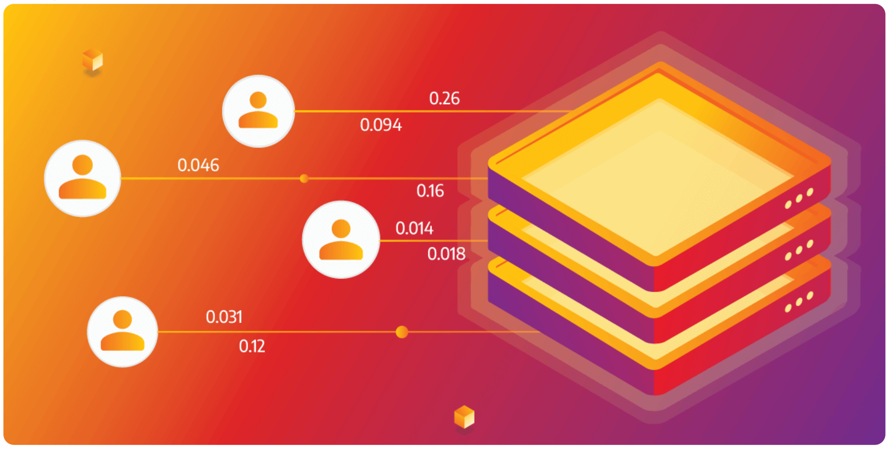
Product Use cases Neo4j vs Memgraph NetworkX Resources Docs Pricing Star 1,715 DOV

Not Much different to other blogs by this guy
except:
(1) State that Docker inflicts a 'Significant' performance overhead.
(2) Gives the benchmark process, including starting db, loading it, snapshot, restore from snapshot + query
3) Highlight that cold start (No caching etc) is worst-case + should be accounted for.



In this article

Introduction to Benchgraph and its Architecture

Ante Javor April 18, 2023

Topics: Under the Hood

Building a performant graph database comes with many different challenges, from architecture design, to implementation details, technologies involved, product maintenance — the list goes on and on. And the decisions made at all of these crossroads influence the database performance.

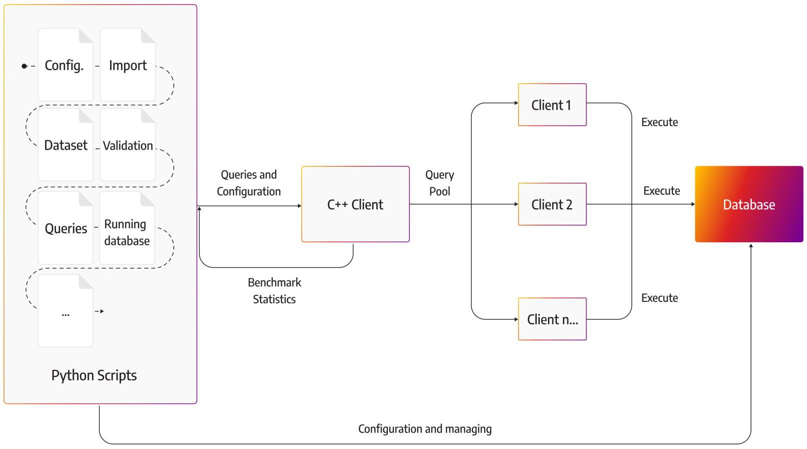
Share Blog

This is where **benchgraph** comes into play. To ensure the consistency of Memgraph's performance, tests are run via benchgraph, Memgraph's in-house benchmarking tool. On each commit, theCI/CD infrastructure runs performance tests to check how each code change influenced the performance. Let's take a look at benchgraph architecture.

Benchgraph architecture

At the moment, benchgraph is a project under Memgraph repository (previously Mgbench). It consists of Python scripts and a C++ client. Python scripts are used to manage the benchmark execution by preparing the workload, configurations, and so on, while the C++ client actually executes the benchmark.

System Design.



Some of the more important Python scripts are:

- benchmark.py - The main entry point used for starting and managing the execution of the benchmark. This script

- benchmark_context.py - It gathers all the data that can be configured during the benchmark execution.
- base.py - This is the base workload class. All other workloads are subclasses located in the workloads directory. For example, ldbc_interactive.py defines ldbc interactive dataset and queries (but this is NOT an official LDBC interactive workload). Each workload class can generate the dataset, use custom import ofthe dataset or provide a CYPHERL file for the import process.
- compare_results.py - Used for comparing results from different benchmark runs.

The C++ bolt benchmark client has its own set of features. It runs over the Bolt protocol and executes Cypher queries on the targeted database. It supports validation, time-dependent execution, and running queries on multiple threads.

Let's dive into benchmark.py, runners.py, and the C++ client a bit further.

benchmark.py

The benchmak.py script is filled with important details crucial for running the benchmark, but here is just a peek at a few of them.

All arguments are passed and interpreted in the benchmark.py script. For example, the following snippet is used to set the number of workers that will import the data and execute the benchmark:

```
        type=int,
        default=multiprocessing.cpu_count
        help="number of workers used to i
    )
    benchmark_parser.add_argument(
        "--num-workers-for-benchmark",
        type=int,
        default=1,
        help="number of workers used to e
    )
…
```

In the procedure that generates the queries for the workload, you can set up the seed for each specific query. As some arguments in the queries are generated randomly, the seed ensures that the identical sequence of randomly generated queries is executed.

```
def get_queries(gen, count):
    # Make the generator deterministic.
    random.seed(gen.__name__)
    # Generate queries.
    ret = []
    for i in range(count):
        ret.append(gen())
    return ret
```

The following methods define how warmup, the mixed, and realistic workload are executed and how the query count is

```
        ...

def mixed_workload(...):
          ...

def get_query_cache_count(...):
        ...
```

runners.py

The runners.py script manages the database, and the C++ client executes the benchmark. Runners script can manage both native Memgraph and Neo4j, as well as Docker Memgraph and Neo4j.

Vendors are handled with the following classes; of course, implementation details are missing:

```
        …
class MemgraphDocker(BaseRunner):
        …
 class Neo4jDocker(BaseRunner)
        …
```

Keep in mind that the Docker versions have a noticeable performance overhead compared to the native versions.

Interesting!

The C++ Bolt client executing benchmarks can also be managed in the native and Docker form.

```

class BoltClient(BaseClient):
        …
class BoltClientDocker(BaseClient):
        …
```

C++ Bolt client

The other important piece of code is the C++ bolt client. The client is used for the execution of the workload. The workload is specified as a list of Cypher queries. The client can simulate multiple concurrent connections to the

```
for (int worker = 0; worker < FLAGS_num_wc
    threads.push_back(std::thread([&, work
      memgraph::io::network::Endpoint endp
      memgraph::communication::ClientConte
      memgraph::communication::bolt::Clien
      client.Connect(endpoint, FLAGS_usern
…
```

By using multiple concurrent clients, the benchmark simulates different users connecting to the database and executing queries. This is important because you can simulate how your database handles higher data loads.

Each worker's latency values are collected during execution, and after that, some basic tail-latency values are calculated.

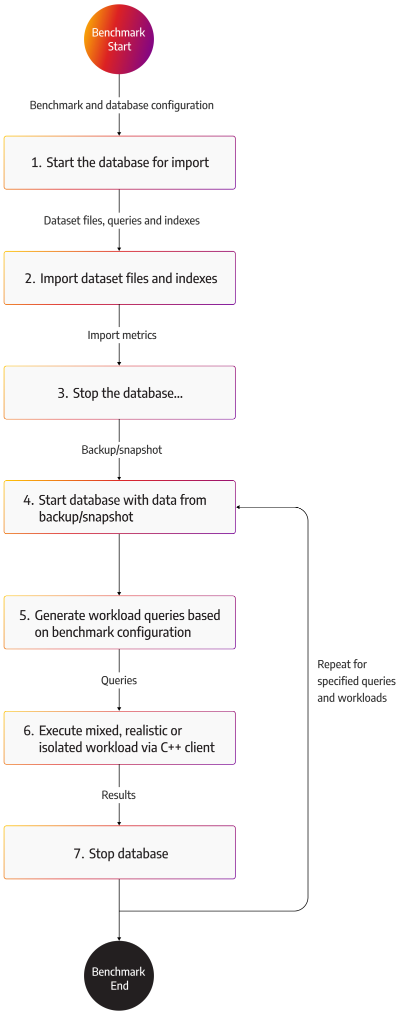
```
      query_latency.push_back(e);
    }
  }
  auto iterations = query_latency.size();
  const int lower_bound = 10;
  if (iterations > lower_bound) {
    std::sort(query_latency.begin(), quer
    statistics["iterations"] = iterations
    statistics["min"] = query_latency.frc
    statistics["max"] = query_latency.bac
    statistics["mean"] = std::accumulate(
    statistics["p99"] = query_latency[flc
    statistics["p95"] = query_latency[flc
    statistics["p90"] = query_latency[flc
    statistics["p75"] = query_latency[flc
    statistics["p50"] = query_latency[flc

  } else {
    spdlog::info("To few iterations to ca
    statistics["iterations"] = iterations
  }
…
```

For the rest of the details about all the components mentioned above and used in Memgraph’s CI/CD, feel free to refer to the code base.

Benchgraph benchmark process

methodology but for running a benchgraph. It's important to understand each step well because each step influences benchmark results. The image below shows the key steps in running a benchmark with Benchgraph:


Benchmark Start
Benchmark and database configuration
1. Start the database for import
Dataset files, queries and indexes
2. Import dataset files and indexes
Import metrics
3. Stop the database...
Backup/snapshot
4. Start database with data from backup/snapshot
5. Generate workload queries based on benchmark configuration
Queries
6. Execute mixed, realistic or isolated workload via C++ client
Results
7. Stop database
Repeat for specified queries and workloads
Benchmark End

Benchmark process

The first step is to start the database with predefined configuration options. Running Memgraph with different configuration options allows us to see respective performance implications.

different types of customer workloads quickly. After the import, the newly imported data is exported as a snapshot and reused to execute all the workloads that follow. But, if the first workload executed write queries, the dataset will be changed and therefore unusable because it will influence the results of the benchmark.

Once the import is finished, the database is stopped. Importing a dataset can stress the database and influence measurements, so the database must be restarted. After the restart, the database imports data from a snapshot, and the performance test can begin.

At the beginning of the test, queries are generated based on the benchmark configuration and the type of workload.

Benchgraph support three types of workloads:

- **Isolated** - Concurrent execution of a single type of query.
- **Mixed** - Concurrent execution of a single type of query mixed with a certain percentage of queries from a designated query group.
- **Realistic** - Concurrent execution of queries from write, read, update, and analyze groups.

Each of the workloads can be used to simulate a different production scenario.

Once the queries are executed, metrics such as basic latency measurements, queries per second, and peak RAM usage during the execution are collected and reported daily via our CI/CD infrastructure to graphana.

Just a side note on restarting a database during the execution of a benchmark: In an average use case, databases can run for a prolonged period of time. They are not being restarted very often, let's say, in some edge cases when you are doing some upgrades or are having issues. That being said, why are databases restarted during benchmarks? Executing test after test on the non-restarted database can lead to tests being influenced by previously run tests. For example, if you want to measure the performance of queries X and Y, they should be run under the same conditions, which means a database with a fresh dataset and without any caches.

Benchmark configuration options

Here are just some of the flags that can be used to configure benchgraph during the performance runs.

`--num-workers-for-benchmark` - This flag defines the number of workers that will be used to query the database. Each worker is a new thread that connects to Memgraph and executes queries. All threads share the same pool of queries, but each query is executed just once.

`--single-threaded-runtime-sec` - This flag defines the pool number of queries that will be executed in the benchmark. The question at hand is how many of each specific query you wish to execute as a sample for a database benchmark. Each query can take a different time to execute, so fixating a number, let's say 100 queries, could be finished in 1 second, and 100 queries of a different type could run for an hour. To avoid such an issue, this flag

`--warm-up` - The warm-up flag can take three different arguments, `cold`, `hot`, and `vulcanic`. Cold is the default. There is no warm-up being executed, `hot` will execute some predefined queries before the benchmark, while `vulcanic` will run the whole workload first before taking measurements. Databases in production environments are usually run pre-warmed and pre-cached since they are running for extended periods of time. But this is not always the case; warming up the database takes time, and caching can be ruined by the volatility of the data inside the database. Cold performance is the worst-case scenario for every database and should be considered.

– good point. Cold is worst case.

`--workload-realistic`, `--workload-mixed` - These flags are used to specify the workload type. By default, Benchgraph runs on an isolated workload.

`--time-depended-execution` - A flag for defining how long the queries will be executed in seconds. The same pool of queries will be re-run all over again until time-out. This is useful for testing caching capabilities of the database.

Why does Memgraph build an in-house benchmark tool?

Building benchmark infrastructure is time-consuming, and there are a few options for load-testing tools on the market. On top of that, building benchmarking infrastructure is error-prone; memories of unconfigured indexes and running benchmarks on debug versions of Memgraph are fun 😂

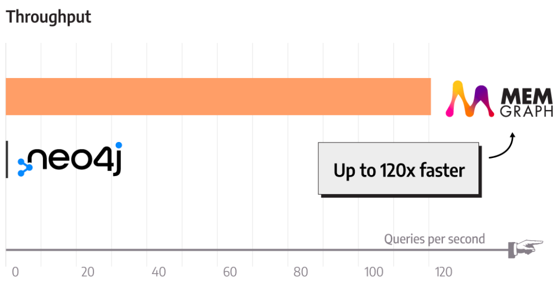
benchmarking needs. But in the long run, an internal benchmarking tool provides a lot of flexibility, and an external tool is a great addition as validation and support to all performance tests being executed.

Read next

Memgraph vs. Neo4j: A Performance Comparison

Memgraph delivers results up to 120 times faster than Neo4j while consuming one quarter of the memory!

November 30, 2022



Memgraph DB
Cloud
Lab
MAGE
GQLAlchemy

How it works
Use Cases
Pricing

Join us on Discord

Our growing community of graph enthusiasts awaits you!

JOIN OUR COMMUNITY

Resources

Docs
Playground
Community
Blog
Webinars
Email Courses
Code with Buda
Newsletter

Company

About Us
Careers
Teams
Legal
Partners
Press Room
Contact