



LinGBM: A Performance Benchmark for Approaches to Build GraphQL Servers

Sijin Cheng  and Olaf Hartig 

Department of Computer and Information Science (IDA), Linköping University
firstname.lastname@liu.se

Abstract. GraphQL is a popular new approach to build Web APIs that enable clients to retrieve exactly the data they need. Given the growing number of tools and techniques for building GraphQL servers, there is an increasing need for comparing how particular approaches or techniques affect the performance of a GraphQL server. To this end, we present LinGBM, a GraphQL performance benchmark to experimentally study the performance achieved by various approaches for creating a GraphQL server. In this paper, we discuss the design considerations of the benchmark and describe its main components (data schema; query templates; performance metrics). Thereafter, we present experimental results obtained by applying the benchmark in two different use cases, which demonstrate the broad applicability of LinGBM.

Keywords: GraphQL, benchmark, performance, testbed, experiments

1 Introduction

GraphQL is a new approach to build data access APIs for Web and mobile applications [7]. Since its first published specification in 2015, the approach has become highly popular with a flourishing ecosystem of related software tools and programming libraries [15], and many adopters. For instance, an early-2019 study of open source projects identified more than 37,000 code repositories that depend on the GraphQL reference implementation (which is just one of several implementations of the approach) [10]. A similar study found 8,399 unique GraphQL API schemas on Github [17]. Besides open source projects, many companies are adopting GraphQL for their commercial software applications, including household names such as Airbnb, AWS, Expedia, IBM, Paypal, and Twitter [14].

What makes GraphQL interesting from a systems research perspective is that it is based on a declarative query language which enables clients to define precisely the data they want to retrieve. In comparison to REST interfaces, this approach reduces the number of requests that need to be issued and the amount of data transferred from server to client [2,3]. Leveraging this advantage, however, requires GraphQL servers that can process such query requests efficiently.

While there exists a plethora of Web tutorials and blog posts, as well as several books (e.g., [12,8,4,5,11,16]), that all describe approaches to implement a GraphQL server and to avoid typical performance pitfalls, studies that show or

even compare how using particular approaches may affect the performance of the resulting GraphQL server are rare and remain often anecdotal. Yet, understanding the pros and cons of different solutions is crucial for building an *efficient* GraphQL server that provides an optimal performance for a given application.

Achieving such an understanding requires performance tests, for which suitable experimentation frameworks, methods, and tooling are needed. Although there are a few performance-related test suites for specific GraphQL tools (cf. Section 2) and some basic experimental results [13], we observe that there does not exist any methodological approach to thoroughly evaluate and compare the performance of approaches to create a GraphQL server. In this paper we introduce a GraphQL performance benchmark called LinGBM to fill this gap.

Contributions and organization of the paper: Our main contribution in this paper is LinGBM, that is, a benchmark to experimentally study and compare the performance achieved by various approaches to create a GraphQL server. The benchmark consists of¹ i) a data schema for creating benchmark datasets at different scales, ii) 16 query templates that cover different performance-related challenges of GraphQL, iii) performance metrics and execution rules, and iv) the necessary tooling to conduct experiments with the benchmark (e.g., dataset and query workload generators, test drivers). Before describing these elements of the benchmark in detail (Section 4), we discuss the design considerations for the benchmark, including the design methodology and design artifacts (Section 3).

Given the benchmark, we make further contributions by demonstrating several microbenchmarking use cases in which we apply LinGBM (Section 5). In particular, we show that LinGBM can be used i) to evaluate the effectiveness of optimization techniques for GraphQL servers and ii) to study approaches that focus on improving the read scalability of GraphQL servers. In this context, we also present experimental results that highlight the pros and cons of selected techniques, and we outline further application scenarios for the benchmark.

Due to space limitations, this paper assumes familiarity with GraphQL. In an extended version of this paper we provide an overview of GraphQL and of approaches to create GraphQL servers, and discuss LinGBM in more detail [6].

2 Existing Test Suites

While there is no work on GraphQL benchmarks in the research literature, there exist a few performance-related test suites (cf. Table 1). These test suites are GraphQL variations of HTTP load testing tools such as wrk and vegeta.² Each of them consists of a specific dataset (of a comparably small size), a few GraphQL queries, and a test driver that records and visualizes throughput measurements obtained by issuing these queries to a GraphQL server built over the dataset.

¹ All the material related to LinGBM is available online (including, e.g., files with the query templates, the source code of tools, and documentation). In the related parts of this paper we provide links to the relevant Web pages.

² <https://github.com/wg/wrk> and <https://github.com/tsenart/vegeta>

Table 1: Comparison of existing GraphQL test suites.

test suite	number of datasets	number of queries	design method
gbench	https://github.com/graphql-quiver/gbench 1 (100 empty objects and 1 string)*	5 queries	unclear
The Benchmark framework	https://github.com/the-benchmark/graphql-benchmarks 1 (10 tuples)*	1 query	unclear
PostGraphile's GraphQL Bench	https://github.com/benjie/graphql-bench-prisma 1 (15,607 tuples)	9 queries	unclear
Hasura's GraphQL Bench	https://github.com/hasura/graphql-bench 1 (23,288 tuples)	3 queries	unclear
GraphQL server benchmark	https://github.com/tsegismont/graphql-server-benchmark 1 (60 tuples)	4 templates with up to 10 instances each	unclear
LinGBM (<i>our proposal</i>)	https://github.com/LiUGraphQL/LinGBM unlimited (unbounded 16 templates with scale factor)	100-1M+ instances each	choke-point based

*need to be hardcoded in the resolver functions of the tested servers

We argue that these test suites are insufficient for benchmarking the performance achieved by different approaches to build GraphQL servers. By focusing on a single (small) dataset, these test suites cannot be used to study the behavior of GraphQL server implementations at scale. By using only a small number of fixed queries, it is not possible to extensively test or compare the throughput of systems that may apply caching on various levels. Additionally, it is not clear whether the few selected queries test all important aspects of approaches to build GraphQL servers. Our work in this paper addresses the limitations of the existing test suites and, more generally, the lack of a well-designed performance benchmark for evaluating and comparing approaches to build GraphQL servers.

3 Design of the Benchmark

The aim of our benchmark is to provide a framework that can be used to test and to compare the performance that can be achieved by different approaches to build GraphQL servers. To make this aim more concrete we identified two use case scenarios for the benchmark. Given these scenarios, we developed the benchmark by applying the design methodology for benchmark development of the Linked Data Benchmark Council [1]. The main artifacts created by the process of applying this methodology are i) a data schema, ii) a workload of operations to be performed by the system under test, iii) performance metrics, and iv) benchmark execution rules. A crucial aspect of the methodology is to identify key technical challenges, so-called *choke points*, for the types of systems for which the benchmark is designed. These choke points then inform the creation of the aforementioned artifacts. In this section we describe the two use case

scenarios and provide an overview of the choke points defined for our benchmark. The benchmark artifacts shall then be introduced in the next section.

3.1 Use Case Scenarios

Scenario 1 represents use cases in which data from a *legacy database* has to be exposed as a *read-only* GraphQL API with a *user-specified GraphQL schema*. Hence, this scenario focuses primarily on tools and techniques to implement GraphQL servers manually.

Scenario 2 represents use cases in which data from a *legacy database* has to be exposed as a *read-only* GraphQL API provided by an *automatically generated GraphQL server*. Hence, this scenario focuses on tools that auto-generate all artifacts necessary to set up a GraphQL API over a legacy database. Notice that such tools do not support the first scenario out of the box because any GraphQL API created by such a tool is based on a tool-specific generated GraphQL schema (not a user-specified one).

Due to the space limitation, the rest of this paper focuses primarily on Scenario 1.

3.2 Choke Points

As mentioned before, we have applied a choke-point based methodology [1] for designing our benchmark. To this end, we have identified 16 choke points for GraphQL servers. As per our two benchmark scenarios (which capture read-only use cases), these choke points focus only on queries. Table 2 (left-hand side) lists these choke points, which are grouped into the following five classes.³

Choke Points Related to Attribute Retrieval: Queries may request the retrieval of multiple attributes (scalar fields) of the data objects selected by the queries. The technical challenge captured by the corresponding choke point is to fetch these attributes from the underlying data source using a single operation rather than performing a separate fetch operation for each attribute.

Choke Points Related to Relationship Traversal: One of the main innovations of GraphQL in comparison to REST APIs is that it allows users to traverse the relationships between data objects in a single request. Supporting such a traversal in a GraphQL server may pose different challenges, which are captured by the choke points in this class. For instance, choke point CP 2.4 captures the challenge to avoid unnecessary operations in cases in which relationships between requested objects form directed cycles. Queries that traverse along these relationships may come back to an object that has been visited before on the same traversal path. A naive implementation may end up requesting the same data multiple times from the underlying data source. Even a more sophisticated solution that caches and reuses the results of such requests may end up repeating the same operations over the cached data.

³ For a detailed description of all 16 choke points covered by our benchmark we refer to our wiki: <https://github.com/LiUGraphQL/LinGBM/wiki/Choke-Points>

Table 2: Choke points of LinGBM and their coverage by the 16 query templates.

	QT:															
	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
Attribute Retrieval																
CP 1.1 Multi-attribute retrieval	X							X		X						X
Relationship Traversal																
CP 2.1 Traversal of 1:N relationship types	X	X	X	X	X	X	X	X	X		X	X	X	X		
CP 2.2 Traversal of 1:1 relationship types	X	X	X	X	X	X	X	X	X	X						
CP 2.3 Traversal with retrieval of intermediate object data			X	X	X						X	X	X	X		
CP 2.4 Traversal of relationship cycles						X										
CP 2.5 Acyclic relationship traversal that visits objects repeatedly			X		X	X			X	X			X	X		
Ordering and Paging																
CP 3.1 Paging without offset								X	X							
CP 3.2 Paging with offset							X									
CP 3.3 Ordering								X	X							
Searching and Filtering																
CP 4.1 String matching									X				X	X		
CP 4.2 Date matching														X		
CP 4.3 Subquery-based filtering											X		X	X		
CP 4.4 Subquery-based search									X							
CP 4.5 Multiple filter conditions														X		
Aggregation																
CP 5.1 Calculation-based aggregation																X
CP 5.2 Counting																X

Choke Points Related to Ordering and Paging: Since an exhaustive traversal of a sequence of 1:N relationships may easily result in reaching a prohibitively large number of objects, providers of GraphQL APIs aim to protect their servers from queries that cause such resource-intensive traversals. A common solution in this context is to enforce clients to use paging when accessing 1:N relationships, which essentially establishes an upper bound on the maximum possible fan-out at every level of the traversal. A feature related to paging is to allow users to specify a particular order over the objects visited by traversing a 1:N relationship. This class of choke points focuses on implementing these features efficiently.

Choke Points Related to Searching and Filtering: Field arguments in GraphQL queries are powerful not only because they can be used as a flexible approach to expose paging and ordering features. Another use case, which is perhaps even more interesting from a data retrieval point of view, is to expose arbitrarily complex search and filtering functionality. The choke points in this class capture different challenges related to this use case.

Choke Points Related to Aggregation: Another advanced feature that GraphQL APIs may provide is to execute aggregation functions over the queried data. Challenges in this context are to compute aggregations efficiently (CP 5.1)—e.g., by pushing their computation into the underlying data source—and to rec-

ognize that for counting, the corresponding objects/values may not actually have to be retrieved from the underlying data source (CP 5.2).

4 Elements of the Benchmark

4.1 Data Schema

The data schema of the benchmark⁴ consists of i) a database schema for synthetic datasets that can be generated in the form of an SQL database or an RDF graph database, ii) rules for generating such datasets in different sizes, iii) a GraphQL schema for a GraphQL server that may provide access to any version of the benchmark dataset, and iv) a schema mapping that defines how the elements of the GraphQL schema map to the database schema.

Datasets. Instead of creating a new dataset generator from scratch, LinGBM reuses the dataset generator of the Lehigh University Benchmark (LUBM) [9]. LUBM is a popular benchmark in the Semantic Web community for evaluating the performance of storage and reasoning systems for RDF data. The generated datasets capture a fictitious scenario of universities with departments, different types of faculty (lecturers, assistant professors, etc.), students, courses, research publications, and other related types of entities as well as corresponding relationships between them. It is easy to imagine different Web or mobile applications in such a scenario that enable students or researchers to browse and interact with the data, where these applications access the data via a GraphQL API. Hence, these datasets are a suitable starting point for a GraphQL benchmark.

In order “*to make the data[sets] as realistic as possible,*” the dataset generator applies “*restrictions [that] are [...] based on common sense and domain investigation*” [9]. For instance, each university has 15–25 departments, each department has 7–10 full professors, and the undergraduate student/faculty ratio per department is between 8 and 14, whereas the graduate student/faculty ratio is between 3 and 4.⁵ The actual cardinalities are selected from these ranges uniformly at random. Similarly, when generating relationships between generated entities, the entities to be connected are selected uniformly at random from the corresponding pool of possible entities. Depending on the type of relationship, this pool of possible entities is either context specific (e.g., students may take courses only from their department) or global (e.g., grad students may have their undergraduate degree from any university). The advantage of using uniform distributions for the data is that different queries of the same query template have the same predictable performance footprint; that is, they are roughly the same in terms of properties such as intermediate result sizes and overall result sizes.

In addition to being sufficiently realistic and diverse in terms of different types of relationships, another important property for our purposes is that these datasets can be generated at different sizes where the number of universities to be created serves as the *scale factor*. That is, the smallest dataset, at scale

⁴ <https://github.com/LiUGraphQL/LinGBM/wiki/Data-Schema-of-the-Benchmark>

⁵ <http://swat.cse.lehigh.edu/projects/lubm/profile.htm>

factor 1, consist of the data about one university. Yet another important property is that the data generation process is both deterministic and monotonic; hence, all data that is generated at a smaller scale factor is guaranteed to be contained in every dataset generated with the same random seed at a greater scale factor. Due to these properties, we consider the LUBM datasets as a suitable basis for our benchmark. The fact that LUBM has been designed for a different purpose is not an issue in this context because its focus on reasoning systems is reflected mainly in the queries defined for LUBM, not in its datasets.

The only relevant limitation of the LUBM datasets is that they can be created only as RDF data. For LinGBM we wanted to also support SQL databases as underlying data sources for the tested GraphQL servers. Therefore, we have defined a relational database schema⁶ that resembles the concepts and relationships of the LUBM ontology, and we have developed a mapping from the generated RDF graphs to SQL databases that are instances of our database schema.

GraphQL schema. In addition to the benchmark datasets, LinGBM introduces a GraphQL schema for exposing any version of these datasets as a GraphQL API. Essentially, this schema contains an object type for each type of entities in the benchmark dataset (universities, departments, graduate students, etc). The fields of each such object type match both the attributes of the corresponding entity type and its relationships to other entity types. For example, the object type `GraduateStudent` in the LinGBM GraphQL schema has fields such as `emailAddress` and `memberOf` where the former is for the email-address attribute of each graduate student in the generated datasets and the latter is for the relationship that such students have to the department they belong to. Hence, the value type of this `memberOf` field is the object type `Department` which, in turn, contains a field called `graduateStudents`, with a `GraduateStudent` list as value type, to allow for GraphQL queries that traverse the relationship in the reverse direction.

In addition to the object types that we created by this straightforward translation of the database schema into a GraphQL schema, we added a few more fields and types to the GraphQL schema in order to cover all the aforementioned choke points of the benchmark. For example, some fields were extended with arguments to express filter conditions or requirements for sorting and paging. The complete LinGBM GraphQL schema can be found online⁷, and we also provide a definition of the mapping⁸ between this GraphQL schema and the schema of the benchmark datasets. Notice that this GraphQL schema is relevant only for Scenario 1 of the benchmark (cf. Section 3.1). GraphQL schemas as used in Scenario 2 are auto-generated by the corresponding systems under test.

Statistics. The size of each LinGBM dataset depends on the corresponding scale factor. Table 3 presents these statistics for scale factors 1, 10, 100, and 150, using three

Table 3: Dataset sizes at different scale factors.

	<i>sf</i> = 1	<i>sf</i> = 10	<i>sf</i> = 100	<i>sf</i> = 150
file size	12 MB	161 MB	1.66 GB	2.60 GB
overall rows	43,319	542,467	5,707,958	8,490,274
overall objects	17,195	207,426	2,179,766	3,243,523

⁶ <https://github.com/LiUGraphQL/LinGBM/wiki/Datasets>

⁷ <https://github.com/LiUGraphQL/LinGBM/tree/master/artifacts>

⁸ <https://github.com/LiUGraphQL/LinGBM/wiki/Schema-Mapping>

different metrics: i) the file size of the generated SQL import scripts, ii) the sum of the number of rows across all tables of the generated SQL database, and iii) the sum of the overall number of objects for all types of the LinGBM GraphQL schema. As can be observed from these numbers, for each of the three metrics, the dataset size increases linearly with the scale factor.

4.2 Query Templates

As a basis for creating query workloads, we have hand-crafted a mix of 16 templates of GraphQL queries such that, on one hand, these queries cover the choke points identified in the initial design phase of our benchmark (cf. Section 3.2). At the same time, given the university scenario represented by the benchmark datasets, the queries capture data retrieval requests that may be issued by Web or mobile applications built for such a scenario. We emphasize that these queries are completely independent of the queries considered by the aforementioned LUBM benchmark. Although we adopt (and extend) the dataset generator of LUBM, the queries of that benchmark are irrelevant for our purpose because they have been created with a focus on testing RDF stores and reasoners. In contrast, the mix of query templates that we have created for LinGBM focuses on GraphQL servers and their specific choke points. Table 2 illustrates the coverage of these choke points by our LinGBM query templates.

Each such template is a GraphQL query that contains at least one placeholder for specific values that exist in the generated benchmark datasets. To instantiate such a template into an actual query, every placeholder has to be substituted by one of the possible values. For some placeholders, the number of possible values depends on the scale factor (bigger versions of the benchmark datasets may contain more possible values), whereas other placeholders are independent of the scale factor. For query templates with a placeholder of the former type, the number of possible instances of the template increases with the scale factor. Table 4 lists these numbers for each template at different scale factors.

While all 16 templates can be found online⁹, including a detailed description of each of them¹⁰, in the following, we describe one of them as an example.

Figure 1 presents query template QT5, which is a typical example of queries that traverse relationships in cycles and that, thus, may come back to the same objects multiple times. In the particular case of QT5, the traversal starts from a

Table 4: Number of query instances at different scale factors.

	<i>sf</i> = 1	<i>sf</i> = 10	<i>sf</i> = 20
QT1	540	6,843	14,457
QT2	1,000	1,000	1,000
QT3	224	2,827	6,032
QT4	93	1,128	2,399
QT5	15	189	402
QT6	1,000	1,000	1,000
QT7	48,950	493,250	989,250
QT8	15,000	15,000	15,000
QT9	2,000	2,000	2,000
QT10	27,077	344,750	728,208
QT11	1,000	1,000	1,000
QT12	14,685	1,864,485	7,953,570
QT13	899,701	113,921,020	480,241,305
QT14	6,297,907	797,447,140	3,361,689,135
QT15	1,000	1,000	1,000
QT16	1,000	1,000	1,000

⁹ <https://github.com/LiUGraphQL/LinGBM/blob/master/artifacts/queryTemplates>

¹⁰ <https://github.com/LiUGraphQL/LinGBM/wiki/Query-Templates>

given department, retrieves the university of this department, then proceeds to retrieve all graduate students with an undergraduate degree from this university,

```

query qt5($departmentID:ID) {
  department(nr:$departmentID) {
    id
    subOrganizationOf {
      id
      undergraduateDegreeObtainedBystudent {
        id
        emailAddress
        memberOf {
          id
          subOrganizationOf {
            id
            undergraduateDegreeObtainedBystudent {
              id
              emailAddress
              memberOf { id }
            }
          }
        }
      }
    }
  }
}

```

Fig. 1: Query template QT5.

and then to the departments that these students are members of. This cycle is repeated two times. Hence, QT5 covers choke point CP 2.4. Additionally, by requesting the students' email addresses along the way, the template also covers choke point CP 2.3. Furthermore, the template covers CP 2.1 (because of the traversal from a university to graduate students) and CP 2.2 (because of the traversal from departments to their respective university, as well as from each graduate student to their department). The placeholder of this query template is `$departmentID`, which is used to select a department based on its number as a starting point for the traversal. Hence, for any benchmark dataset, all department numbers in this dataset can be used to instantiate QT5 in order to obtain queries that can be used for the dataset, as well as for all datasets generated with scale factors greater than the given dataset.

4.3 Performance Metrics

The performance metrics considered by LinGBM are defined based on the following two notions: i) *Query execution time* (*QET*) is the amount of time that passes from the begin of sending a given query request to the GraphQL server under test until the complete query result has been received in return. ii) *Throughput* is the number of queries that are processed completely by a GraphQL-based client-server system within a specified time interval, where a query is considered to be processed completely after its complete result has been received by the client that requested the execution of the query.

Then, for **single queries**, we define the metric $aQET_q$ as the average of the QETs measured when executing an individual query multiple times with the GraphQL server under test. When reporting this metric, the corresponding standard deviation has to be reported as well.

For whole **query templates**, we define the following two metrics: i) QET_t is the distribution of the individual QETs measured for multiple queries of the same template and ii) aTP_t is the average of the throughput measured when running the same query workload multiple times, where the queries in the workload are all from the same template.

For **mixed workloads** with queries from multiple templates, i) aTP_w is the average of the throughput measured when running the same mixed query workload multiple times and ii) aTP_m is the average of the throughput measured for multiple mixed workloads, where each such workload is run once.

4.4 Tools

To enable users to perform experiments with the benchmark we have developed a number of tools, including a dataset generator, a query generator, and test drivers for both throughput and QET experiments. For more information about them refer to the github repository of the benchmark.¹¹

5 Application of the Benchmark

In this section we demonstrate the applicability of LinGBM for two different microbenchmarking use cases and present corresponding experimental results.

5.1 General Experiments Setup

All experiments described in the following have been performed on a server machine with two 8-core Intel Xeon E5-2667 v3@3.20GHz CPUs and 256 GB of RAM. The machine runs a 64-bit Debian GNU/Linux 10 server operation system. On this machine, we use Docker (v9.03.6) to run all components of the experiment setups in a separate, virtual environment (e.g., the GraphQL server under test, the database server used as data source, and the LinGBM test driver).

All GraphQL servers that we implemented manually for the experiments are node.js (v10.21.0) applications that use the Apollo Server package (v2.17.0) and, for database access, the knex.js package (v0.20.15). As database server we use PostgreSQL (v12.1, default configuration options), given as a public Docker image, for which we limit the available resources to two vCPUs and 1 GB RAM. To obtain the relevant measurements we used the LinGBM test drivers.

Based on preliminary tests, we selected the following default parameters for the experiments. Unless specified otherwise, we use scale factor 100 and, to connect to the database server, the manually-implemented GraphQL servers use connection pooling with up to 10 parallel connections. Most experiments focus on average throughput per template (aTPt, cf. Section 4.3) with one client, for which we always do six runs of 60 secs where the first run is regarded as warm-up and the number of successfully completed queries per each of the other five runs are averaged. To have a sufficiently high number of distinct queries for these throughput runs, we generated 5000 queries for every template for which this is possible at scale factor 10 (which is the smallest scale factor used in our experiments), and for the other templates we used the maximum possible (cf. Table 4).

5.2 Evaluation of Optimization Techniques

The aim of our first use case is to evaluate the effectiveness of two prominent optimization techniques used in GraphQL servers, and to show which choke points each of these techniques can address. The techniques are called server-side

¹¹ <https://github.com/LiUGraphQL/LinGBM/tree/master/tools>

caching and batching.¹² The idea of *server-side caching* is to cache the response to every request that the resolver functions in the GraphQL server make to the underlying data source, and if the exact same request is made again within the scope of executing a given GraphQL query, then use the response from the cache instead of accessing the data source again. The idea of *batching* is to combine multiple similar requests to the underlying data source into a single request. Typically, this can be done for the requests issued by the same resolver function based on different inputs. A popular tool to implement this form of server-side request batching is called DataLoader¹³ which also supports server-side caching.

For this evaluation we have developed a GraphQL server for Scenario 1 of the benchmark by using a straightforward implementation in which resolver functions for the elements of the GraphQL schema issue SQL queries to fetch the relevant data from the underlying database. This GraphQL server implementation represents the baseline for the evaluation and we call it the *naive* server. Thereafter, we have extended this server in three different ways to obtain three additional test servers: As a first variation, we have integrated server-side caching using memoization. For the second variation, we have replaced the naive resolvers by resolvers that use DataLoader to implement both server-side caching and batching. The third variation is a version of the second with caching disabled in DataLoader (i.e., it uses only batching). The source code for these four test servers (the naive one plus the three extended variations) is available online.¹⁴

Initial macro-level comparison. We begin by comparing the test servers based on the aTPm metric using six different mixed workloads. Each of them is a randomly sorted sequence of 100 queries from each template (i.e., 1600 queries per workload in total). We have measured aTPm with one client and a runtime of 600 seconds per workload.¹⁵ For each tested server, the first workload was used as a warm-up and the throughputs achieved for the other five workloads were averaged to calculate aTPm. In this experiment, naive server achieved an average throughput of 200 queries; for the server with caching, it is 312; with batching, 729; and with both batching and caching together, 735. These numbers show that i) both optimization techniques improve upon the baseline of the naive server, ii) batching is significantly more effective than caching, and iii) adding caching to batching does not lead to a significant improvement over batching alone. While this experiment, with its diverse mix of queries, gives us a general idea of the effectiveness of the two optimization techniques, it does not allow us to derive more detailed insights about them. To gain such insights we can leverage the template-specific and query-specific metrics of LinGBM as follows.

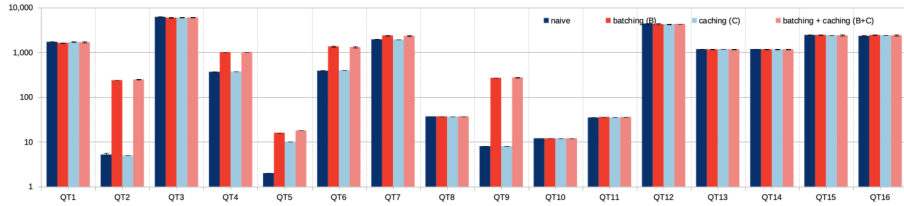
Experiments. As a first microbenchmarking experiment, we have measured the aTPt that the four test servers achieve for each of the 16 query templates at scale factor 100 with one client. Figure 2a illustrates these measurements (er-

¹² <https://graphql.org/learn/best-practices/#server-side-batching-caching>

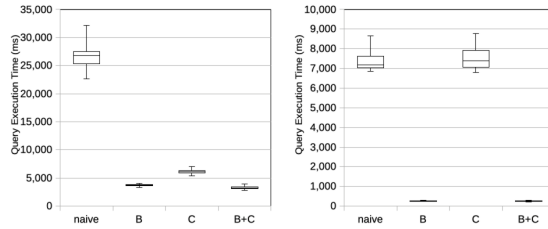
¹³ <https://github.com/graphql/dataloader>

¹⁴ <https://github.com/LiUGraphQL/LinGBM-OptimizationTechniquesExperiments>

¹⁵ We use a longer duration for these runs (600s rather than 60s) to ensure that the tested servers have to process a greater selection of queries from each template.



(a) aTPt (one client) for each query template as achieved with different optimizations



(b) QETt (in ms) for QT5 (c) QETt (in ms) for QT9

Fig. 2: Comparison of GraphQL server implementations with different optimization techniques, in terms of average throughput with one client (a) and execution times of 100 queries of template QT5 (b) and QT9 (c), at scale factor 100.

ror bars represent one standard deviation). Thereafter, we have measured the corresponding QETt required by the test servers for a single run with 100 randomly selected queries per template. The box plots in Figures 2b–2c illustrate these measurements for templates QT5 and QT9, respectively.¹⁶ As a last experiment, we have increased the scale factor from 100 to 125, and then to 150, and measured aQETq for five randomly selected queries per template, as illustrated in Figure 3 for five queries of QT5.

General observations. A first, expected observation is that smaller query execution times result in a greater throughput, as can be seen in Figure 2 by comparing the aTPt and the corresponding QETt that each test server achieves for QT5 and QT9. We also observe that, for the queries of some query templates, the execution times increase significantly at increasing scale factors (e.g., QT5, cf. Figure 3), whereas for other templates the changes are less substantial. We explain these differences by differences in how the respective query result sizes increase at greater scale factors (i.e., for some templates the result sizes increase more than for others).

Server-side caching. In our experiment, server-side caching has a major impact only for QT5. The reason why executions of QT5 queries can leverage caching is because the template captures choke point CP 2.4 (traversal of relationships that form cycles). More precisely, these queries retrieve data about particular

¹⁶ The complete set of QETt charts for all templates can be found in a companion document in the aforementioned github repository with the four test servers.

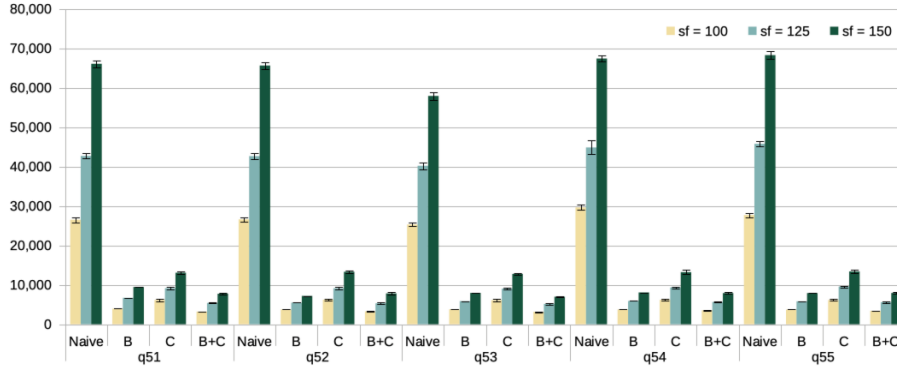


Fig. 3: aQETq (in ms) for individual queries of QT5 at increasing scale factors.

graduate students once, and then come back to these graduate students later in a subquery; additionally, for these graduate students, the queries retrieve data about the students’ departments and multiple students belong to the same department. Server-side caching enables the GraphQL server to avoid fetching the same data for these students and departments multiple times from the database.

Server-side batching. The idea to combine multiple requests to the underlying database helps to increase the throughput for queries of QT2, QT4–QT7, and QT9 (cf. Figure 2a). The choke point that is common to these query templates is CP 2.1 which captures traversals of 1:N relationship types. The execution of such queries involves at least two resolvers where the first one returns an array of objects and then, for each of these objects, the second resolver is invoked once. If this second resolver performs an SQL request in the context of the given input object, these requests for the different input objects from the array can be batched, and that is exactly the case for the aforementioned templates.

While the same is true also for templates QT12–QT14, batching has no effect for queries of these templates. The reason is that these templates contain filter conditions regarding the objects in the corresponding array, and only few objects satisfy this condition. As a result, the number of SQL requests that are batched in these cases is just too small to make a difference.

A question that remains is why QT1 is not affected by batching although it covers CP 2.1 as well. The reason is that, in this case, none of the resolvers that are invoked multiple times issues any SQL requests (the data they use has already been fetched by parent resolvers). Hence, we conclude that batching addresses CP 2.1 for queries in which any subquery that follows a traversal of a 1:N relationship requires further requests to the underlying data source.

5.3 Evaluation of Connection Pooling

With the second use case we aim to demonstrate that LinGBM can be employed to evaluate the effectiveness of approaches to achieve read scalability of a

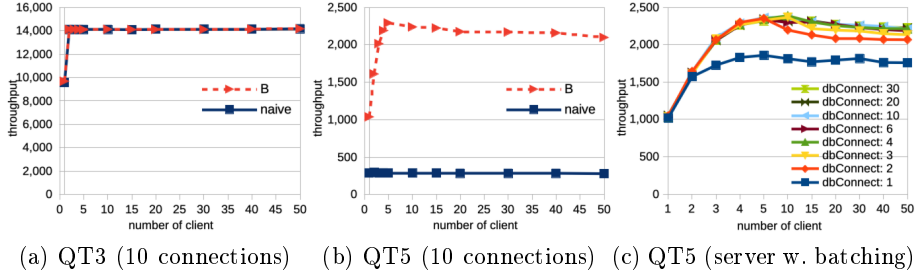


Fig. 4: Average throughput (aTPt, $sf = 100$) for multiple concurrent clients, where the test servers in (a)–(b) use a max of 10 connections to the underlying DB.

GraphQL server. While there is a wide range of options to this end, we consider a simple option for demonstrating this use case, namely, the option to vary the number of connections between the GraphQL server to the database server.

Experiments. As a first experiment, to understand how the number of clients affects the performance of our manually-implemented GraphQL servers, we have repeated the earlier throughput experiment with an increasing number of clients that issue sequences of GraphQL queries concurrently (first two clients, then three, four, five, ten, 15, 20, 30, 40, and 50). Figures 4a–4b illustrate these measurements for both the naive server and the server with batching, for the queries of QT3 and QT5, respectively. Thereafter, for the server with batching, we have repeated this experiment with different values for the maximum number of database connections. Figure 4c illustrates these measurements for the queries of QT5 (note that the x-axis in this chart is stretched to better see the measurements for smaller numbers of clients). For these experiments we used a smaller dataset ($sf = 10$) because, in some cases for the bigger dataset, the test servers became overloaded when serving multiple clients; in particular, this was the case for queries that have much bigger results at greater scale factors (e.g., QT5).

Observations. For QT3, and both server variants, we observe that the throughput increases when going from one to two clients, but then it does not increase further when adding more clients. We explain this behavior as follows: QT3 queries traverse along three N:1 relationship types and, thus, have results that consist of a single leaf node, and batching cannot be leveraged for these queries. During the execution of each such query, the GraphQL server issues four SQL requests to the database server, one after the other. Hence, with the default connection pool size of 10, the queries of two clients can be served concurrently without any interference. However, when aiming to serve three clients or more, the executions of the concurrent queries are competing for the available database connections and, thus, the throughput stagnates.

For QT5, without batching, the naive test server issues several hundred SQL requests per GraphQL query. In this case, the limited number of available database connections becomes a bottleneck already for one client. In contrast,

when using batching, the test server needs only three SQL requests for each QT5 query and, thus, the throughput starts to increase when serving more than one client concurrently. In comparison to QT3, however, for QT5 queries, fetching data from the database is not the only major task of the test server but, instead, the fetched data also needs to be combined into larger result trees. As a consequence, even if concurrent query executions compete for the available database connections, the overall throughput increases up to five clients. However, when increasing the number of concurrent clients beyond five, the throughput starts to drop slightly. At this point, since each of the batched SQL requests fetches more data, constantly switching between requests for different concurrent queries means that the waiting times of each concurrent query execution are affected more and more as the number of concurrent query executions increases.

If we now consider the option to vary the connection pool size (cf. Figure 4c), we make two observations in our setting for QT5: First, a connection pool size that is smaller than the default value of 10 causes the throughput to drop already for smaller numbers of clients, which is not unexpected of course. Second, however, increasing the connection pool size does not help to improve the throughput anymore. At this point, the database server becomes the bottleneck.

6 Concluding Remarks

This paper introduces LinGBM, a benchmark to evaluate the performance that can be achieved by various approaches to build a GraphQL server over an existing database. LinGBM captures key technical challenges (“choke points”) to be addressed when building such server. After introducing LinGBM, we have demonstrated its applicability for two different microbenchmarking use cases.

We emphasize, however, that these are not the only types of use cases for which LinGBM can be employed. For instance, our experiments may be extended to setups in which multiple machines are used (e.g., to study the effect of remote database servers or to analyze different load-balancing approaches for GraphQL servers). Moreover, given that the benchmark datasets can also be generated in the form of RDF graphs, approaches to provide GraphQL-based access to such RDF data—and to graph databases in general—can be tested with the benchmark. Further use cases may focus on stress testing of systems by using mixed workloads from multiple selected templates (e.g., all templates that cover a particular choke point). In fact, the definition of mixed workloads may go beyond considering only an equal and uniform distribution of queries from different templates (as done in Section 5.2). LinGBM provides everything needed to design and run experiments with a specific mix of queries that is typical in a particular application scenario (where some types of queries are more frequent than others).

One of our future work tasks is to define and evaluate such workloads for selected application scenarios. Another task will be to extend the benchmark with update operations and possible read-write workloads.

Acknowledgements

This work was funded by the Swedish Research Council (Vetenskapsrådet, project reg. no. 2019-05655), by CUGS (the National Graduate School in Computer Science, Sweden), and by the CENIIT program at Linköping University (project no. 17.05). We also thank Lukas Lindqvist, David Ångström, and Markus Larsson for contributing to the development of LinGBM-related software.

References

1. Angles, R., Boncz, P.A., Larriba-Pey, J., Fundulaki, I., Neumann, T., Erling, O., Neubauer, P., Martínez-Bazan, N., Kotsev, V., Toma, I.: The Linked Data Benchmark Council: A Graph and RDF Industry Benchmarking Effort. *SIGMOD Rec.* **43**(1), 27–31 (2014)
2. Brito, G., Mombach, T., Valente, M.T.: Migrating to GraphQL: A Practical Assessment. In: Proc. of the 26th Int. Conf. on Software Analysis, Evolution and Reengineering (SANER) (2019)
3. Brito, G., Valente, M.T.: REST vs GraphQL: A Controlled Experiment. In: Proc. of the 2020 IEEE Int. Conf. on Software Architecture (ICSA) (2020)
4. Buna, S.: Learning GraphQL and Relay. Packt Publishing (2016)
5. Buna, S.: GraphQL in Action. Manning Publications (2020)
6. Cheng, S., Hartig, O.: LinGBM: A Performance Benchmark for Approaches to Build GraphQL Servers (Extended Version). CoRR [abs/2208.04784](https://arxiv.org/abs/2208.04784) (2022), <https://arxiv.org/abs/2208.04784>
7. Facebook, Inc.: GraphQL. Online at <http://spec.graphql.org/June2018> (2018)
8. Grebe, S.: Hands-on Full-Stack Web Development with GraphQL and React. Packt Publishing (2019)
9. Guo, Y., Pan, Z., Heflin, J.: LUBM: A Benchmark for OWL Knowledge Base Systems. *J. Web Semant.* **3**(2-3), 158–182 (2005)
10. Kim, Y.W., Consens, M.P., Hartig, O.: An Empirical Analysis of GraphQL API Schemas in Open Code Repositories and Package Registries. In: Proc. of the 13th Alberto Mendelzon Int. Workshop on Foundations of Data Management (2019)
11. Kimokoti, B.: Beginning GraphQL. Packt Publishing (2018)
12. Porcello, E., Banks, A.: Learning GraphQL: Declarative Data Fetching for Modern Web Apps. O'Reilly Media, Inc. (2018)
13. Roksel, P., Konieczny, M., Zielinski, S.: Evaluating Execution Strategies of GraphQL Queries. In: Proc. of the 43rd Int. Conf. on Telecommunications and Signal Processing (TSP) (2020)
14. The GraphQL Foundation: 2019 Annual Report. Online at <https://graphql.org/foundation/annual-reports/2019/> (2019)
15. The GraphQL Foundation: GraphQL Landscape. Online at <https://landscape.graphql.org> (2021)
16. Williams, B., Wilson, B.: Craft GraphQL APIs in Elixir with Absinthe. The Pragmatic Programmers, LLC (2018)
17. Wittern, E., Cha, A., Davis, J.C., Baudart, G., Mandel, L.: An Empirical Study of GraphQL Schemas. In: 17th Int. Conf. on Service-Oriented Computing (2019)