2021 36th IEEE/ACM International Conference on Automated Software Engineering (ASE) | 978-1-6654-0337-5/21/$31.00 ©2021 IEEE | DOI: 10.1109/ASE51524.2021.9678513

Learning GraphQL Query Cost

Georgios Mavroudeas,* Guillaume Baudart,† Alan Cha,‡ Martin Hirzel,‡ Jim A. Laredo,‡
Malik Magdon-Ismail,* Louis Mandel,‡ and Erik Wittern,‡

*Rensselaer Polytechnic Institute
†Inria Paris, École normale supérieure – PSL university
‡IBM Research

***Abstract*—GraphQL is a query language for APIs and a runtime for executing those queries, fetching the requested data from existing microservices, REST APIs, databases, or other sources. Its expressiveness and its flexibility have made it an attractive candidate for API providers in many industries, especially through the web. A major drawback to blindly servicing a client's query in GraphQL is that the cost of a query can be unexpectedly large, creating computation and resource overload for the provider, and API rate-limit overages and infrastructure overload for the client. To mitigate these drawbacks, it is necessary to efficiently estimate the cost of a query *before executing it*. Estimating query cost is challenging, because GraphQL queries have a nested structure, GraphQL APIs follow different design conventions, and the underlying data sources are hidden. Estimates based on worst-case static query analysis have had limited success because they tend to grossly overestimate cost. We propose a machine-learning approach to efficiently and accurately estimate the query cost. We also demonstrate the power of this approach by testing it on query-response data from publicly available commercial APIs. Our framework is efficient and predicts query costs with high accuracy, consistently outperforming the static analysis by a large margin.**

I. Introduction

GraphQL is an open-source technology for building APIs to support client-server communication [17]. GraphQL has two interconnected components: a *query language* that clients use to specify the data they want to retrieve or mutate, and a *server-side runtime* to validate and execute these queries.

A central architectural design choice in GraphQL is to shift control over what data a request can receive or mutate from API providers to clients. In competing technologies, like the REpresentational State Transfer (REST) architecture, providers define accessible resources and their API endpoints. In GraphQL, clients define queries that can retrieve or mutate multiple related resources in a single request (thus avoiding unwanted round-trips), and select only data they intend to use (thus avoiding over-fetching) [5, 6]. As a result, GraphQL is very suitable for creating diverse client experiences and many organizations, such as Shopify, GitHub, Yelp, Starbucks, NBC, among others, have elected to use GraphQL to build mobile applications and engage with their ecosystem partners [25].

Web API management is a challenging software engineering problem, for which GraphQL provides advantages but also introduces new challenges. A significant downside for providers when shifting control to clients is the risk of overly complex queries, which are expensive and lead to overloaded servers and/or databases. Even small GraphQL queries can yield excessively large responses [8, 11]. Empirical work shows that on many public GraphQL APIs, a linear increase in query size can cause an exponential increase in result size due to the nested nature of queries [27].

Unlike in REST APIs, where providers can avoid excessive use by limiting the number of allowed requests per time interval, in GraphQL, limiting the number of requests is not enough since a single query can break the system. As such, some GraphQL server implementations track query *costs* dynamically during execution [21]. Once a critical threshold is met, the server aborts execution and returns a partial result or an error. Unfortunately, this approach can lock up resources while producing unusable results. Hartig et al. propose to analyze the cost of queries before executing them [11]. Their analysis relies on probing the backend server for data-size information, for example, determining how many users are in the database if a query requests a list of users. However, this requires the server to offer probing facilities, which could themselves strain resources. In contrast, Cha et al. propose a static query cost analysis that does not depend on dynamic information from the server, but only provides upper bounds on cost [8]. This approach has been incorporated into IBM API Connect [14], a commercial API management product.

Unfortunately, these upper bounds are often loose and this gap between estimated and actual cost makes the upper bound excessively conservative as a query filter, resulting in low amortized efficiency. More accurate cost estimates could allow providers to loosen their cost thresholds and help them better provision server resources. In addition, clients can better understand the costs of their queries and how often they can execute them for given rate limits.

Therefore, we propose a machine-learning (ML) solution that predicts query costs based on experience generated over multiple user-server communication sessions. Our solution extracts features from query code by combining approaches from natural-language processing, graph neural networks, as well as symbolic features including ones from static compiler analysis (such as the cost estimate in [8]). It then builds separate regressors for each set of features and combines the component models into a stacking ensemble.

Compared to the static approaches, our solution can underestimate cost of a query but provides estimates that are closer to the actual value.

978-1-6654-0337-5/21/$31.00 ©2021 IEEE
DOI 10.1109/ASE51524.2021.00120

```
query {
  licenses { name }
  repository(owner: "graphql", name: "graphiql") {
    issues(first: 2) { nodes { id } }
    languages(first: 100) { nodes {name} } } }
```

Fig. 1. Query for the GitHub GraphQL API.

```
{ "licenses": [
    {"name": "GNU Affero General Public License v3.0"},
    {"name": "Apache License 2.0"}, ... ],
  "repository": {
    "issues": {
      "nodes": [ {"id": "...NTQ="}, {"id": "...ODg="} ] },
    "languages": {
      "nodes": [ {"name": "HTML"}, {"name": "JavaScript"},
                 {"name": "Shell"}, ... ] } } }
```

Fig. 2. Response corresponding to the query of Figure 1.

This paper makes the following contributions:

- A set of feature extractors for GraphQL query code.
- A general ML workflow to estimate query cost that can be applied to any given GraphQL API.
- A search space of ML model architectures for GraphQL query cost prediction, comprising of choices for ensembling, preprocessing, and regression operators.
- An empirical study of our approach on two commercial APIs, comparing it to previous work and evaluating the practical applicability.

Our approach can help API providers better evaluate the risk of client queries, and it can help clients better understand the cost of their queries to make the best use of their budget.

II. Background

GraphQL queries are executed via a set of data retrieval functions called *resolvers*, i.e., functions that retrieve data for each field in an object type. A resolver can obtain the data from any source, be it from a database, another API, or even from a file. GraphQL queries are a set of nested fields with optional parameters. Fulfilling a query is a matter of calling the resolvers (with their respective parameters) of each field in the query and composing the returned values into a response, resulting in a JSON object containing the same fields as the query. Therefore, the structures of GraphQL queries and responses correspond to each other.

For instance, the query in Figure 1 retrieves the list of open-source licenses available on GitHub and information about the `"graphiql"` repository from the `"graphql"` organization, specifically the IDs of the first 2 issues and the names of the first 100 programming languages used in the repository. Figure 2 shows the response returned by the GitHub GraphQL API. For each field in the query with an object type (e.g., `repository`), the corresponding field in the response contains an object with the fields requested by the sub-query (e.g., `"issues"`). For each field in the query with a list type (e.g., `licenses`), the corresponding field in the response contains a list where each element is an object with the fields requested by the sub-query (e.g., `"name"`).

To reflect the cost of the response, Cha et al. introduced the *type complexity* [8]: the sums of the fields present in either the query or the response, weighted by a configuration associated to the type of each field. For instance, with a configuration where the weight of a scalar type is 0 and the weight of all other types is 1, the type complexity of the response in Figure 2 with 13 `"licenses"` and 5 `"languages"` is 23 (= 13 licenses + 1 repository + 1 issue connection + 2 issues + 1 language connection + 5 languages).

Nested lists can yield exponentially large responses [11]. In our example, the length of the lists `issues` and `languages` are bounded by the argument `first`, as dictated by the connection model [24]. Cha et al. [8] use this information to statically compute an upper bound on the response size from the query. While this upper bound is as tight as possible, it can grossly differ from the actual cost. For example, the static analysis assumes that the query of Figure 1 returns at worst a list of 100 programming languages, but the GraphiQL repository uses only 5 programming languages.

III. Methodology

The goal of this work is to automatically learn more accurate query cost estimates from data. First, we propose a set of specialized features that can be applied to any GraphQL API. These features turn GraphQL queries into suitable input for classic machine learning techniques. Second, we propose a hierarchical model to learn a cost estimate given a GraphQL query. Separate regressors for each features are combined into a stacking ensemble to obtain the final estimate.

A. Feature Extraction

We design three distinct feature extraction methods.

Field Features: A GraphQL API defines a finite number of resolvers. We can thus represent all possible response fields by a vector of fixed size where each index represents a field. We create a feature vector for each GraphQL query, enumerating the total number of times a specific field appears.

Graph Embeddings: The field features only capture information about the cardinality of fields. To capture information about the syntactic structure of the query, we use a second set of features based on graph embeddings. The idea is that a graph neural network can map the abstract syntax tree of a GraphQL query into a low-dimensional embedding space, from which we can then extract the numerical features. To do that, we employed the graph2vec [20] technique, one of the most popular approaches in this area.

Summary Features: The last set of features is a six-dimensional encoding of the queries using symbolic code analysis techniques. They include 1) the *static analysis upper bound* of Cha et al. [8]. They also include features that summarize the tree structure of the queries. These are 2) *query size*, the number of nodes in the query tree, 3) *width*, the maximum number of children a tree node has, and 4) *nesting*, the maximum depth of the tree. Finally, we extract two features related to lists: 5) *lists*, the number of fields in a query requesting a list, and 6) the *sum of list limits* (e.g, `first`). The features vector of Figure 1 is $[118, 17, 2, 3, 3, 115]$ (the list `licenses` has default length 13).

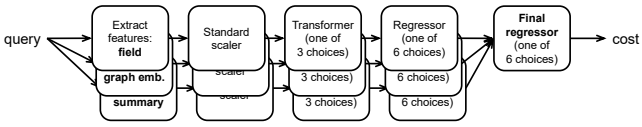


Fig. 3. Stacking ensemble overview.

B. Learning

There are many well-known *operators* that implement regression algorithms (e.g., linear regression, gradient boosting regressors) and also feature preprocessing (e.g., polynomial features transformer). A library like scikit-learn [7] implements many of these operators, but picking the right operators and configuring their hyperparameters is a tedious task and depends on the dataset. We thus use Lale [4], an automated machine learning (auto-ML) tool, to select the best operators and tune the hyperparameters given a query/response dataset.

Definition of three models: For each set of features (field, graph embeddings, and summary features), we train a model independently, but all three of these models have the same architecture. We define a pipeline where 1) the first step uses a standard scaler to give all features a mean of 0 and a standard deviation of 1; 2) the second step does other feature transformations (either no transformation, a polynomial expansion of the features, or a Nystroem transformer); and 3) the last step does the prediction using one of the following six predictors: linear regression, decision tree, ridge regression, random forest, k-nearest neighbors where the number of neighbors is fixed to 3, and gradient boosting regressors.

Model selection: This pipeline defines a space of 18 possible combinations for each of the three models. Furthermore, each of the operators of the pipeline also has a set of hyperparameters to configure. We have fixed some of the hyperparameters, such as $k = 3$ for the k-nearest neighbors, but we left 43 hyperparameters free. The auto-ML tool then chooses the best solution among the possible combinations of algorithms and hyperparameters configurations. To select the best model, we use n-fold cross validation and the Bayesian optimizer from Hyperopt [16].

Combination of models: We train the three models independently and define a new hierarchical model using the outputs of the three models as input for a final model, as shown in Figure 3. This final model provides the estimation in the prediction phase. In general, using a *stacked* ensemble in an ML framework [28], learning each predictor separately and using the predictions as features for the final predictor, can improve accuracy. After experimentation, we found that in our case, this method performs better than concatenating the features into a wide vector and using a single regressor.

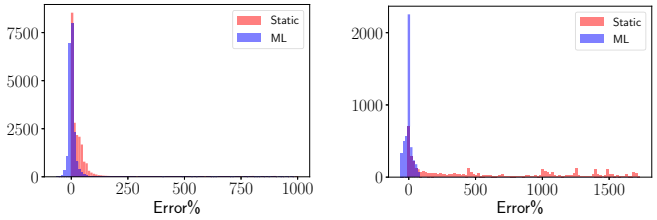
IV. Results

The evaluation addresses the following research questions:

RQ1: Does our approach return accurate estimates?
RQ2: Are all the features useful for the estimation?
RQ3: What are the practical benefits of the new estimation?

TABLE I
Data statistics for the GitHub and Yelp datasets.

	GitHub				Yelp			
	mean	std	min	max	mean	std	min	max
Query Size	109	43	7	1,425	66	30	5	229
Width	23	8	2	53	11	3	1	21
Nesting	3	1.4	1	9	3	0.5	1	3
Lists	47	23	0	503	74	53	0	370
Response	79	67	0	2,548	1,301	2,111	0	7,363



(a) GitHub error distributions. (b) Yelp error distributions.

Fig. 4. Error percentage distribution for the ML and static predictions. For visual purposes we have removed outliers with error% reaching up to $120,000\%$ in static analysis.

A. Experimental Setup

Data: Following the methodology in [8], we collected over 100,000 responses from the GitHub GraphQL API and 30,000 from the Yelp GraphQL API. The discrepancy between the sizes of the two datasets is due to the limit on queries imposed by the two APIs. The queries are synthetically generated but the responses come from industrial APIs. The dataset is available at https://github.com/Alan-Cha/graphql-complexity-paper-artifact. Table I presents the dataset characteristics, where *query size*, *width*, *nesting*, and *lists* are the corresponding summary features from Section III-A, and *response* is the actual cost of the query result.

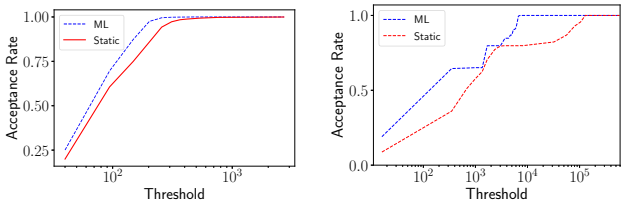
Training: The final model is selected using 5-fold cross validation [15]. We let our optimizer run for sixty hours for each trained pipeline for both the Yelp and GitHub datasets, exploring a total of 1,500 combinations of models and hyperparameters, whichever of the two finishes first. Once operators and hyperparameters are chosen, training a given model is relatively fast. In our experiments, the preferred estimator chosen by the Hyperopt optimizer in most of the training pipelines was the gradient boosting regressor for both datasets.

B. RQ1: Accuracy

We compare our approach to the static analysis proposed in [8], which was shown to outperform the three most popular libraries for computing GraphQL query cost. To quantify the precision of the ML approach compared to the static analysis, Figure 4 presents the error distribution percentages of the ML and static analyses. Given a query with response cost c and a prediction $\hat{c}$, we define the error percentage as $Error\% = (\hat{c} - c)/c$. The accuracy gain of the ML approach compared to the static analysis is striking both in terms of average value and standard deviation (see also the last two lines of Table II).

TABLE II
ACCURACY COMPARISON FOR EACH FEATURE
(MAE $= 1/n \sum_{i=1}^{n} |c_i - \hat{c}_i|$).

	GITHUB		YELP	
	MAE	std	MAE	std
Summary features	8.7	36.4	102.4	280.8
Field features	14.9	40.2	320.8	715.6
Embedding features	31.58	45.7	880.9	813.4
Final combination	8.2	35.5	60.7	180.4
Static analysis	31.5	263.8	14,180.5	30,827.9



(a) GitHub (b) Yelp

Fig. 5. Comparison of the acceptance rate against increasing threshold for the static analysis (red) and the ML approach (blue). The acceptance rate is computed as the number of queries whose sum of estimated complexity is below the threshold level divided by the number of sampled queries.

C. RQ2: Features Selection

As described in Section III, the ML estimates are based on three groups of features, namely summary features (including the result of the static analysis), field features, and graph embedding features. But are all these features necessary? To answer this question, we looked at estimates obtained using each group of features separately. Table II summarizes the results. We observe that for both datasets, none of the feature alone is competitive with the stacked ensemble that combines the results of cost estimation models trained from all three groups of features separately.

The performance of each group of features depends on the dataset. For instance, while the *summary features* give reasonable estimates for both datasets, the *field features* are much more useful for GitHub than for Yelp. This could be related to the underlying structure of the two datasets as well as to the data generation process. Table II also shows that the automatic feature extraction of the neural networks used to build the *graph embedding* features fails to produce accurate estimates for either dataset, underscoring the importance of the more descriptive features

D. RQ3: Practicality

Now that we have access to accurate complexity estimates, the main question is: *how useful are these estimates in practice?* API managers offer, through a client-selected plan, a rate limit, allowing a certain number of points per time window. Points could be attributed to individual REST calls or to the query cost in the case of GraphQL [10, 14, 23]. To mimic this behavior, we built a simulator that acts as an API manager whose goal is to filter queries based on the client plan. We select a threshold of points to represent the rate limit on a given time window.

First, the client sets a threshold, that is, the maximal aggregate cost that the client is willing to pay for a query. Then the simulator acts as a gateway between the API and the client, rejecting queries for which the estimated cost is above the threshold. To evaluate the benefit of our approach, we compare the acceptance rate of a simulator relying on the static analysis against the acceptance rate of a simulator relying on our ML approach. Figure 5 shows the evolution of the acceptance rate for increasing values of the simulation threshold. We used a different range of thresholds for the experiments in Yelp and GitHub respectively due to their specific characteristics (in general Yelp contains queries with larger costs). The results are averaged over 1,000 simulations, and for each simulation we randomly select 1,000 queries. Overall, as expected, the ML cost estimation policy is able to accept a bigger proportion of queries for both APIs. The staircase shape of the Yelp results can be explained by the peculiar cluster-like distribution of query complexity in the dataset. For the same reason, the static analysis plateaus at 80% until the threshold is considerably larger due to the substantial overestimation within this range. When the threshold reaches a high enough value, the static analysis reaches 100% acceptance rate too.

A more thorough evaluation and detailed analysis is available in the extended version of this paper [19].

V. RELATED WORK AND CONCLUSION

Our work is an instance of *machine learning for code* (ML for code). ML for code has been extensively studied in the software engineering community [2, 13, 22], including for optimizing computational performance [26]. There are several works that use code as input, usually in the form of a token sequence, and then train ML models for a variety of tasks (for example, code completion) [3, 9]. To the best of our knowledge, our work is the first to apply ML to GraphQL.

The database community has also studied *query performance prediction* (QPP) using machine learning techniques [1, 12, 18]. In contrast to these works, our approach focuses on GraphQL and uses a sound conservative upper bound on query cost as well as graph neural network features.

Our paper proposes a methodology for using ML to estimate the cost of GraphQL queries. We experimentally show that our ML approach outperform the leading existing static analysis approach using two commercial GraphQL APIs, namely GitHub and Yelp. We believe that an ML approach to query complexity estimation can be useful for both API providers and clients. API providers benefit by allowing them to loosen cost thresholds and better provision server resources, while clients benefit by allowing them to better understand the costs of their queries and what is allowable within their rate limits. In addition, our approach can be used in conjunction with other types of analyses to create an overall more robust API management system.

References

[1] M. Akdere, U. Cetintemel, M. Riondato, E. Upfal, and S. B. Zdonik. Learning-based query performance modeling and prediction. In *ICDE*, 2012.

[2] M. Allamanis, E. T. Barr, P. Devanbu, and C. Sutton. A survey of machine learning for big code and naturalness. *CSUR*, 51(4):81:1–81:37, 2018.

[3] M. Allamanis, M. Brockschmidt, and M. Khademi. Learning to represent programs with graphs. In *ICLR*, 2018.

[4] G. Baudart, M. Hirzel, K. Kate, P. Ram, and A. Shinnar. Lale: Consistent automated machine learning. In *AutoML@KDD*, 2020.

[5] G. Brito, T. Mombach, and M. T. Valente. Migrating to GraphQL: A practical assessment. *CoRR*, abs/1906.07535, 2019.

[6] G. Brito and M. T. Valente. REST vs GraphQL: A controlled experiment. In *ICSA*, 2020.

[7] L. Buitinck, G. Louppe, M. Blondel, F. Pedregosa, A. Mueller, O. Grisel, V. Niculae, P. Prettenhofer, A. Gramfort, J. Grobler, R. Layton, J. VanderPlas, A. Joly, B. Holt, and G. Varoquaux. API design for machine learning software: experiences from the scikit-learn project. *CoRR*, abs/1309.0238, 2013.

[8] A. Cha, E. Wittern, G. Baudart, J. C. Davis, L. Mandel, and J. A. Laredo. A principled approach to GraphQL query cost analysis. In *FSE*, 2020.

[9] C. Cummins, P. Petoumenos, Z. Wang, and H. Leather. End-to-end deep learning of optimization heuristics. In *PACT*, 2017.

[10] Google. Apigee. https://cloud.google.com/apigee, 2020.

[11] O. Hartig and J. Pérez. Semantics and complexity of GraphQL. In *WWW*, 2018.

[12] R. Hasan and F. Gandon. A machine learning approach to SPARQL query performance prediction. In *WI-IAT*, 2014.

[13] A. Hindle, E. T. Barr, Z. Su, M. Gabel, and P. T. Devanbu. On the naturalness of software. In *ICSE*, 2012.

[14] IBM. API Connect. https://www.ibm.com/cloud/api-connect, 2020.

[15] R. Kohavi. A study of cross-validation and bootstrap for accuracy estimation and model selection. In *IJCAI*, pages 1137–1145. Morgan Kaufmann, 1995.

[16] B. Komer, J. Bergstra, and C. Eliasmith. Hyperopt-sklearn. In *Automated Machine Learning - Methods, Systems, Challenges*. 2019.

[17] Linux Foundation. The GraphQL Foundation. https://foundation.graphql.org, 2019.

[18] R. Marcus and O. Papaemmanouil. Plan-structured deep neural network models for query performance prediction. In *VLDB*, 2019.

[19] G. Mavroudeas, G. Baudart, A. Cha, M. Hirzel, J. A. Laredo, M. Magdon-Ismail, L. Mandel, and E. Wittern. Learning GraphQL query cost (extended version). *CoRR*, abs/2108.11139, 2021.

[20] A. Narayanan, M. Chandramohan, R. Venkatesan, L. Chen, Y. Liu, and S. Jaiswal. graph2vec: Learning distributed representations of graphs. *CoRR*, abs/1707.05005, 2017.

[21] Prisma. Security and GraphQL. https://www.howtographql.com/advanced/4-security/, 2019.

[22] M. Rahman, D. Palani, and P. C. Rigby. Natural software revisited. In *ICSE*, 2019.

[23] RedHat. 3Scale. https://www.3scale.net, 2020.

[24] The GraphQL Foundation. Pagination – Complete Connection Model, 2021.

[25] The GraphQL Foundation. Who's Using GraphQL?, 2021.

[26] Z. Wang and M. O'Boyle. Machine learning in compiler optimization. *Proceedings of the IEEE*, 2018.

[27] E. Wittern, A. Cha, J. C. Davis, G. Baudart, and L. Mandel. An empirical study of GraphQL schemas. In *ICSOC*, 2019.

[28] D. H. Wolpert. Stacked generalization. *Neural networks*, 5(2):241–259, 1992.