NoSQL Hospital Information System (HIS) benchmark report - Neo4j

Vittorio Romeo (mat. 444962)
Sergio Zavettieri (mat. 447265)

Università degli Studi di Messina
Dipartimento di Matematica e Informatica

Table of Contents

Chapter 1

Introduction

As part of the Databases II course, a group project consisting in the development of a benchmark for multiple NoSQL DBMSs was requested. All the DBMSs have to perform and time the same queries, in order to fairly compare their performance on a randomly-generated HIS database.

This report covers the Neo4j benchmark development and implementation.

1.1 Client request

The client requests an application that benchmarks the Neo4j NoSQL DBMS with randomly-generated data, in order to understand its performance with a specific schema *(used for an Hospital Information System)* and to see how the system's efficiency scales with an increasing data amount.

The desired schema is shown below:

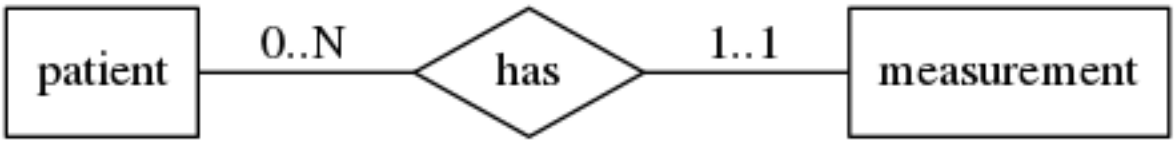

Figure 1.1: Client dataset schema

The requested application must import several randomly-generated datasets, and benchmark three different queries:

1. Select all patients.
2. Select all patients matching a specific condition.
3. Select all patients matching a specific condition with at least 5 measurements.

The generated datasets must be five and contain the following element amounts: `10`, `100`, `1000`, `10000`, `100000`.

After running the queries, the client requests histogram plots of the results, by repeating the same query `50` times.

1.2 Environment

The application was implemented using **Python 3.5.1** and the **Neo4j** NoSQL DBMS.

The following libraries were used as dependencies:

- **neo4j-rest-client**: a Python interface to interact with Neo4j.
- **matplotlib**: a Python library to easily create plots and graphs.

1.3 Architecture and implementation

Three different Python scripts were developed for the project:

- `load_dataset.py`, which, given a JSON dataset, loads it into the current Neo4j instance.
- `queries.py`, which runs the queries on the current Neo4j instance and outputs the query execution times to text files.
- `make_plots.py`, which reads the data outputted by the previous script and uses `matplotlib` to create histogram plots.

An helper bash script, `run_benchmarks.sh`, was developed to automate the benchmarking process.

1.3.1 Dataset loader

The dataset loader is conceptually very simple. The dataset path and a **chunk value** *(which represents the number of insertions to execute in batch)* are taken as command-line parameters.

A `master` class was created to wrap the Neo4j connection and functions to run the queries:

```
# Class containing an open neo4j connection
# and functions to manage the data
class master:
    # Defines a label `l` and stores it in the
    # labels list
    def define_label(self, l):
        self.labels[l] = self.db.labels.create(l)

    # Constructor
    # Given an open connection, stores the connection
    # in `master`
    # Defines labels for every entity
    def __init__(self, db):
        self.db = db
        self.labels = {}

    # Completely clears the database
    def delete_everything(self):
        q = 'MATCH (n) DETACH DELETE n'

        self.db.query(q)

        self.define_label("patient")
        self.define_label("measurement")

        tx = self.db.transaction(for_query=True)
        tx.append("CREATE INDEX ON :patient(id)")
        tx.execute()
        tx.commit()

    # Executes the string `x` as a query
    def do_query(self, x):
        tx = self.db.transaction(for_query=True)
        tx.append(x)
        tx.execute()
        tx.commit()
```

The rest of the application deals with dataset loading and query generation, using an

efficient string concatenation method thanks to the `StringBuilder` class:

```
# Helper class for efficient string concatenation
class StringBuilder(object):
    def __init__(self):
        self._stringio = io.StringIO()

    def __str__(self):
        return self._stringio.getvalue()

    def append(self, *objects, sep=' ', end=''):
        print(*objects, sep=sep, end=end, file=self._stringio)
```

The `main` function is as follows:

```
# Create a `master` and clear the database
m = master(make_connection("neo4j", "admin"))
m.delete_everything()

# Read dataset path from command line arguments
dataset_path = sys.argv[1]

# Read how many queries to batch per transaction
chunk_size = int(sys.argv[2])

# Read the dataset file as json
ds_patients = json.loads(open(dataset_path, "r").read())

# Index used to generate unique node names
idx = 0

# Execute all insertions
for i in range(0, len(ds_patients), chunk_size):
    q = StringBuilder()

    # Iterate patients in chunks
    for p in ds_patients[i:i + chunk_size]:
        # Stringify `idx`
        sidx = str(idx)

        # Generate patient node creation query
        q.append("CREATE (n")
        q.append(sidx)
        q.append(":patient {")
        q.append(make_patient_dict(p))
        q.append("})\n")

        # Generate measurement queries, which build relationships
        for s in p["step_datas"]:
            q.append("CREATE (n")
            q.append(sidx)
            q.append(")-[:measure]->(:measurement {")
            q.append(make_measurement_dict(s))
            q.append("})\n")

        # Increment next unique node id
        idx += 1

    m.do_query(str(q))
```

The `make_patient_dict` and `make_measurement_dict` functions efficiently build **Cypher** strings for the insertion of multiple parameters.

1.3.2 Query benchmarker

The second Python script, which runs the queries, benchmarks them and produces the graphs thanks to **matplotlib**, has a very straightforward implementation:

```
# Create a `master`
m = master(make_connection("neo4j", "admin"))

# Benchmark queries
bench_query('query0', '''
    MATCH (n:patient)
    RETURN n''')

bench_query('query1', '''
    MATCH (n:patient)
    WHERE n.n = "SIVV33W0"
    RETURN n''')

bench_query('query2', '''
    MATCH (p:patient)-[r:measure]->(m:measurement)
    WITH p, m, count(m) as relcount
    WHERE p.lwalk_td < 5000 AND p.w <> 5000 AND relcount > 4
    RETURN p''')
```

The `bench_query` functions is implemented as follows:

```
# Executes `q`, timing it and outputting results
def bench_query(lbl, q):
    # Perform queries and time them
    # Write results as newline-separated values
    for i in range(0, 30):
        start_timer()
        m.exec_query(q)
        print(end_timer())
```

The `start_timer` and `end_timer` functions make use of the `time.perf_counter()` Python high-precision timer in order to retrieve the execution time of every single query:

```
# Benchmark utilities
t0 = []
def start_timer():
    global t0
    t0.append(time.perf_counter())

def end_timer():
    global t0
    val = time.perf_counter() - t0.pop()
    return val
```

1.3.3 Plotting script

After having generated all dataset results in text files, where every query iteration execution time is written to a different line, the plotting script will take care of reading the files and producing histogram plots.

Given a list of dataset output paths `datasets` and a count of measurements per output, the `create_plot` is called for every query:

```
create_plot("query 0", datasets, 0, count, "plots/query0.png")
create_plot("query 1", datasets, 30, count, "plots/query1.png")
create_plot("query 2", datasets, 60, count, "plots/query2.png")
```

Its implementation is as follows:

```
def create_plot(plot_title, datasets, offset, count, output_path):
    # Iterate over the dataset benchmark outputs
    for dataset_path in datasets:
        # Get the first query execution time and
        # the remaining queries' average time
        first,avg = first_and_avg(dataset_path, offset, count)

        # Create two bars using pyplot
        b_first = plt.bar(x, first, 0.5, color='b')
        b_avg = plt.bar(x, avg, 0.5, color='r')

    # Plot to file
    plt.savefig(output_path)
    plt.clf()
```

The `first_and_avg` function simply loads the dataset timing results in memory and returns a tuple containing the first query time and the average time of the remaining

queries.

1.3.4 Automation script

In order to automate the whole benchmarking process, a simple **bash** script was implemented to load all datasets and execute the queries on them:

```
# Create `results` folder if required
mkdir -p results

# Dataset N array
VALUES=(10 100 1000 10000 100000)

# Load dataset chunk N array
CHUNKS=(1 1 1 5 10)

# Next chunk value index
ICHUNK=0

for i in "${VALUES[@]}"
do
    # Dataset path
    DS="../../dataset_lokomat/output/ds${i}.json"

    # Output graph path
    OF="./results/r${i}.png"

    # Load dataset
    python3 -O ./load_dataset.py "${DS}" "${CHUNKS[$ICHUNK]}"

    # Increment index for next chunk
    ((ICHUNK++))

    # Run queries and create plots
    python3 -O ./queries.py "${OF}"
done

# Create plots
python3 -O ./make_plots.py
```

The script, in short, simply runs the previously described Python scripts for every randomly-generated dataset, automatically passing the dataset path and a reasonable chunk value in every execution. When the datasets have been processed, the `make_plots` script is finally called to produce the histogram images.

Chapter 2

Conclusion

The results of the queries are provided as histograms. Every plot image represents a single query, over all datasets. Two bars are plotted per dataset size: the first bar represents the **execution time of the first query**, the second bar represents the **average execution time of the remaining queries**.

The **X axis** represents the size of the datasets.

The **Y axis** represents the execution time, in milliseconds.

2.1 Result graphs

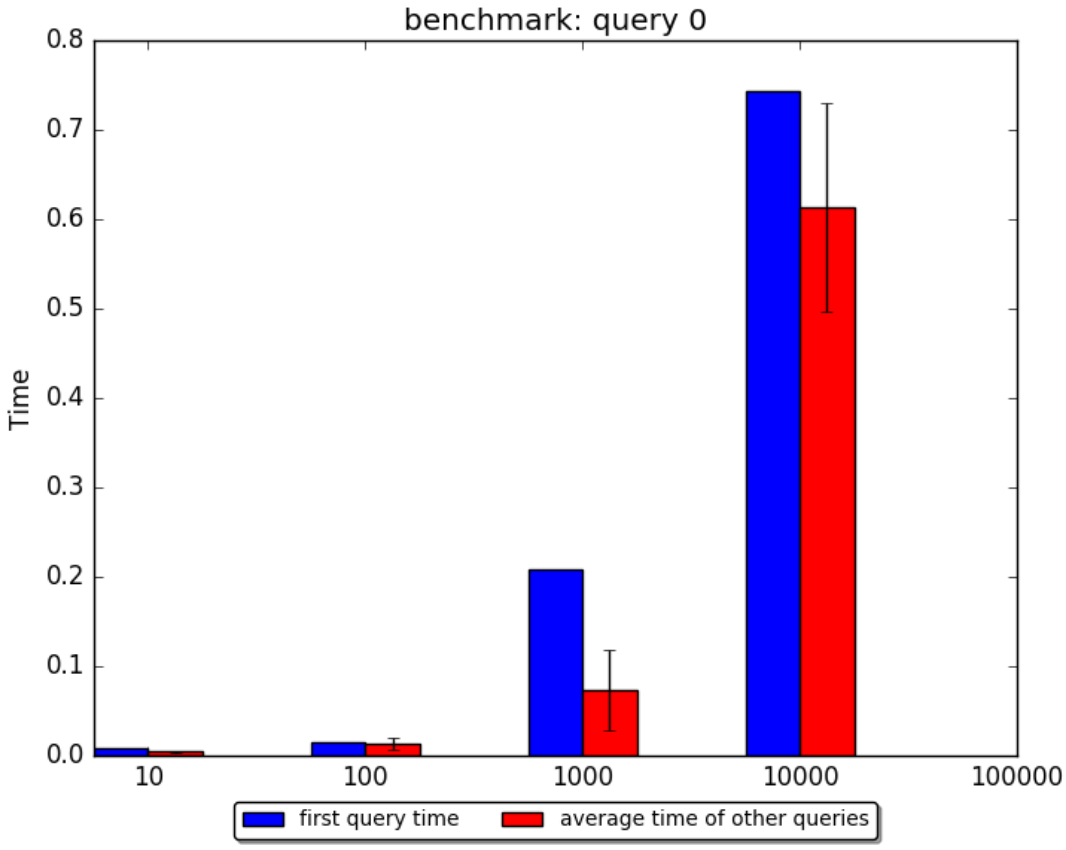

Figure 2.1: Benchmark results: query 0

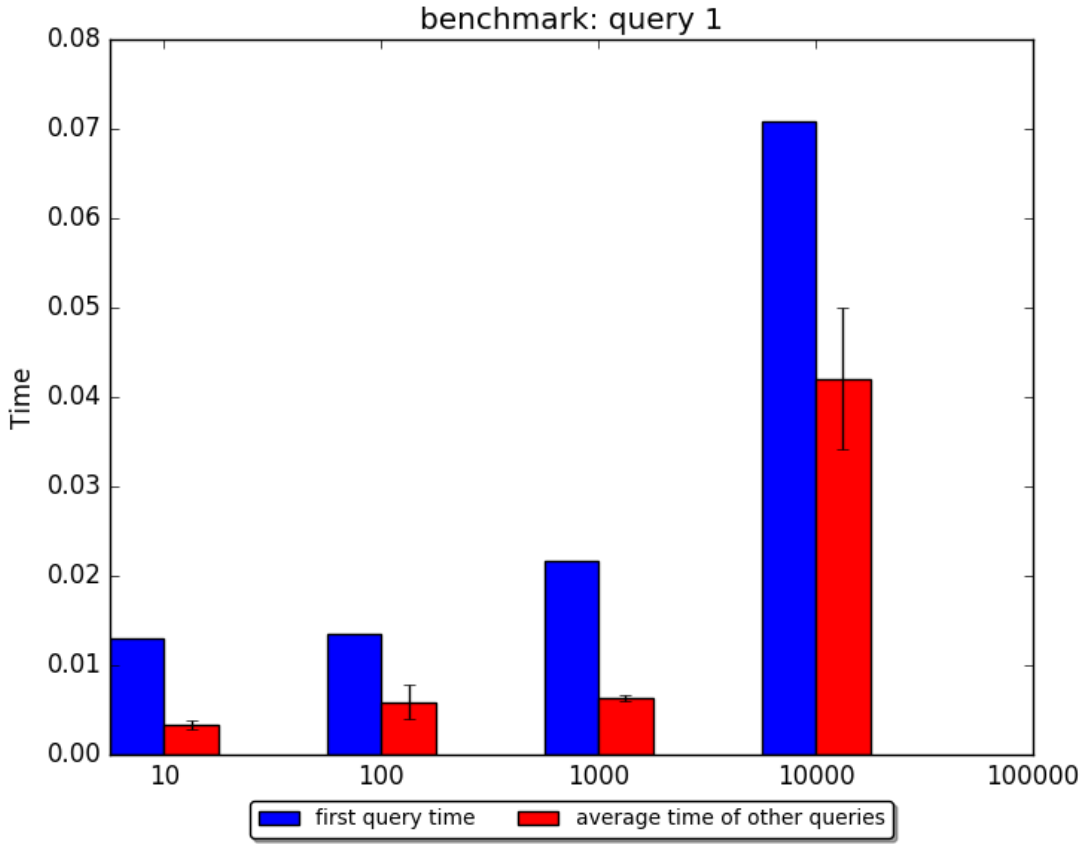

Figure 2.2: Benchmark results: query 1

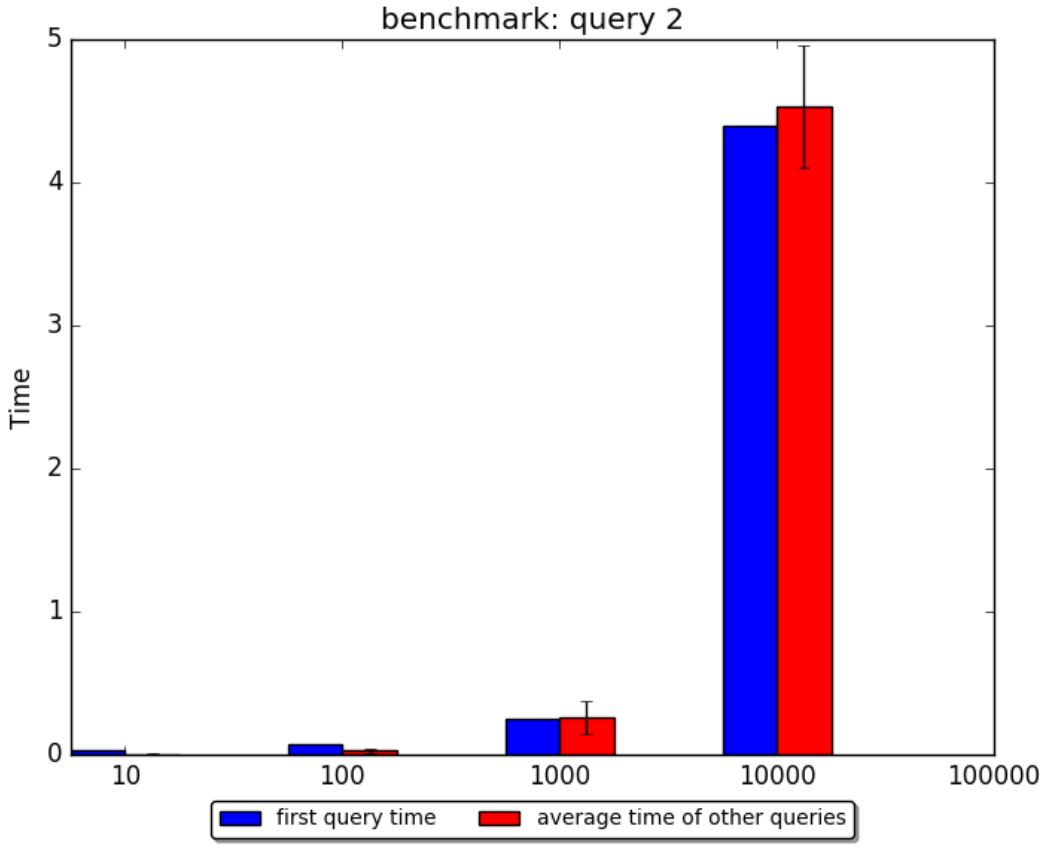

Figure 2.3: Benchmark results: query 2

Benchmarking 100000 patients was not feasible on the machine used for the other tests, due to extremely long Neo4j data loading times.

`query1` is the fastest for large dataset sizes, as the filter used to match patients is very strict. `query2`, due to the complex filtering rules, is always the slowest because it's necessary to iterate over the `measurement` nodes connected to the `patient` nodes.

2.2 Links

The project is available on GitHub: https://github.com/SuperV1234/db2.