

* Crit. Repo that Summarizes the Benchmarking Methodology.
to define private ops:
① Write
② Read
③ Update
④ Aggregate
⑤ Analyze [NFT]

They have 3 details (all same) that are S.M. and L.M. size (can run on all of Magnum's).

While simplistic, they DO support varying the.

In terms of benchmarking they have:
① Write set of varying size (but not clearly)
② Varying the.
③ Reference load - queries.
④ Compare Neo4j + Memgraph

GOOD STAFF POINT For or Benchmarking effort.

- They are the best protocol but don't examine anything (Needs to be F.O.A.E.E.L. to show the 100 example)
- Datasets are all (small)
- No statistical analysis of results is presented outside of index of counts.

<> Code Issues 163 Pull requests 35 Discussions Actions

master

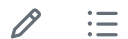
memgraph / tests / mgbench /

Josipmrden Add high write set prope... eb5167d · 4 days ago History

Name	Name	Last commit date
..		
cypher	Add mgbench tutorial ...	5 months ago
workloads	Add high write set pro...	4 days ago
.gitignore	Fix typo in mgbench	4 months ago
CMakeLists.txt	Initial implementation ...	3 years ago
Dockerfile.mgb...	Add mgbench tutorial ...	5 months ago
README.md	Flix methodology links...	5 months ago
benchgraph.sh	Add mgbench tutorial ...	5 months ago
benchmark.py	Fix a bunch of spelling...	2 months ago
benchmark_co...	Add mgbench tutorial ...	5 months ago
client.cpp	Fix a bunch of spelling...	2 months ago
compare_result...	Add mgbench tutorial ...	5 months ago
graph_bench.py	Add mgbench tutorial ...	5 months ago
helpers.py	Add bigger LDBC data...	6 months ago

📄 how_to_use_be...	Add mgbench tutorial ...	5 months ago
📄 log.py	Add mgbench tutorial ...	5 months ago
📄 runners.py	Add mgbench tutorial ...	5 months ago
📄 setup.py	Add mgbench tutorial ...	5 months ago
📄 validation.py	Add mgbench tutorial ...	5 months ago
📄 zip_benchgrap...	Add mgbench tutorial ...	5 months ago

README.md



🔥 Benchgraph: Benchmark for graph databases

📋 Benchmark Overview











Benchgraph is primarily designed to benchmark graph databases (Currently, Neo4j and Memgraph). To test graph database performance, this benchmark executes Cypher queries that can write, read, update, aggregate, and analyze dataset present in database. There are some predefined queries and dataset in Benchgraph. The present datasets and queries represent a typical workload that would be used to analyze any graph dataset and are pure Cypher based. BenchGraph shows the results of running these queries on specified hardware and under certain conditions. It shows the overall performance of each system under test relative to other, best in test being the baseline.

There is also a [tutorial on how to use benchgraph](#) to define your own dataset and queries, and run them on supported vendors. If you are interested in running benchmarks on your data, read tutorial, otherwise if you wish to run and validate results from Benchgraph read on.

Primitives?

analyze what?

This methodology is designed to be read from top to bottom to understand what is being tested and how, but feel free to jump to parts that interest you.

-  **Benchgraph: Benchmark for graph databases**
 -  **Benchmark Overview**
 -  **Design goals**
 - Reproducibility and validation
 - Database compatibility
 - Workloads
 - Fine-tuning
 - Limitations
 -  **Benchgraph**
 - Important files
 - Prerequisites
 - Running the benchmark
 - Database conditions
 - Comparing results
 -  **Results**
 -  **Datasets**
 - Pokec
 - Query list
 -  **Platform**
 - Intel - HP
 -  **Supported databases**
 - Database notes
 -  **Contributions**
 -  **History and Future of Benchgraph**
 - History of Benchgraph
 - Future of Benchgraph
 - **Changelog benchgraph**

Design goals

Reproducibility and validation

Good.

1

Running this benchmark is automated, and the code used to run benchmarks is publicly available. You can [run benchgraph](#) with settings specified under to validate the results at [BenchGraph](#). The results may differ depending on the hardware, benchmark run configuration, database configuration, and other variables involved in your setup. But if the results you get are significantly different, feel free to [open a GitHub issue](#).

In the future, the project will be expanded to include more platforms to see how systems perform on different OS and hardware configurations. If you are interested in what will be added and tested, read the section about [the future of Benchgraph](#)

🔗 Database compatibility

At the moment, support for graph databases is limited. To run the benchmarks, the graph database must support Cypher query language and the Bolt protocol.

Using Cypher ensures that executed queries are identical or similar as possible on every supported system. A single C++ client queries database systems (Currently, Neo4j and Memgraph), and it is based on the Bolt protocol. Using a single client ensures minimal performance penalties from the client side and ensures fairness across different vendors.

If your database supports the given requirements, feel free to contribute and add your database to Benchgraph. If your database does not support the mentioned requirements, follow the project because support for other languages and protocols in graph database space will be added.

🔗 Workloads

They should measure the impact of the bolt protocol.

Running queries as standalone units is simple and relatively easy to measure, but vendors often apply various caching and pre-aggregations that influence the results in these kinds of scenarios. Results from running single queries can hint at the database's general performance, but in real life, a database is queried by multiple clients from multiple sides. That is why the Benchgraph client supports the consecutive execution of various queries.

Concurrently writing, reading, updating and executing aggregational and analytical queries provides a better view of overall system performance than executing and measuring a single query. Queries that the Benchgraph executes are grouped into 5 groups - write, read, update, aggregate and analytical.

The [BenchGraph platform](#) shows results made by Benchgraph by executing three types of workloads:

- **Isolated workload**
- **Mixed workload**
- **Realistic workload**

Each of these workloads has a specific purpose:

Isolated workload is the simplest test. An isolated workload goes through all the queries individually, concurrently executing a single query a predefined number of times. It is similar to executing a single query and measuring time but more complex due to concurrency. How many times a specific query will be executed depends on the approximation of the query's latency. If a query is slower, it will be executed fewer times, if a query is faster, it will be executed more times. The approximation is based on the duration of execution for several concurrent threads, and it varies between vendors. If a query takes arguments, the argument value is changed for each execution. Arguments are generated non-randomly, so each vendor gets the same sequence of queries with the same arguments. This enables a deterministic workload for both vendors. The good thing about isolated workload is that it yields a better picture of single query performance. There is also a negative side, executing the same queries multiple times can trigger strong results caching on the vendor's side, which can result in false query times.

Mixed workload executes a fixed number of queries that read, update, aggregate, or analyze the data concurrently with a certain percentage of write queries because writing from the database can prevent aggressive caching and thus represent a more realistic performance of a single query.

The negative side is that there is an added influence of write performance on the results. Currently, Benchgraph client does not support per-thread performance measurements, but this will be added in future iterations.

Realistic workload represents real-life use cases because queries write, read, update, and perform analytics in a mixed ratio like they would in real projects. The test executes a fixed number of queries, the distribution of which is defined by defining a percentage of queries performing one of four operations. The queries are selected non-randomly, so the workload is identical between different vendors. As with the rest of the workloads, all queries are executed concurrently.

🔗 Fine-tuning

Each database system comes with a wide variety of possible configurations. Changing each of those configuration settings can introduce performance improvements or penalties. The focus of this benchmark is "out-of-the-box" performance without fine-tuning with the goal of having the fairest possible comparison. Fine-tuning can make some systems perform magnitudes faster, but this makes general benchmark systems hard to manage because all systems are configured differently, and fine-tuning requires vendor DB experts.

Some configurational changes are necessary for test execution and are not considered fine-tuning. For example, configuring the database to avoid Bolt client login is valid since the tests are not performed under any type of authorization. All non-default configurations are mentioned in [database notes](#)

🔗 Limitations

Benchmarking different systems is challenging because the setup, environment, queries, workload, and dataset can benefit specific database vendors. Each vendor may have a particularly strong use-case scenario. This benchmark aims to be neutral and fair to all database vendors.

Acknowledging some of the current limitations can help understand the issues you might notice:

1. **Benchmark measures and tracks just a tiny subset of everything that can be tracked and compared during testing.** Active benchmarking is strenuous because it requires a lot of time to set up and validate. Passive benchmarking is much faster to iterate on but can have a few bugs.
2. **The scale of the dataset** used is miniature for production environments. Production environments can have up to trillions of nodes and edges.
3. All tests are performed on **single-node databases**.
4. Architecturally different systems can be set up and measured biasedly.

Benchmarkgraph

Important files

Listed below are the main scripts used to run the benchmarks:

- `benchmark.py` - The main entry point used for starting and managing the execution of the benchmark. This script initializes all the necessary files, classes, and objects. It starts the database and the benchmark and gathers the results.
- `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 of the dataset or provide a CYPHERL file for the import process..
- `runners.py` - The script that configures, starts, and stops the database.
- `client.cpp` - Client for querying the database.
- `graph_bench.py` - Script that starts all tests from Benchmarkgraph.

- `compare_results.py` - Script that visually compares benchmark results.

Except for these scripts, the project also includes query files, dataset files and index configuration files. Once the first test is executed, those files can be located in the newly generated `.cache` and `.temp` folders.

🔗 Prerequisites

To execute a Benchgraph benchmark and validate results, you need to compile Memgraph and benchmark C++ bolt client from source, for more details on compilation process, take a look into this [guide](#). For Neo4j just download a binary version of Neo4j database you want to benchmark. Python version 3.7 and above is requirement for running benchmarks. Each database vendor can depend on external dependencies, such as Cmake, JVM, etc., so make sure to check specific vendor prerequisites during compilation process or running requirements.

🔗 Running the benchmark

To run benchmarks, you can use the `graph_bench.py`, which calls all the other necessary scripts. You can start the benchmarks by executing the following command:

```
graph_bench.py
--vendor memgraph /home/memgraph/binary
--dataset-group basic
--dataset-size small
--realistic 500 30 70 0 0
--realistic 500 50 50 0 0
--realistic 500 70 30 0 0
--realistic 500 30 40 10 20
--mixed 500 30 0 0 0 70
```



Isolated workload are always executed, and this commands calls for the execution of four realistic workloads with different distribution of queries and one mixed workload on a small size dataset.

Look @
Datasets.
↓
SIO is 2.18
gigs → TNY

The distribution of queries from write, read, update and aggregate groups are defined in percentages and stated as arguments following the `--realistic` or `--mixed` flags.

In the example of `--realistic 500 30 40 10 20` the distribution is as follows:

- 500 - The number of queries to be executed.
- 30 - The percentage of write queries to be executed.
- 40 - The percentage of read queries to be executed.
- 10 - The percentage of update queries to be executed.
- 20 - The percentage of analytical queries to be executed.

For `--mixed` workload argument, the first five parameters are the same, with an addition of a parameter for defining the percentage of individual queries.

Feel free to add different configurations if you want. Results from the above benchmark run are visible on [BenchGraph platform](#)

The other option is to use `benchgraph.sh` that can execute all benchmarks for each of specified number of workers.

🔗 Database conditions

In a production environment, database query caches are usually warmed from usage or pre-warm procedure to provide the best possible performance. Each workload in Benchgraph will be executed under the following conditions:

- **Hot run** - before executing any benchmark query and taking measurements, a set of defined queries is executed to pre-warm the database.
- **Cold run** - no warm-up was performed on the database before taking benchmark measurements.
- **Vulcanic run** - The workload is executed twice. The first time is used to pre-warm the database, and the second time is used to take measurements. The workload does not change between the two runs.

The details specification of warmup procedure is visible in the `benchmark.py` file, `warmup` function.

🔗 Comparing results

Once the benchmark has been run for a single vendor, all the results are saved in appropriately named `.json` files. A summary file is also created for that vendor and it contains all results combined. These summary files are used to compare results against other vendor results via the `compare_results.py` script:

```
compare_results.py
--compare
"path_to/neo4j_summary.json"
"path_to/memgraph_summary.json"
--output neo4j_vs_memgraph.html
--different-vendors
```



The output is an HTML file with the visual representation of the performance differences between two compared vendors. The first passed summary JSON file is the reference point.

🔗 Results

Results visible in the HTML file or at [BenchGraph](#) are throughput, memory, and latency. Database throughput and memory usage directly impact database usability and cost, while the latency of the query shows the base query execution duration.

Throughput directly defines how performant the database is and how much query traffic it can handle in a fixed time interval. It is expressed in **queries per second**. In each concurrent workload, execution is split across multiple clients. Each client executes queries concurrently. The duration of total execution is the sum of all concurrent clients' execution duration in seconds. In Benchgraph, the total count of executed queries and the total duration defines throughput per second across concurrent execution.

Here is the code snippet from the client, that calculates *throughput* and metadata:

```
// Create and output summary.
Metadata final_metadata;
uint64_t final_retries = 0;
double final_duration = 0.0;
for (int i = 0; i < FLAGS_num_workers; ++i) {
    final_metadata += worker_metadata[i];
    final_retries += worker_retries[i];
    final_duration += worker_duration[i];
}

auto total_time_end = std::chrono::steady_clock::now();
auto total_time = std::chrono::duration_cast<std::chrono:

final_duration /= FLAGS_num_workers;
nlohmann::json summary = nlohmann::json::object();
summary["total_time"] = total_time.count();
summary["count"] = queries.size();
summary["duration"] = final_duration;
summary["throughput"] = static_cast<double>(queries.size(
summary["retries"] = final_retries;
summary["metadata"] = final_metadata.Export();
summary["num_workers"] = FLAGS_num_workers;
summary["latency_stats"] = LatencyStatistics(worker_query
(*stream) << summary.dump() << std::endl;
```

Memory usage is calculated as **peak RES** (resident size) memory for each query or workload execution within Benchgraph. The result includes starting the database, executing the query/workload, and stopping the database. The peak RES is extracted from process PID as VmHVM (peak resident set size) before the process is stopped. The peak memory usage defines the worst-case scenario for a given query or workload, while on average, RAM footprint is lower. Measuring RES over time is supported by `runners.py`. For each vendor, it is possible to add RES tracking across workload execution, but it is not reported in the results.

Latency is calculated during the execution of each workload. Each query has standard query statistics and tail latency data. The result includes query execution times: max, min, mean, p99, p95, p90, p75, and p50 in seconds. Here is the code snippet that calculates latency:

```
...
std::vector<double> query_latency;
for (int i = 0; i < FLAGS_num_workers; i++) {
    for (auto &e : worker_query_latency[i]) {
        query_latency.push_back(e);
    }
}
auto iterations = query_latency.size();
const int lower_bound = 10;
if (iterations > lower_bound) {
    std::sort(query_latency.begin(), query_latency.end());
    statistics["iterations"] = iterations;
    statistics["min"] = query_latency.front();
    statistics["max"] = query_latency.back();
    statistics["mean"] = std::accumulate(query_latency.begin(),
    statistics["p99"] = query_latency[floor(iterations * 0.99)];
    statistics["p95"] = query_latency[floor(iterations * 0.95)];
    statistics["p90"] = query_latency[floor(iterations * 0.9)];
    statistics["p75"] = query_latency[floor(iterations * 0.75)];
    statistics["p50"] = query_latency[floor(iterations * 0.5)];
}
...
```

Each workload and all the results are based on concurrent query execution. As stated in [limitations](#) section, Benchgraph tracks just a subset of resources, but the chapter on [Benchgraph future](#) explains the expansion plans.

Datasets

Before workload execution, appropriate dataset indexes are set. Each vendor can have a specific syntax for setting up indexes, but those indexes should be schematically as similar as possible.

After each workload is executed, the database is cleaned, and a new dataset is imported to provide a clean start for the following workload run. When executing isolated and mixed workloads, the database is also restarted after executing each query to minimize the impact on the following query execution.

🔗 Pokec

The **Slovenian social network**, Pokec is available in three different sizes, small, medium, and large.

- **small** - vertices 10,000, edges 121,716
- **medium** - vertices 100,000, edges 1,768,515
- **large** - vertices 1,632,803, edges 30,622,564.

Dataset is imported as a CYPHERL file of Cypher queries. Feel free to check dataset links for complete Cypher queries.

Index queries for each supported vendor can be downloaded from

"https://s3.eu-west-1.amazonaws.com/deps.memgraph.io/dataset/pokec/benchmark/vendor_name.cypher", just make sure to use the proper vendor name such as `memgraph.cypher`.

🔗 LDBC Interactive

The LDBC interactive dataset is a **social network dataset** has support for multiple sizes, currently supported are sf01, sf1, sf3 and sf10. Keep in mind that bigger datasets will take longer to import and execute queries. The dataset is available in the following sizes:

- **sf01** - vertices 327,588 edges 1,477,965
- **sf1** - vertices 3,181,724, edges 17,256,038
- **sf3** - vertices 9,281,922, edges 52,695,735

Dataset is imported as a CYPHERL file of Cypher queries. Feel free to check dataset links for complete Cypher queries. Keep in mind that the dataset is imported differently for each vendor. For example, Memgraph uses Cypher queries to import the dataset, while Neo4j uses `neo4j-admin import` tool.

order of
magnitude for
each I guess...
Still small
? only
Social Media
Data
↓
Lacks
Diversity.

Index queries for each supported vendor can be downloaded from

"[https://s3.eu-west-](https://s3.eu-west-1.amazonaws.com/deps.memgraph.io/dataset/ldbc/benchmark/vendor_name.cypher)

[1.amazonaws.com/deps.memgraph.io/dataset/ldbc/benchmark/vendor_name.cypher](https://s3.eu-west-1.amazonaws.com/deps.memgraph.io/dataset/ldbc/benchmark/vendor_name.cypher)", just make sure to use the proper vendor name such as

```
memgraph.cypher
```

More details about the dataset in the [Interactive workload class](#)

DISCLAIMER: This is NOT an official implementation of an LDBC Benchmark.

🔗 LDBC Bussines Intelligence

The LDBC business intelligence dataset is a **social network dataset** has support for multiple sizes, currently supported are sf1, sf3 and sf10. Keep in mind that bigger datasets will take longer to import and execute queries. The dataset is available in the following sizes:

- **sf1** - vertices 2,997,352 edges 17,196,776
- **sf3** - vertices 1 edges 1
- **sf10** - vertices 1 edges 1

Dataset is imported as a CYPHERL file of Cypher queries. Feel free to check dataset links for complete Cypher queries. Keep in mind that the dataset is imported differently for each vendor. For example, Memgraph uses Cypher queries to import the dataset, while Neo4j uses `neo4j-admin import` tool.

Index queries for each supported vendor can be downloaded from

"[https://s3.eu-west-](https://s3.eu-west-1.amazonaws.com/deps.memgraph.io/dataset/ldbc/benchmark/vendor_name.cypher)

[1.amazonaws.com/deps.memgraph.io/dataset/ldbc/benchmark/vendor_name.cypher](https://s3.eu-west-1.amazonaws.com/deps.memgraph.io/dataset/ldbc/benchmark/vendor_name.cypher)", just make sure to use the proper vendor name such as

```
memgraph.cypher
```

More details about the dataset in the [Business Intelligence workload class](#)

DISCLAIMER: This is NOT an official implementation of an LDBC Benchmark.

🔗 Query list

The queries are executed for each dataset independently and on each dataset size the identical queries are used. Query parameters differ between different dataset sizes. The complete list of queries can be found in the following files for each workload:

- Pokec
- LDBC Interactive
- LDBC Business Intelligence

🔗  **Platform** → They do note: Support various HW configs

Testing on different hardware platforms and cloudVMs is essential for validating benchmark results. Currently, the tests are run on two different platforms.

🔗 Intel

- Server: HP DL360 G6
- CPU: 2 x Intel Xeon X5650 6C12T @ 2.67GHz
- RAM: 144GB
- OS: Debian 4.19

🔗 AMD

- CPU: AMD Ryzen 7 3800X 8-Core Processor
- RAM: 64GB

🔗 Supported databases

Due to current [database compatibility](#) requirements, the only supported database systems at the moment are:

1. Memgraph v2.7
2. Neo4j Community Edition v5.6.


🔗 Database notes

Need
Summary
Stats on
each to
judge

Running configurations that differ from default configuration:

- Memgraph - `storage_snapshot_on_exit=true` ,
`storage_recover_on_startup=true`
- Neo4j - `dbms.security.auth_enabled=false`

Contributions

As previously stated, Benchgraph will expand, and we will need help adding more datasets, queries, databases, and support for protocols in Benchgraph. Feel free to contribute to any of those, and throw us a start !

History and Future of Benchgraph

History of Benchgraph

Infrastructure around Benchgraph (previously mgBench) was developed to test and maintain Memgraph performance. When critical code is changed, a performance test is run on Memgraph's CI/CD infrastructure to ensure performance is not impacted. Due to the usage of Benchgraph for internal testing, some parts of the code are still tightly connected to Memgraph's CI/CD infrastructure. The remains of that code do not impact benchmark setup or performance in any way.

Future of Benchgraph

We have big plans for Benchgraph infrastructure that refers to the above mentioned [limitations](#).

Also high on the list is expanding the list of vendors and providing support for different protocols and languages. The goal is to use Benchgraph to see how well Memgraph performs on various benchmarks tasks and publicly commit to improving.

Benchgraph is currently a passive benchmark since resource usage and saturation across execution are not tracked. Sanity checks were performed, but these values are needed to get the full picture after each test. Benchgraph also deserves its own repository, and it will be decoupled from Memgraph's testing infrastructure.

🔗 Changelog Benchgraph public benchmark

Latest version: <https://memgraph.com/benchgraph>

🔗 Release v2 (latest) - 2023-25-04

- Benchmark process changes:
 - Executed query count is now identical on both vendors (Memgraph and Neo4j)
- Benchmark presets:
 - single-threaded-runtime = 30 seconds
 - number-of-workers-for-benchmark = 12, 24, 48
 - query-count-lower-bound = 300 queries
 - Mixed and realistic workload queries = 500 queries
- [Full results](#)
- Memgraph got a label index on :User node for Pokec dataset, Neo4j has that index by default.

🔗 Release v1 - 2022-30-11

- Benchmark presets:
 - single-threaded-runtime = 10 seconds
 - number-of-workers-for-benchmark = 12
 - query-count-lower-bound = 30 queries
 - Mixed and realistic workload queries = 100 queries
- Results summary for [Pokec small](#)

- Results summary for [Pokec medium](#)
- Results for [Memgraph cold](#)
- Results for [Memgraph hot]https://github.com/memgraph/benchgraph/blob/main/results/v1/memgraph_hot.json)
- Results for [Neo4j cold](#)
- Results for [Neo4j hot](#)