Building Learned Federated Query Optimizers

Victor Giannakouris

Advised by: Immanuel Trummer
Cornell University, NY, USA

Abstract
The goal of this thesis is to introduce a new design for building federated query optimizers, based on machine learning. We propose a modular and flexible architecture, allowing a federated query optimizer to integrate with any database system that supports SQL, with close-to-zero engineering effort. By observing the performance of the external systems, our optimizer learns and builds cost models on-the-fly, enabling federated query optimization with negligible communication with the external systems. To demonstrate the potential of this research plan, we present a prototype of our federated query optimizer built on top of Spark SQL. Our implementation effectively accelerates federated queries, achieving up to 7.5x better query execution times compared to the vanilla implementation of Spark SQL.

Keywords
federated query processing, query optimization, machine learning,

1. Introduction

In the complex infrastructure of the modern "big data" ecosystem, data are usually distributed across multiple, diverse database systems. This has led to the development of federated query engines that enable users to simultaneously query multiple databases, using a unified, SQL-based interface. For instance, it is common for a data scientist to issue a query that joins a "small" table in a relational database with a bigger table that resides in a distributed data lake, like Amazon S3[1] or Delta Lake[2]. A number of federation engines developed by some of the largest database vendors, including Athena Federated Query[3], BigQuery[4], Spark SQL [1], Presto[5] or Dremio[6] over the last years, provide clear evidence for the importance of federated query engines. Taking into account the heterogeneity of the underlying systems that a federation engine integrates with, optimizing federated queries is one of the most challenging tasks for these systems. Usually, a federated query engine follows a one-size-fits-all approach, to connect with as many external database systems as possible. In summary, the query lifecycle in most federation systems (e.g. Spark, Presto) is quite simple. First, the federated engine transfers all the tables and views included in the query from the external database systems to federation execution engine through the network. A number of specific rule-based optimizations, e.g. subquery pushdown, might also be applied. Finally, the resulting query plan is executed in the federation engine.

Ideally, an efficient optimizer should be able to generate more sophisticated federated query plans, like in traditional databases. For example, instead of just pushing down selections to the external systems, it should consider pushing down larger parts of the federated query, like a join sub-tree. However, the heterogeneous nature and architectural differences of the external systems make the task of deciding *which parts of the query to push down and where* particularly complex. One of the main challenges is the complexity of estimating the subquery execution cost in an external system. This is a tricky task, for a number of factors. For example, due to the lack of access to statistics in the remote database system, estimating the local execution cost (in the external system) and result size is very challenging. Furthermore, the larger search space that derives from the additional planning decisions (i.e. where to execute each operator) due to federated execution, makes optimization even more challenging.

As a result, the majority of federation engines apply very few rule-based optimizations (i.e., selection pushdown), discarding optimization opportunities that would leverage the full potential of the external systems. While there have been some attempts to develop wrappers [2, 3] and custom cost models [4] to enable more fine-grained federated query plan generation, these approaches face the following challenges. First, developing custom wrappers and cost models for new systems is a tedious task, making the integration with new systems extremely difficult. Second, the communication with external systems to obtain cost estimates can slow optimization down, something known as *cost of costing* [5]. Taking into account these challenges, we try to answer the following question: *Can we develop a generic design for federated*

Proceedings of the VLDB 2022 PhD Workshop, September 5, 2022. Sydney, Australia
vg292@cornell.edu (V. Giannakouris)

[1]https://aws.amazon.com/s3
[2]https://delta.io/
[3]https://aws.amazon.com/athena
[4]https://cloud.google.com/bigquery
[5]https://prestodb.io/
[6]https://www.dremio.com

query optimizers that integrate with any external database system with 1. close-to-zero engineering effort and 2. minimal communication overhead during optimization?

To answer this question, we present an engine-agnostic approach for federated query optimization, that copes with the heterogeneity of the underlying infrastructure. Using machine learning, our solution allows the optimizer to *learn* the performance of the external database systems, without relying on any system-specific knowledge. Instead, it treats the external systems as black boxes. The key idea behind our approach is the following. In contrast to previous approaches that depend on cost estimates obtained from external systems (e.g. by parsing the output of EXPLAIN clause [6]), we use a *unified query vector model*, to represent queries in the vector space. Using this model, query trees can be simply transformed to vectors, and fed to various machine learning models in order to *learn* and *predict* the performance of the external systems. We can then leverage these learned cost models in order to develop a federated query optimizer that can easily connect to different systems, and has zero communication cost during optimization. In summary, our contributions are the following:

- We introduce a machine learning based architecture for federated query optimization that is able to integrate with any SQL-based database system with close-to-zero engineering effort.
- We present an implementation of our architecture on top of Spark SQL and we demonstrate how our system can effectively optimize federated queries over multiple systems, with zero communication overhead.
- We discuss an experimental evaluation that demonstrates our system's ability to effectively learn the performance of the external systems, while generated federated query plans always outperform Spark SQL.

The rest of the paper is organized as follows. Section 2 presents some background on federated query optimization, as well as a brief overview of the federated query processing research. Next, we present a prototype and architecture of a machine learning based federated query optimizer in Section 3. Then, we present some early experimental evaluation in Section 4. Finally, in Sections 5 and 6, we conclude and present ongoing and planned PhD research.

2. Background and Related Work

A federated query contains tables that reside in one, or multiple external database systems. The corresponding system is called a federation engine. A federated query processor follows a similar design to a traditional, single-engine query processor. The main extension is the ability to load data from external data sources. In this work, our main focus will be the optimizer. Federated query optimization is more complex than traditional query optimization, as it has to tackle the challenges that arise from the heterogeneity of the external systems.

Federated Query Processing. Federated query processing is not a new problem, and there has been extensive work over the last few decades [7, 8] that aims at optimizing queries across diverse data sources. For instance, Garlic [2] introduces a federated query optimizer based on a cost-based, dynamic-programming approach that uses data wrappers in order to integrate, and execute queries across different data sources. Recent works, like MuSQLE [4] and System-PV [3] present federated (a.k.a. multi-engine) query optimization approaches that perform both inter- and intra-engine optimizations. A similar approach based on data wrappers, is followed by another notable category of systems, called *polystores* [9, 10]. However, these approaches depend strongly on cost models, provided by the external systems. If a system does not provide cost estimates, the only way to integrate new systems is to implement the cost models. This process makes the integration with new systems impractical. Moreover, the communication needed with the external systems to obtain cost estimates of local query executions leads to excessive overheads that make the optimization process slow (cost-of-costing [5]).

Learned Query Optimization. The idea of learned query optimization has gained a lot of attention through the last years. Approaches like Neo [11] and Bao [12] are representative examples that showcase how machine learning can be utilized for self-driving query optimizers. However, most of these works are focusing mainly on single-node database systems. The closest approach to our system is the one presented by Liqi Xu et al. [6], which presents a supervised-learning approach for federated query optimization. The system learns the performance of the externally connected systems and it predicts the best federation engine, which can be any of the connected data sources. However, this work does not consider splitting further the complement queries, ignoring potential query plans that could achieve better performance. Furthermore, it relies on information (e.g., cost or row estimates), gathered by the EXPLAIN clause of the target data source. Continuously invoking EXPLAIN during query planning on multiple external data sources can make the process significantly slower, due to the communication overhead. Furthermore, the assumption that any of the connected data sources can be considered as the federation engine is not realistic, as it is not common for a DBMS to support reading data from external data sources.

The main weaknesses of prior work are the following.

First, integration with new systems can be tedious. Next, the cost estimation and interpretation for the external systems, as well as the continuous communication of the external systems during query optimization makes the process relatively slow, resulting in a high cost-of-costing. In the next section, we describe how we address these problems with machine learning, as well as a unified query vectorizer that maps queries to vectors.

3. ML-Based Federated Query Optimization

3.1. Architecture

Figure 1 depicts the architecture of our federated query optimizer prototype. In this section, we describe in detail each individual component of our system.

Query Vectorizer. The query vectorizer takes as input a parsed SQL query in its abstract syntax tree (AST) form and converts it into a vector that represents the semantics of the query, e.g., which tables are joined in the query or in which columns a `GROUP BY` operator is applied. In the current version, we follow a simple one-hot-encoding approach. Each query operator is represented by a vector. For example, the aggregation vector $g = [1, 0, 0, 1]$ represents a query in which the `GROUP BY` clause is applied on the first and the fourth columns. We combine all the vectors for all the predicates that we need to include in our search space and create a unified vector that represents the full query.

Cost Model Learning. In order to learn cost models, we use data obtained from past and current workloads. For each query, we keep its execution time and its vector form. We feed this data to a machine learning model, that predicts the execution time of future queries. Our current prototype trains its models with respect to execution time. However, the approach can be easily modified in order to take into account more objectives, like monetary cost in a cloud setting.

Federated Query Optimizer. Our federated query optimizer uses the first two components in order to generate near-optimal federated plans. First, it transforms the AST form of the query to a graph, in which each vertex represents one table and its location, while an edge represents a join between two tables. The optimizer works in two phases. The first pass, which we call *Location-First Search*, is an extension of the traditional Breadth-First-Search algorithm which traverses the graph, and generates a new binary tree with the following property. It is guaranteed that all vertices (tables) that reside in the same location, will be co-located under the same subtree (whenever that is possible, given the query semantics). The second pass processes each subtree at each location, and makes the required transformations, being advised by the learned cost model. For example, in some cases it might make sense to break a subtree that joins four tables into two subtrees that join two tables, in order to avoid computing a large result, and fetching that result to the federation engine over the network. This is achieved by adjusting a parameter called `join_limit`, which defines the maximum subquery size that can be pushed down for local execution to an external system.

Federated Rewriter. This module takes as input the federated query plan produced by the previous step. For each subtree that refers to a specific location (database system) of the query plan, it performs on-the-fly SQL code generation that will be pushed down to the external system for local execution. Finally, it generates the SQL code that will be executed in the federation engine, which will aggregate the results of each component query executed in the external locations.

3.2. Query Lifecycle

The query lifecycle follows the same steps as in the previous section. First, an SQL query is parsed and transformed to the corresponding AST form. Then, this query is passed to the vectorizer, which will transform it to the corresponding vector form. Next, the optimizer takes the AST of the query, it converts it to the corresponding graph, and it produces the final federated query plan. Finally, the federated plan is passed to the rewriter which will perform the necessary SQL code generation for the external systems and the federation engine. The query is then executed by leveraging both the external systems and the federation engine, and the result is returned to the user. At the end of execution, we also keep the query execution metrics, like the total execution time and the individual execution times of the subqueries in the external engines. We keep these metrics in order to re-train and refine our learned cost models and keep them up-to-date.

As mentioned in the previous sections, the key advantages of our federated query optimizer are the following. First, the query vectorizer allows our system to be easily integrated with any system that supports SQL, making the design specifics of the external system transparent to the federation engine. For example, in our implementation over Spark SQL, we use JDBC drivers in order to connect to the external system. Then, our system operates only on Spark's intermediate representation (AST) of the input query. This design will work exactly the same over any possible set of connected systems. Second, we minimize the potential communication overhead during optimization. By leveraging learned cost models, the cost estimates for each subquery in each external system are computed fast, while most of the optimization time is spent on useful work, i.e. plan enumeration.

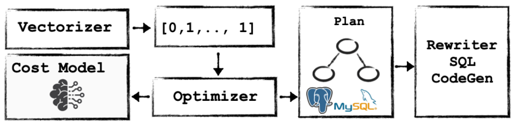

Figure 1: Optimizer Design and Query Lifecycle

Table 1
Execution Time

System	Avg. Exec Time (s)
Spark SQL	1.15
Optimal	0.122
Federated Optimizer	0.152

4. Preliminary Results

We evaluate our prototype experimentally and compare to Spark SQL. Our goal is to show that our design can effectively chose the right engine to execute each subquery, and the resulting plans outperform the vanilla implementation of Spark SQL. We evaluate our first prototype on a MacBook Pro with 16GB of memory and an Apple M1 chip. The infrastructure consists of a standalone, single-node Spark SQL cluster, one Postgres 14.0 instance and one MySQL 8.0.27 instance, everything running on the same machine. We used the TPC-H [7] (1GB) and the JOB [13] benchmarks.

Learned Cost Models. We first evaluate our optimizer's effectiveness in choosing the most performant execution engine. We use the TPC-H dataset for this experiment, making all tables available in all systems. First, we run a set of micro-benchmarks by randomly picking some of the TPC-H queries in order to collect data and train the cost models. Next, we run all TPC-H queries to evaluate our system. For each TPC-H query, the optimizer is assisted by the learned cost models, and decides whether it is better to push the full query into MySQL, Postgres, or fetch all the data and execute the query in Spark. For the sake of the experiment, we run each query in all three modes, which allows us to compare our optimizer's decision both with Spark SQL and the optimal decision (i.e. minimum execution time). The results are reported in Table 1, and depict the end-to-end execution time, including query processing and data fetch from the external systems. Using our optimizer, we achieve an average speedup of 7.5x compared to Spark SQL.

Optimized Queries. We use the JOB benchmark for this section. In this scenario, tables are randomly placed across MySQL and Postgres. For those queries, we experimented with changing the number of the maximum tables (`join_limit` parameter) that can be included in a subquery that is pushed-down for local execution to an external engine. This parameter needs adjustment for the following reason. Imagine pushing down a large join, that produces a very large intermediate result. Fetching this result from the external database system to the federation engine will result in excessive network overheads. Thus, the query performance will decrease. Using our very first prototype of a Reinforcement Learning based optimizer that tries out different values of the `join_limit` parameter, we conclude that for this setup the join limit should be either two or three. Sticking to these values, the average query execution time is 60% of time that the vanilla implementation of Spark SQL requires. This improvement is achieved mainly by splitting the query execution across Spark SQL and the external systems (e.g. pushing down part of the join), utilizing both the federation engine and the systems it connects to.

[7]https://www.tpc.org/tpch/

5. Research Plan

There is still work that needs to be done in order to release a fully-functional prototype of our ML-based optimizer. We presented individual parts of the system that we currently work on, and an early experimental evaluation of those components that showcase the current performance improvements that our system achieves.

Short-Term Goals. We are working towards the full integration of our optimizer with the learned cost models. Our main idea is to adopt a dynamic programming approach for plan enumeration, as in traditional databases. We plan to modify the algorithms and make them leverage the learned cost models in order to evaluate the cost of the enumerated federated, cross-database plans. We foresee the following challenges. First, in case of updates, the cost models will become outdated and might mislead the optimizer. In this case, the system should notice the performance degradation and re-calibrate models with respect to new data, possibly by re-training the models. Next, for queries with many joins, the large optimization space due to the multiple execution engine options might slow the optimizer down. We plan to develop specialized heuristics to prune the search space to maintain reasonable optimization time.

Long-Term Plan. Our current design already has some promising results for OLAP workloads on static data. However, as previously mentioned, relying on past query executions might be limiting if any updates are included in the workload, i.e. the learned cost models will become outdated. Furthermore, the larger search space that derives from the multiple-systems scenario might result in slow optimization for larger join queries. To address these challenges, we are working towards an extension, based on Reinforcement Learning. Instead of using cost estimates, this RL-based optimizer will try out different splits and join orders for the initial query during the *exploration* phase. Using RL, we can develop an adaptive optimizer, that will explore different subquery combinations and pushdowns across the different systems. This will not require prior training, and past workloads. Our goal is to create a solution that quickly adapts to data changes (in case of updates), while avoiding the costly enumerations in query planning.

Vision. A federated query engine should be flexible, connect easily to new data sources and hide the complexity of the underlying infrastructure from the user. Existing approaches on federated query optimization, like System-PV [3] and MuSQLE [4], still require a lot of manual work from the user. Our intuition is that the more generic, ML-based design that we develop as part of this PhD research will *democratize federated query optimization*, and make it possible to adopt these optimization schemes in the real world. While we implement our optimizer on top of Spark SQL, our design is generic enough and can be easily implemented in similar systems. The long-term goal of this PhD is to introduce new federated query optimization designs that are autonomous, and can be easily adopted by systems both in industry and academia.

6. Conclusions

We presented a PhD research plan that proposes a new federated query optimization design, based on machine learning. Our design is still under development and in an early stage. The preliminary results show that ML-based federated query optimization achieves notable performance improvements when compared to Spark SQL. In contrast to past works on federated query processing, our prototype leverages machine learning in order to cope with the heterogeneity of the underlying database systems. Our optimizer is able to connect new systems with close-to-zero engineering effort, and effectively optimize federated queries with minimal communication overhead.

References

[1] M. Armbrust, R. S. Xin, C. Lian, Y. Huai, D. Liu, J. K. Bradley, X. Meng, T. Kaftan, M. J. Franklin, A. Ghodsi, et al., Spark sql: Relational data processing in spark, in: Proceedings of the 2015 ACM SIGMOD international conference on management of data, 2015, pp. 1383–1394.

[2] V. Josifovski, P. Schwarz, L. Haas, E. Lin, Garlic: a new flavor of federated query processing for db2, in: Proceedings of the 2002 ACM SIGMOD international conference on Management of data, 2002, pp. 524–532.

[3] M. Karpathiotakis, A. Floratou, F. Özcan, A. Ailamaki, No data left behind: real-time insights from a complex data ecosystem, in: Proceedings of the 2017 Symposium on Cloud Computing, 2017, pp. 108–120.

[4] V. Giannakouris, N. Papailiou, D. Tsoumakos, N. Koziris, Musqle: Distributed sql query execution over multiple engine environments, in: 2016 IEEE International Conference on Big Data (Big Data), IEEE, 2016, pp. 452–461.

[5] A. Deshpande, J. M. Hellerstein, Decoupled query optimization for federated database systems, in: Proceedings 18th International Conference on Data Engineering, IEEE, 2002, pp. 716–727.

[6] L. Xu, R. L. Cole, D. Ting, Learning to optimize federated queries, in: Proceedings of the Second International Workshop on Exploiting Artificial Intelligence Techniques for Data Management, 2019, pp. 1–7.

[7] D. McLeod, D. Heimbigner, A federated architecture for database systems, in: Proceedings of the May 19-22, 1980, national computer conference, 1980, pp. 283–289.

[8] A. P. Sheth, J. A. Larson, Federated database systems for managing distributed, heterogeneous, and autonomous databases, ACM Computing Surveys (CSUR) 22 (1990) 183–236.

[9] J. Duggan, A. J. Elmore, M. Stonebraker, M. Balazinska, B. Howe, J. Kepner, S. Madden, D. Maier, T. Mattson, S. Zdonik, The bigdawg polystore system, ACM Sigmod Record 44 (2015) 11–16.

[10] J. LeFevre, J. Sankaranarayanan, H. Hacigumus, J. Tatemura, N. Polyzotis, M. J. Carey, Miso: souping up big data query processing with a multistore system, in: Proceedings of the 2014 ACM SIGMOD international conference on Management of data, 2014, pp. 1591–1602.

[11] R. Marcus, P. Negi, H. Mao, C. Zhang, M. Alizadeh, T. Kraska, O. Papaemmanouil, N. Tatbul, Neo: A learned query optimizer, arXiv preprint arXiv:1904.03711 (2019).

[12] R. Marcus, P. Negi, H. Mao, N. Tatbul, M. Alizadeh, T. Kraska, Bao: Making learned query optimization practical, ACM SIGMOD Record 51 (2022) 6–13.

[13] V. Leis, A. Gubichev, A. Mirchev, P. Boncz, A. Kemper, T. Neumann, How good are query optimizers, really?, Proceedings of the VLDB Endowment 9 (2015) 204–215.

Acknowledgments

This research project is supported by NSF grant IIS1910830 (“Regret-Bounded Query Evaluation via Reinforcement Learning”).