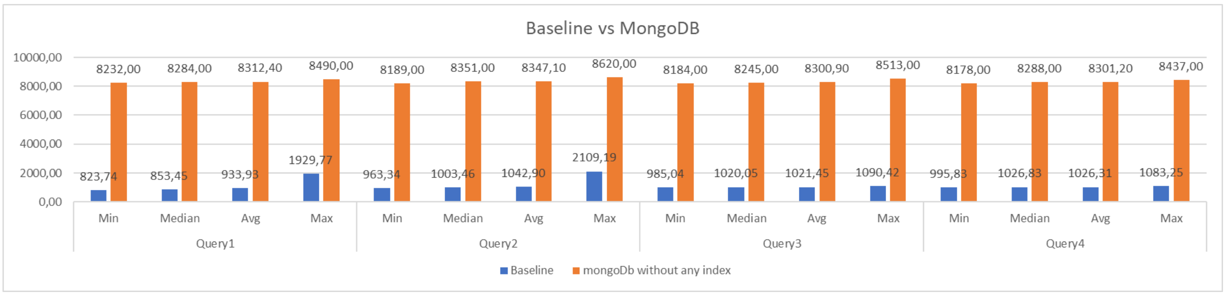

Figure 1: Baseline vs. MongoDB

It can be seen very clearly that the MongoDB underperforms when we compared it with PostgreSQL. However, if we make some modifications to the dataset, for example, using an index, it might be better. In Figure 2, you can see the result when we created an index on (“Chromesome”, “Type”, “PositionStart”) columns.

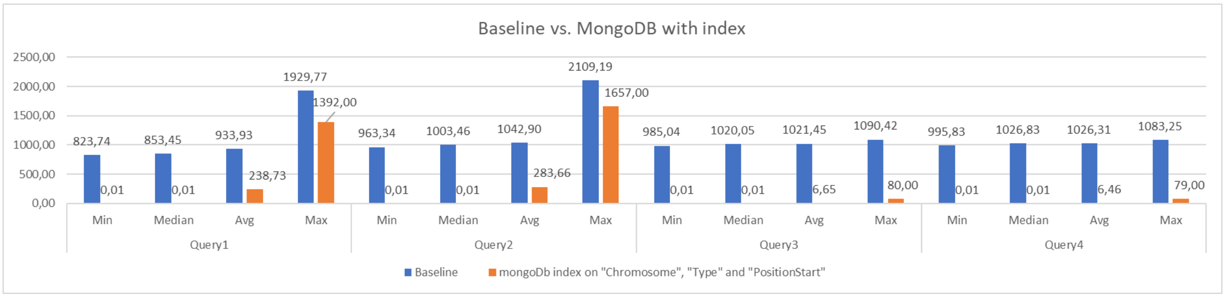

Figure 2: Baseline vs. MongoDB with Index

In the minimum cases, the queries are highly optimized, however, for the maximum cases, there is still lots of room for improvement. Also, you will see that these MongoDB results are not good enough when we compare the results with using indexes on PostgreSQL.

PostgreSQL with Indexes

I aimed to use lots of indexes such as B+ Tree, Hash, Trigram, and Block Range indexes. However, during the query analysis phase, I observed that using B+ Tree and Trigram indexes will be more meaningful, because of our query types.

To measure the results, again, I created a script to make the process automatic. The script first measures the baseline which we don’t create any index, and then measures the results by creating different indexes that are stated in the ‘indexes.txt’ file. It gives us the ability to measure results easily using different indexes or a combination of them.

```python
import psycopg2
from typing import List
from psycopg2.extensions import cursor, connection

def read_file(file_path: str) -> List[str]:
    with open(file_path, "r") as file:
        lines = file.readlines()
    return [line.strip("\n").strip() for line in lines]

def connect_database() -> connection:
    connection = psycopg2.connect(
        host="localhost",
        port="5432",
        database="adv_db",
        user="postgres",
        password="1234"
    )
    return connection

def run_query(cursor: cursor, query: str) -> float:

    cursor.execute("EXPLAIN ANALYZE " + query)
    execution_time = float(cursor.fetchall()[-1][0].split(" ")[2])
    return execution_time

def get_stats(cursor: cursor, queries: List[str]) -> None:
    query_multiplier = 3
    execution_times = []

    for q_index in range(len(queries)):
        if q_index % 20 == 0:
           print(f"\nStarting Query group {q_index // 20 + 1}")
        query = queries[q_index]
        for _ in range(query_multiplier):
            execution_time = run_query(cursor, query)
            execution_times.append(execution_time)

        if q_index % 20 == 19:

            # Print the average, minimum, and maximum execution times
            print("\n", end="")
            print(f"Minimum query execution time: {min(execution_times)}
milliseconds.")
            print(f"Median query execution time:
{sorted(execution_times)[len(execution_times) // 2]} milliseconds.")
            print(f"Average query execution time: {sum(execution_times) /
len(execution_times)} milliseconds.")
            print(f"Maximum query execution time: {max(execution_times)}
milliseconds.")
            print("\n", end="")

            execution_times = []


if __name__ == "__main__":
  # Read the index statements
  index_statements = read_file("indexes.txt")

  # Read the query statements
  queries = read_file("query.txt")

  # Connect to the database
  connection = connect_database()

  print( "Program started...\n")

  # Print the stats without indexes
  print("Running the queries without indexes")
  cursor = connection.cursor()
  get_stats(cursor, queries)
  cursor.close()
  connection.close()
  print("\n----------------------------------------\n")
  # Print the stats with indexes
  print("Running the queries with indexes")
  execution_times_without_indexes = []
  for indexes in index_statements:

    connection = connect_database()
    cursor = connection.cursor()
    print(f"Creating the index...")
    # Create the indexes
    for index in indexes.split(";"):
      index = index.strip()
      if index != "":
        print(f"Creating the index: {index}")
        cursor.execute(f"{index}")

    print("Running the queries")
    get_stats(cursor, queries)
    cursor.close()
    connection.close()
    print("\n----------------------------------------\n")
```

The first index that we created is Trigram Index in (“ClinicalSignifance”) column. The results, in Figure 3, show that it decreases execution times for the Query 4 set, but doesn’t have an effect for other types. Because the relevant column is not used in other Query sets.

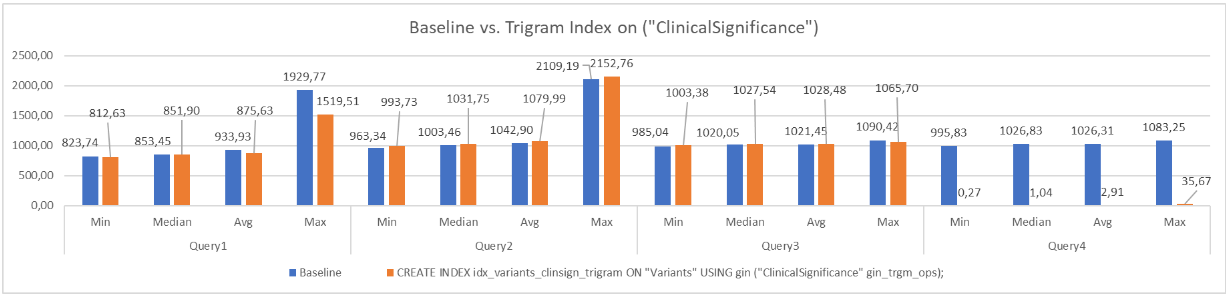

Figure 3: Baseline vs. Trigram index on (“ClinicalSignifance”)

Moreover, I created the same index on (“GeneName”) column which stands in Query 2, 3, and 4 sets, the results can be seen in Figure 4.

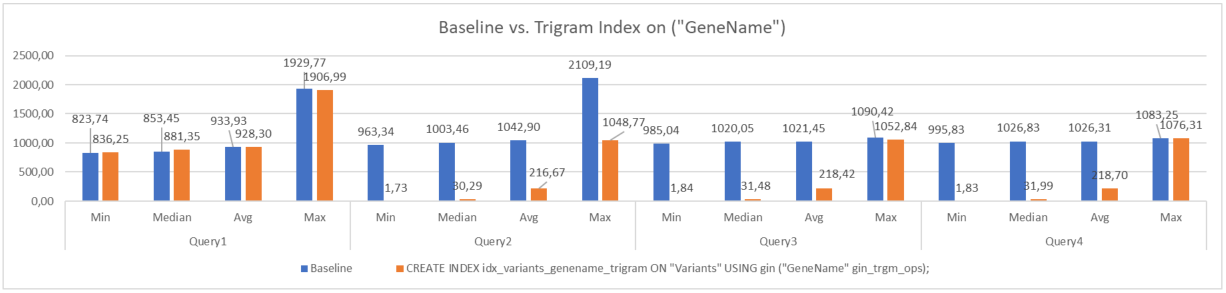

Figure 4: Baseline vs. Trigram index on (“GeneName”)

The result that I learned from these two experiments is Trigram indexes are highly useful when we are dealing with SQL LIKE queries with wildcards on both start and end. One important thing that needs to be mentioned is Trigram indexes are not supported built-in. So, I needed to install the relevant extension for that purpose using the “CREATE EXTENSION pg_trgm;” command.

After being sure that Trigram indexes are working, I used the B+ Tree index for the columns (“Choromesome”, “Type”, “PositionStart”). It is important that these are not different indexes, rather, they are one index on multiple columns. I tried to create this index on different combinations, but, the best one is the previously stated one, and no need to make an over-engineering for this, because it really depends on the parameters of queries.

You can see the results in Figure 5 (log scaled). B+ Tree Index also performs better than Trigram Indexes for Query 2, 3, and 4 sets. However, wait, is it the end? Did we find the best solution by only creating one index on multiple columns? This is a good solution but might not be the best. Also, using multiple of them may perform better results.

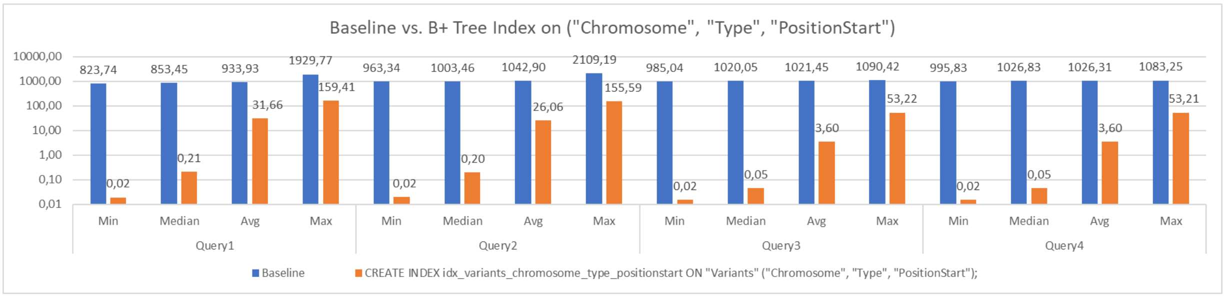

Figure 5: Baseline vs. B+ Tree Index on (“Chromosome”, “Type”, “PositionStart”)

Besides that the results are significantly decreased and very good when we compared it with the previous ones, we can still do better. The first thing is using all three indexes with each other on a table. The results can be seen in Figure 6 (log scaled). Most of the results are decreased from thousand milliseconds to one millisecond, where only there are some exceptions in Query 1 and 2 because the corresponding query returns so many rows.

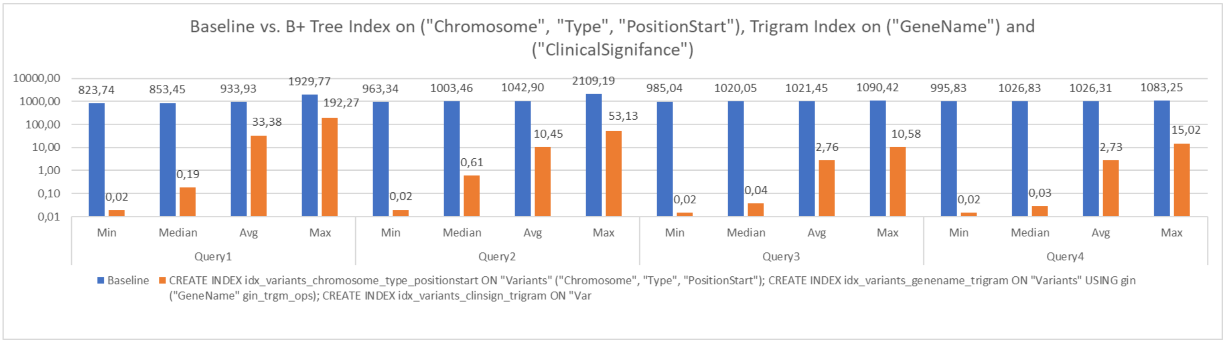

Figure 6: Baseline vs. B+ Tree Index on ("Chromosome", "Type", "PositionStart"), Trigram Index on ("GeneName") and ("ClinicalSignifance")

You can also see the comparison of the two previous methods in Figure 7 (log scaled), one is using only B+ Tree indexes on columns ("Chromosome", "Type", "PositionStart"), other is using Trigram Index on ("GeneName") and ("ClinicalSignifance") Indexes in addition to the B+ Tree index. Nothing is changed for Query 1, however, there is significant improvement when we used all three indexes.

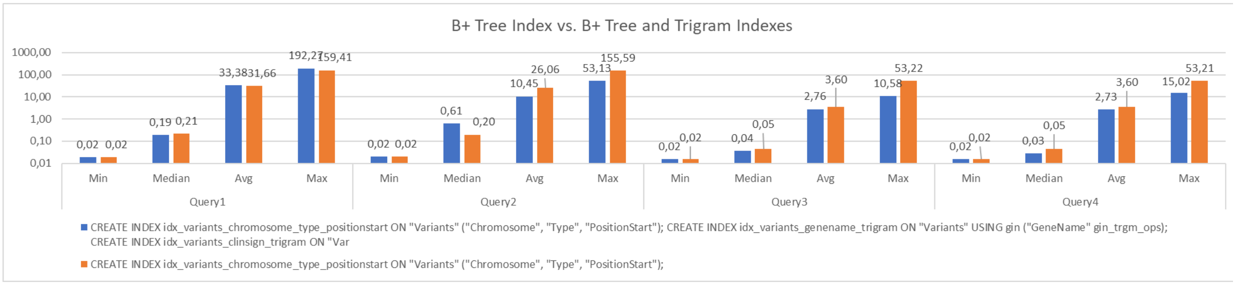

Figure 7: B+ Tree Index on ("Chromosome", "Type", "PositionStart"), Trigram Index on ("GeneName") and ("ClinicalSignifance") vs. B+ Tree Index on (“Chromosome”, “Type”, “PositionStart”)

Partitioning and Indexing on Partitions

Partitioning is a technique to cluster the data using some special rules. By doing this, we have the ability to restrict the area that will be looked for a query, if the query filters cover partitioned columns. In our case, I chose to partition the “Variants” table on the “Type” column, and also each partition table is partitioned on the “Chromosome” column. The nested partitioning is a little bit hard, so I created another script to automize this process.

The list partitioning method is used for “Type” because it is a string column, and the hash method is used for the “Chromosome” column. The main table is divided into 6, and each of them is divided into 24 partitions

```
import psycopg2

# Establish a connection to the PostgreSQL database
def getConnection():

  conn = psycopg2.connect(
          host="localhost",
          port="5432",
          database="adv_db",
          user="postgres",
          password="1234"
      )
  return conn

connection = getConnection()
cursor = connection.cursor()

print("Creating table Variants_partition")

cursor.execute("""
  CREATE TABLE public."Variants_partition"
  (
      "Id" bigint NOT NULL GENERATED BY DEFAULT AS IDENTITY ( INCREMENT 1 START 1 MINVALUE 1 MAXVALUE 9223372036854775807 CACHE 1 ),
      "PositionStart" bigint,
      "Length" integer,
      "Reference" text COLLATE pg_catalog."default",
      "Alternative" text COLLATE pg_catalog."default",
      "Type" text COLLATE pg_catalog."default",
      "GeneName" text COLLATE pg_catalog."default",
      "ClinicalSignificance" text COLLATE pg_catalog."default",
      "Disease" text COLLATE pg_catalog."default",
      "Synonimity" text COLLATE pg_catalog."default",
      "GeneRegion" text COLLATE pg_catalog."default",
      "Chromosome" integer
  ) PARTITION BY LIST ("Type");
  """)

print("Creating partitions")

type_list = ["CN", "DEL", "INDEL", "INS", "INV", "SNP"]

for type_el in type_list:
  create_table = f"""CREATE TABLE public."Variants_partition_{type_el}"
      PARTITION OF public."Variants_partition"
      FOR VALUES IN ('{type_el}')
      PARTITION BY HASH ("Chromosome");"""

  for i in range(0, 24):
    create_table += f"""
    CREATE TABLE public."Variants_partition_{type_el}_{i}"
      PARTITION OF public."Variants_partition_{type_el}"
      FOR VALUES WITH (MODULUS 24, REMAINDER {i});
    """

    print(f"Creating partition {type_el} {i}")
  cursor.execute(create_table)


cursor.execute(f"""INSERT INTO public."Variants_partition"
  SELECT * FROM public.\"Variants\";""")
connection.commit()
cursor.close()
connection.close()
```

The script that is used for query execution time analysis has just a little bit of difference from the previous method. The main difference is in the index creation part of each line in ‘indexes.txt’. Because, when we create an index in the main table, it is not mirrored to the partitioned table, but when we made a select, insert, or delete operation, they are redirected to the relevant partition. For each partition (24*6=144 partitions), every index is created and query execution times are measured with that method.

In Figure 8, the query execution times are compared for the base case and partitioned version without any indexes. It is obvious that it decreases the execution time, but we can do better by using indexes.

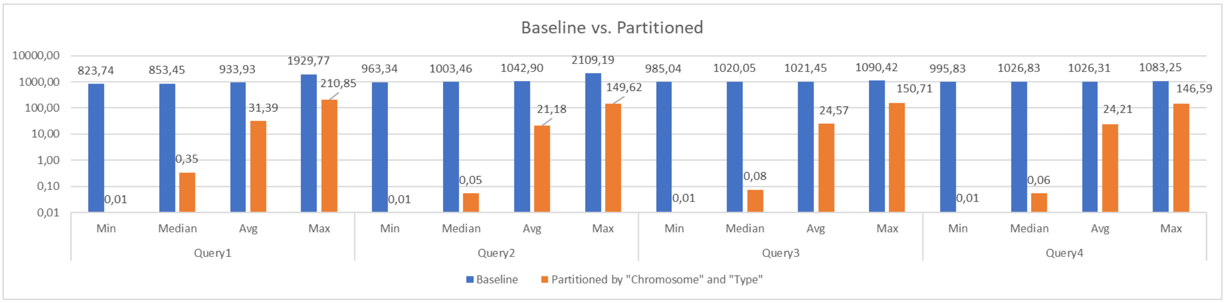

Figure 8: Baseline vs. Partitioned

When I applied the B+ Tree index on the column (“PositionStart”), I observed that the execution time of Query 3 and 4 sets decreased while Query 1 and 2 sets are not affected as expected. The results can be seen in Figure 9.

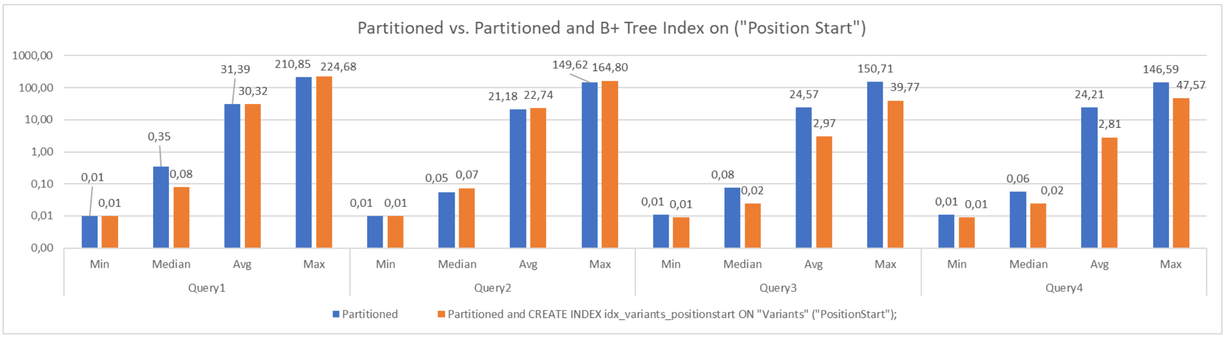

Figure 9: Partitioned vs. Partitioned and B+ Tree Index on (“PositionStart”)

Applying the Trigram index is changed a lot for Query 2, 3, and 4 sets. To decrease the cost of the Query 1 set, we actually can not do so much, since we already created partitioning on that columns, and using an index will not affect so much. The main overhead for Query 1 comes from the dataset size of the result. For other query sets the error rates are significantly decreased. The results can be seen in Figure 10 (figures are in log scale).

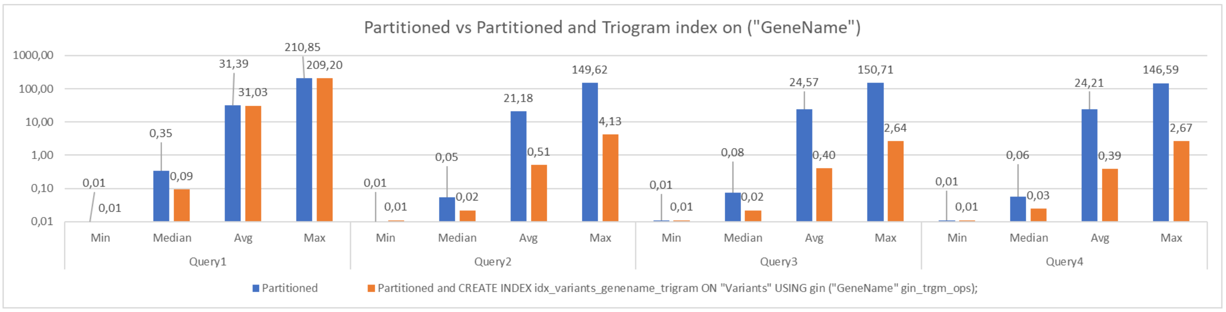

Figure 10: Partitioned vs. Partitioned and Trigram Index on (“GeneName”)

From the unpartitioned section, we knew that using the Trigram indexes for (“GeneName”) and (“ClinicalSignifance”), two separate indexes, decreases the execution time. So, no need to dive deep into them separately.

In Figure 11, you can see the query execution times for the Partitioned table with two Trigram indexes. It is best that we made for partitioning case, and I think, it is the best that we can actually do. Because, all columns have indexes on them, or they are part of the partition.

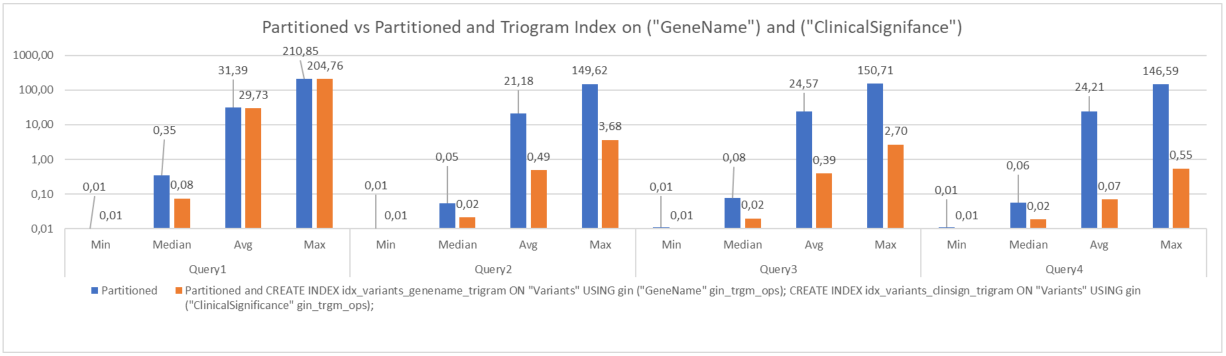

Figure 11: Partitioned vs. Partitioned and Trigram Index on (“GeneName”) and (“ClinicalSignifance”)

Actually, when we compared them with the baseline the execution times decreased a lot.

Interpreting Best Results

The best of the PostgreSQL with Indexes and Partitioning and Indexing on Partitions sections will be compared in this section. They will be also compared to the baseline, and the final results will be interpreted.

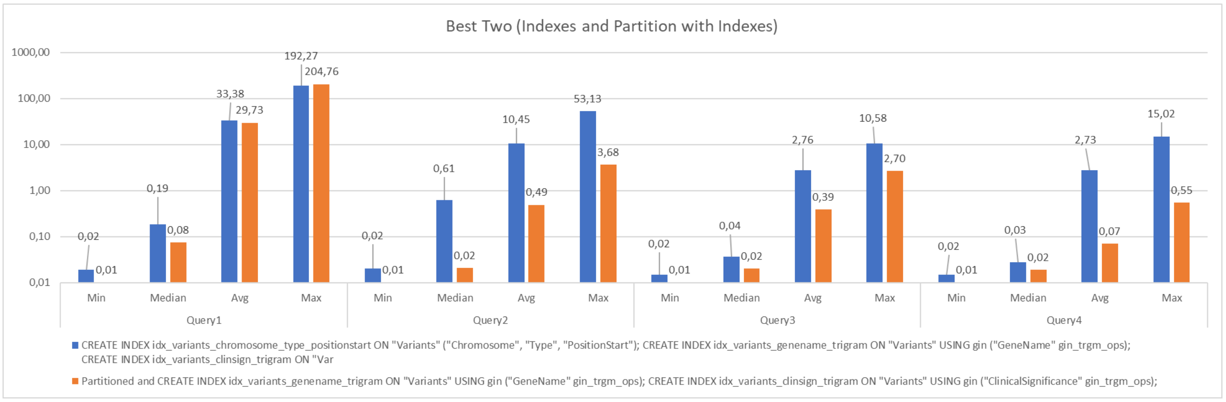

Figure 12: Best Two (Indexes and Partition with Indexes)

In Figure 12, the comparison of the best of two sections is represented. When we made partitioning, we are limiting the places that the data can be. So, after creating indexes on the remaining columns, the execution times will be decreased. However, it is obvious that we couldn’t make any improvement for the Query 1 set. The reason is with indexing or with partitioning using hashing, we did what is best and it is on the limit. The main reason the maximum Query 1 set execution time is pretty high is that there are some queries that cover around 500k rows. Interpreting these column one by one text so much time even though they don’t require any other operation. However, if we put an index, in the case of Query 2, we can significantly decrease the maximum query execution time. Because partitioning the table narrows down the scope of the data can be and prevents to use of two different indexes and getting the same results from them. In the second case, it intersects the result of two indexes.

In Figure 13, you can see the comparison of the Baseline and the best of the two sections. I didn't consider MongoDB because it is obvious that it can not compete with Best Two. According to these results, we improved the execution times of

- Query1
    - Min: 99.999%
    - Median: 99.99%
    - Average: 96.81%
    - Max: 89.39%
- Query2
    - Min: 99.999%
    - Median: 99.998%
    - Average: 99.953%
    - Max: 99.82%
- Query3
    - Min: 99.999%
    - Median: 99.998%
    - Average: 99.961%
    - Max: 99.75%
- Query4
    - Min: 99.999%
    - Median: 99.998%
    - Average: 99.993%
    - Max: 99.949%

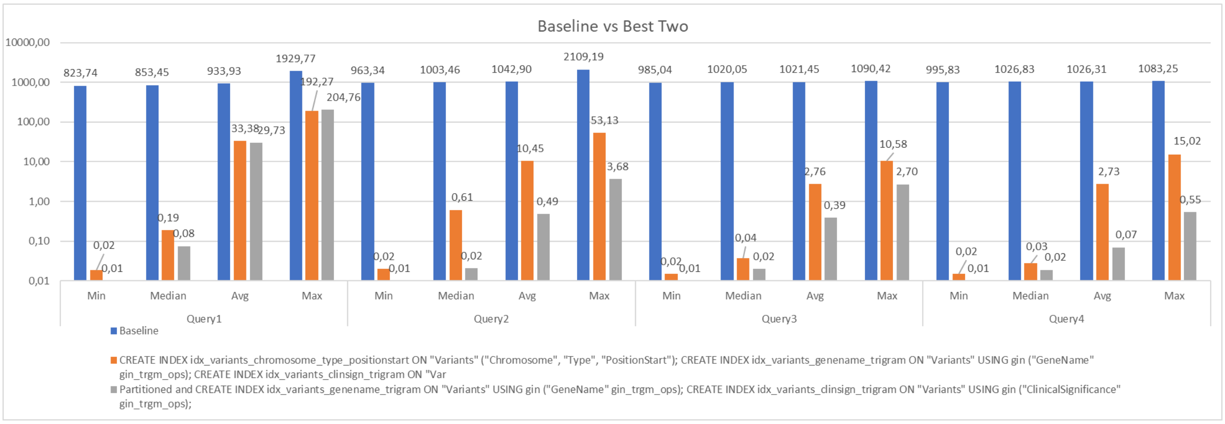

Figure 13: Baseline vs. Best Two

The actual results are interpreted as follows in the form of “from => to”:

- Query1
    - Min: 823.74 ms => 0.01 ms
    - Median: 853.45 ms => 0.08 ms
    - Average: 933.93 ms => 29.73 ms
    - Max: 1929.77 ms => 204.76 ms
- Query2
    - Min: 963.34 ms => 0.01 ms
    - Median: 1003.46 ms => 0.02 ms
    - Average: 1042.9 ms => 0.49 ms
    - Max: 2109.19 ms => 3.68 ms
- Query3
    - Min: 985.04 ms => 0.01 ms
    - Median: 1020.05 ms => 0.02 ms
    - Average: 1021.45 ms => 0.39 ms
    - Max: 1090.42 ms => 2.7 ms
- Query4
    - Min: 995.83 ms => 0.01 ms
    - Median: 1026.83 ms => 0.02 ms
    - Average: 1026.31 ms => 0.07 ms
    - Max: 1083.25 ms => 0.55 ms

Index Sizes Comparison

I measured the index sizes for the “Baseline” as follows:

- B+ Tree Index on columns (“Chromosome”, “Type”, “PositionStart”) uses 405.2MB
- Trigram Index on column (“ClinicalSignifance”) uses 13.76MB
- Trigram Index on column (“GeneName”) uses 208.01MB
- In total 906MB

While some of them use significant memory when we compared to total table size, which is 1.19GB, they helps us to increase the query execution times. However, when we look at the partitioning method, we see that it decreases memory consumption significantly. In partitioning, the index sizes are:

- The two Trigram Indexes use 281MB,
- There is an additional 20MB cost due to the partitioning.

As a result, it is obvious that the partitioning and indexing method is more effective on both total query execution time and memory consumption.

Conclusion

In the end, I saw the benefit of indexing and partitioning rather than managing the database resources and transferring the dataset to the NoSQL database. I created the partitioning for the columns that has equality condition, and Trigram indexes for SQL LIKE query selections in the form of “%w%”. In addition to this, we also tried to use B+ Tree indexes, however, the first approach is more useful than this. I achieved to significant decrease in the query execution times for all query types that are provided. I also created scripts to automize the process.

Resources

https://github.com/ahmetfurkankavraz/QueryOptimization
PostgreSQL, https://www.postgresql.org/
MongoDB, https://www.mongodb.com/