Rethinking Pluggable Federated Query Optimization: From Laptops to Data Warehouses

Victor Giannakouris
Cornell University
Ithaca, NY, USA
vg292@cornell.edu

Immanuel Trummer
Cornell University
Ithaca, NY, USA
it224@cornell.edu

ABSTRACT

Federated query optimization remains a persistent challenge in modern data systems due to the heterogeneity of execution engines and the overhead of estimating costs across external data sources. This challenge is further amplified by the rise of generative AI and retrieval-augmented generation (RAG) applications, which often require real-time access to diverse, distributed databases. In this paper, we introduce Dingo, a pluggable federated query optimizer that can be integrated into any SQL-based federated query engine with minimal effort. Existing federated optimizers typically suffer from two key limitations: (1) they depend on costly and often unreliable cost estimates from external systems, and (2) they require deep, system-specific integration with each external database system. Dingo addresses both issues by using a learned cost model trained on past query executions to avoid remote estimation, and by operating entirely outside the query engine, creating subquery views in external databases and rewriting queries to enforce pushdown. Dingo's generic architecture supports seamless integration across a wide range of systems, from lightweight in-process engines like DuckDB and DataFusion to cloud-scale platforms such as Redshift and Spark SQL. Evaluation on the Join Order Benchmark demonstrates that Dingo achieves average query speedups of up to 5.5x, while requiring fewer than 50 lines of integration code per engine.

VLDB Workshop Reference Format:
Victor Giannakouris and Immanuel Trummer. Rethinking Pluggable Federated Query Optimization: From Laptops to Data Warehouses. VLDB 2025 Workshop: Third International Workshop on Composable Data Management Systems.

1 INTRODUCTION

In the complex infrastructure of the modern lakehouse architecture, data are usually distributed across multiple, diverse data systems. This has led to the development of query engines with federated processing capabilities, enabling users to simultaneously query multiple databases, using a unified, SQL-based interface. As an example, it is common for a user or an application to issue a query that joins a "small" table in a relational database with a bigger table that resides in a cloud service, like Amazon S3[1] or Delta Lake[2]. Over the last years, a number of data warehouses developed by some of the largest database vendors have implemented federated query features, including Redshift [2], Spark SQL [1], DuckDB [21], Presto [23], Athena Federated Query[3], BigQuery[4] or Dremio [5]. This fact provides clear evidence for the popularity of federated query engines. Taking into account the heterogeneity of the underlying systems that a federated 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 federated systems (e.g. Redshift, Presto) is straightforward. First, the federated engine transfers all the tables and views included in the query from the external database systems to the federated execution engine through the network. A number of specific rule-based optimizations, e.g. subquery pushdown to the external database, might also be applied. Finally, the resulting query plan is executed in the federated 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 filters to the external databases, the optimizer could 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 cardinality is very challenging. Furthermore, the larger search space that derives from the additional planning decisions (i.e., *if and where to push down* a subquery) due to federated execution, makes optimization even more challenging. As a result, the majority of federated engines apply very few rule-based optimizations, like filter pushdown.

1.1 Always Pushing Down Does Not Fit All

Let us consider the straight-forward rule-based pushdown approach to optimize federated queries. We analyze the query plan, and do the required transformations in order to pushdown everything that is possible to be pushed down, including larger parts of the query tree, like joins. In the following two examples, we demonstrate that the problem is more complex and it cannot always be solved by applying such rules. We consider a federated version of the

Proceedings of the VLDB Endowment. ISSN 2150-8097.

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

Join Order Benchmark[6], in which tables are located in an AWS RDS infrastructure, consisting of one MySQL, and one Aurora PostgresSQL instance. All tables are loaded as DataFrames in Python using ConnectorX [26], and the queries are executed using DuckDB. For both queries that we will be examining in this example, the main bottleneck is the join of $cast\_info$ with some other table, due to its relatively large size. We use a random placement of the tables depicted in Table 1(we omit the rest of the tables for space efficiency):

Table 1: Table Locations

Database	Tables
PostgreSQL	cast_info, role_type, company_name, name, movie_keyword
MySQL	aka_name, title

We start with query 8c[7]. In this query, bottleneck is the join of $cast\_info$ with either $title$ or $role\_type$. As $cast\_info$ is colocated with $role\_type$ in Postgres, we can only consider pushing down $cast\_info \bowtie role\_type$ to Postgres, or, we can fetch both tables as DataFrames and execute the join in DuckDB. As we can see in Figure 1, query 8c can be 10 times faster if subquery pushdown is utilized. The reason is that $cast\_info \bowtie role\_type$ is quickly executed in Postgres, while the filter $\sigma_{rt.role='writer'}$ on row_type results to an intermediate result much smaller even from the $cast\_info$ itself that we would need to fetch through the network. On the other hand, this is not the case for query 16a. In this query, the only pushdown we can consider along with $cast\_info$ is the join $cast\_info \bowtie movie\_keyword$. However, by performing this pushdown, the query becomes 1.7 times slower compared to the vanilla implementation. The reason is that $cast\_info \bowtie movie\_keyword$ creates a large intermediate result and the data transfer becomes a bottleneck. Thus, for this query it is preferable to fetch all tables and execute in DuckDB. This experiment demonstrates that pushing down arbitrarily does not always work as desired, and the optimizer should take into consideration more factors when deciding which parts of the queries should be pushed down, including intermediate result size and the performance of the external engine. A learned optimizer that could predict these two would be able to generate an efficient query plan consisting of only beneficial pushdown decisions.

1.2 Pluggable Federated Query Optimization

Building query optimizers is provably one of the most challenging tasks in database system implementation. For example Spark SQL introduced a cost-based query optimizer three years after its initial release, while there have been changes in PostgreSQL despite its 20 years old maturity. Similarly, most of the federated query optimizer implementations are tightly-coupled with the federated query engine, and the integration with either new federated engines or external database systems is impossible. In summary, we

[6]https://github.com/gregrahn/join-order-benchmark
[7]https://github.com/gregrahn/join-order-benchmark/blob/master/8c.sql

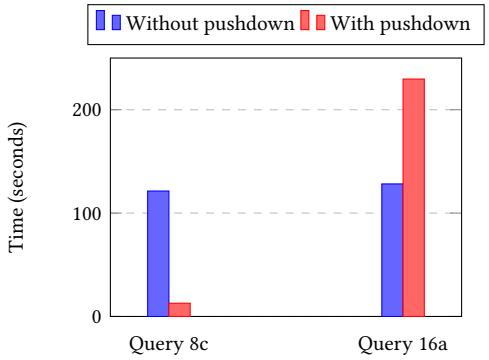

Figure 1: Execution times for queries 8c and 16a with and without pushdown.

identify the following challenges in federated query optimizer implementation.
(1) **Hard Integration.** The design of existing federated query optimizers make integration with external engines impractical. While these works implement external engine wrappers and custom cost models to enable more fine-grained federated query plan generation, developing custom wrappers and cost models for new systems is a tedious task, making the integration with new systems extremely difficult and time-consuming.
(2) **Expensive Optimization.** Next, as already demonstrated in previous works, the communication with external systems to obtain cost estimates can slow optimization down, something known as *cost of costing* [4]. This cost becomes even higher due to the vast search space of the potential federated query plans.
(3) **Lack of Portability.** Existing federated query optimization approaches are implemented on top of a specific engine. For example, Garlic [16] can only work on Db2, while MuSQLE [12] and SystemPV [17] are implemented for Spark SQL. As a result, applying these optimizers to new systems requires a significant number of changes and modifications in order to integrate them with more federated engines.

All these challenges have led to some interesting directions for more pluggable optimizer architectures. One of them is the idea of a Query Optimizer as a Service [15] for cloud databases, which isolates the cardinality estimator, the cost model, and the query planner as external services that can be used from the query optimizer of multiple different systems. At the same time, transfer learning has been proposed as a tool for *transferring knowledge from one query optimizer to another* [28]. All these directions could be extremely beneficial for a federated query optimizer. Combining all these challenges and ideas, we try to answer the following question:

"Can we build a pluggable federated query optimizer that can integrate with any external database and federated query engine with 1. minimum engineering effort and 2. minimum communication overhead during optimization?"

To address this question, we present Dingo [13](Fe**D**erated, Machine Learn**ing**-based Query **O**ptimizer), an engine-agnostic federated query optimizer, that copes with the heterogeneity of the

underlying infrastructure. Using machine learning, Dingo *learns* 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.

Low-overhead cost estimation. In contrast to previous approaches that depend on cost estimates obtained from external systems (e.g. by parsing the output of `EXPLAIN` clause [27]), we use a *unified query vector model*, to represent queries in the vector space. This model focuses on the query tree level only, making it flexible enough to work with any external system that supports SQL. 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.

Easy integration. At the same time, Dingo follows a flexible federated plan execution scheme that allows it to be easily used as an external optimizer over any federated query engine that is connected over any set of external database systems. Given a federated query plan, Dingo creates views of the subqueries that will be pushed down to the external databases, and rewrites the initial, federated query by replacing the pushed-down query parts with the view names. Thus, when a view name is referenced from the outer query, Dingo triggers the local execution of the subquery to the external engine. Moreover, Dingo leverages ideas from both transfer learning and online learning, enabling it to reuse pre-existing knowledge over new federated engines, as well as to adapt over time to workload changes by improving its model. Dingo's ability to adapt and optimize queries over multiple federated query engines makes it easily pluggable over query engines with federation capabilities with minimum effort. We demonstrate this by evaluating Dingo over a set of the most widely known cloud databases, including AWS Redshift and Spark SQL on Elastic MapReduce (EMR).

In summary, our contributions are the following:

- We present Dingo, the first learned federated query optimizer that is able to integrate with any SQL-based database system with minimum engineering effort.
- We introduce a pluggable federated query optimization and plan execution scheme based on view creation, enabling Dingo to optimize queries over any federated engine that is connected over any set of external database systems that support SQL execution via JDBC connection.
- We showcase Dingo's ability to operate as an external optimizer over a variety of federated query engines, ranging from cloud databases (Redshift and Spark SQL) to in-process query engines (DuckDB).
- We show how Dingo leverages ideas from transfer and online learning, enabling it to re-use knowledge from known federated query engines to unseen ones, and gradually adapt to the current workload by periodically re-training its cost models.
- We present a thorough experimental evaluation over Spark SQL and Redshift, and we showcase that Dingo can improve the average query execution time of the Join Order Benchmark up to 5.5 times.

System Overview. In this work, our primary focus is on building a pluggable and composable query optimizer for federated query engines. Most of our design decisions center around modularity—for example, enabling easy selection or extension of the cost model. As a result, we deliberately made simplifications in certain components to highlight the architectural contribution. Specifically, to avoid complications from cardinality or cost estimation errors, we assumed a static workload and dataset. Additionally, we used a straightforward learned cost model that vectorizes queries and feeds them into a multi-layer perceptron regressor for training and inference. We implement Dingo on top of Apache Calcite [3], by modifying the standard Volcano/Cascades optimization scheme to make the optimizer aware of the table locations (external engines), and the possible execution engines for each operator. For the learned cost models, Dingo uses the Deep Java Library[8]. For our current prototype, we are making the *assumption of a static workload and data*, using the Join Order Benchmark. Figure 2 depicts Dingo's architecture, which consists of the following five components: The *query vectorizer*, the *profiler*, the *learned cost models*, the *federated query optimizer*, and the *plan executor (query rewriting and view creator)*.

Query Vectorizer. The query vectorizer takes an input SQL query in its Abstract Syntax Tree (AST) form. It extracts the query semantics and outputs the corresponding query vector. To construct the vectors, we use unique identifiers to index the tables and columns. For example, given the projection vector P, the element P_i denotes whether the column with id i is included in the `SELECT` clause of the query. Our vectorization scheme captures both query and execution location information. For instance, joins are represented using an adjacency matrix J, where the $J_{l,r} = e$ indicates a join between tables l and r which will be executed in the engine e. The final query vector is a flattened version of the union of the vectors of each relational operator, as well as the join adjacency matrix.

Profiler. The profiler collects metrics during query execution that are used later to train the learned cost models. In order to generate more training samples, the profiler collects execution metrics from the individual parts of the query, whenever that is possible. For example, given the federated query plan depicted in Figure 4, the profiler will create three training samples, that is, one of the whole query execution in the federated engine (`FederatedJoin`), one for the join that is being pushed down to Postgres (`ExternalJoin`) and one of the table scan in MySQL (`ExternalScan`). Using this approach, Dingo is able to improve its learned cost model both for the federated engine, as well as for the external ones.

Learned Cost Model. The learned cost model is one of the key components of Dingo that makes the plan enumeration fast. Instead of communicating with the external engines, Dingo *asks* the learned cost model for subquery cost estimations. The learned cost model is completely integrated with our query optimizer, and it is invoked during plan enumeration in order to compare the cost of the candidate intermediate plans. In this prototype, we are using a

[8] https://djl.ai/

very simple predictive model based on a neural network with a single hidden layer of 1024 units, using the ReLU activation function. We treat the cost estimation as *regression problem*, and our model predicts the query execution time in seconds. Our experimental evaluation showcases that even using a simple linear neural network like this, the optimizer learns *effective optimization decisions* quickly using periodical retraining and outperforms the vanilla versions of the integrated federated query engines.

Federated Query Optimizer. The federated query optimizer extends the Calcite Volcano/Cascades planner, using some of the default transformation rules, as well as some custom *federated rules* that we developed in order to enable the optimizer to make location-aware decisions and decide which parts of the query should be pushed down for local execution to the external engines (see Figure 4). Beyond the transformation rules, our planner is integrated with the vectorizer and the learned cost model in order to vectorize the candidate plans and estimate their costs during enumeration.

Executor — Rewriter. Dingo implements a generic plan executor, making it able to integrate with minimal effort with most of the well-known distributed query engines. Currently, Dingo can be used as an external federated query optimizer for AWS Redshift, Spark SQL and DuckDB. The plan executor analyzes the federated query plan and for each external operator (e.g. `ExternalJoin`) it generates the equivalent SQL code, with which it generates a view in the external engine. Then, it replaces the parts of the outer query that are pushed down with `FederatedScan` operators that load the result of the created view. This triggers the local execution of the subquery to the external query engine. The initial outer query is then rewritten, and all the parts of the query that are pushed down are replaced with the corresponding view names. For instance, given the query plan depicted in Figure 4, the final rewritten federated query will become:

```
SELECT *
FROM federatedScan1 fs1, title
WHERE fs1.movie_id = title.id
```

where the reference to `fs1` enforces the local execution of the join between `cast_info` and `role_type` to Postgres and fetches the result to the federated execution engine, which then joins it with the table `title`. A similar approach is followed for embedded engines (e.g. DuckDB or DataFusion) by generating the equivalent Python code using DataFrames (see Figure 3).

2 EXPERIMENTAL EVALUATION

In this section we present the results of our experimental evaluation. We evaluate Dingo as an external optimizer for Spark SQL and Amazon Redshift using the Join Order Benchmark (JOB).

Setup. All experiments were conducted on AWS. For Spark SQL, we deployed a 5-node Elastic MapReduce (EMR) cluster with m2.xlarge instances. The JOB tables were arbitrarily placed across Amazon Aurora PostgreSQL and MySQL (RDS). Table 2 depicts the location of each table. We consider both online and offline training scenarios for Dingo's learned cost model. For cost estimation, we use a simple multi-layer perceptron regression model implemented with the Deep Java Library (DJL). As emphasized earlier, the focus of this work is on building a pluggable architecture for federated query optimization, including pluggable cost models. While our current

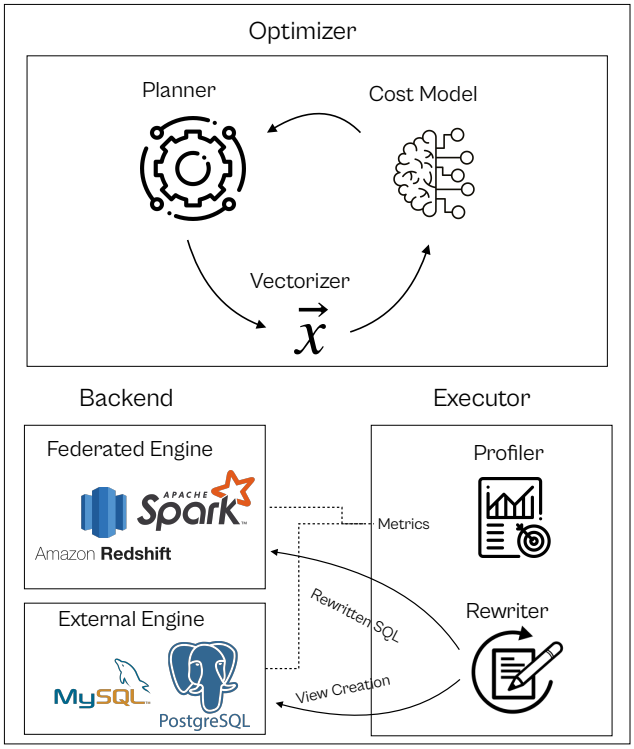

Figure 2: Dingo's Architecture

```
fs0 = cx.read_sql(psql_conn, """
    SELECT cast_info.movie_id
    FROM cast_info
    INNER JOIN (SELECT *
                FROM role_type
                WHERE role = 'writer') AS t
   ON cast_info.role_id = t.id')"""

fs1 = cx.read_sql('SELECT id, title FROM title')

query = """
    SELECT MIN(federatedScan1.title)
    FROM federatedScan0
    INNER JOIN federatedScan1
    ON federatedScan0.movie_id = federatedScan1.id"""
```

Figure 3: Pandas Federated Plan Code

```
FederatedJoin(condition=[=($2, $9)], joinType=[inner])
  ExternalJoin(condition=[=($6, $7)], engine=[Postgres])
    ExternalScan(table=[cast_info], engine=[Postgres])
    ExternalFilter(condition=[=($1, 'writer')])
      ExternalScan(table=[role_type], engine=[Postgres])
  ExternalScan(table=[title], engine=MySQL)
```

Figure 4: Federated Query Plan

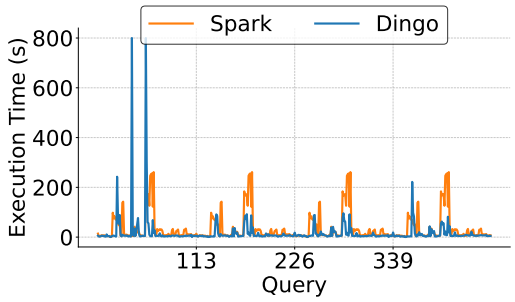

Figure 5: Online Training

model is intentionally kept simple, Dingo's design supports easy integration with more advanced techniques for learned cardinality estimation and cost modeling, making it a flexible foundation for future research.

Table 2: Table Mapping by Engine

Engine	Tables
PostgreSQL	aka_title, cast_info, char_name, comp_cast_type, company_name, company_type, complete_cast, keyword, link_type, movie_companies, movie_info, movie_info_idx, movie_keyword, movie_link, name, person_info, role_type
MySQL	aka_name, info_type, kind_type, title

Online Training. In the online scenario, Dingo encounters each query for the first time. The learned cost models are retrained every n queries, where n is a user-defined parameter (we use $n = 10$). Figure 5 illustrates how Dingo adapts over time. We repeated the 113 JOB queries four times, simulating multiple rounds of a repetitive workload. Initially, Dingo may make suboptimal pushdown decisions, reflected in early performance spikes. However, through periodic retraining, Dingo adapts quickly, outperforming vanilla Spark SQL. On average, Dingo achieves a 5.5x speedup, with some queries experiencing up to 68x improvements. These gains are attributed to Dingo's ability to predict pushdown effectiveness and avoid unnecessary data movement across engines.

Offline Training. This scenario demonstrates Dingo's performance when provided with a well-trained, highly accurate cost model. Our goal here is to showcase the effectiveness of Dingo's architecture across different systems when cost estimation is not a bottleneck. To prepare the model, we trained Dingo offline using a random selection of queries and multiple alternative plans per query. This allowed Dingo to explore the plan space and learn both effective and ineffective pushdown decisions prior to evaluation. Figures 6 and 7 show the cumulative execution time after each query, with the Y-axis representing total time in seconds and the X-axis indicating the query number. In both Spark SQL and Redshift, Dingo significantly accelerates execution: the performance gap becomes evident within the first 20 queries. Table 3 summarizes

Table 3: Performance (Speedup) Improvement Summary

System	Workload	Average	Max
Redshift	1.87	2.31	8.69
Spark SQL	2.87	5.29	68.09

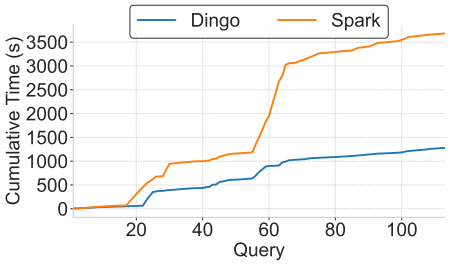

Figure 6: Spark SQL vs Dingo (AWS EMR)

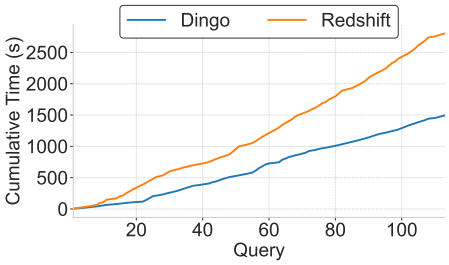

Figure 7: Redshift vs Dingo (AWS EMR)

the performance gains, reporting the **total workload speedup**, the **average speedup per query**, and the **maximum single-query speedup**. Notably, Dingo improves workload performance by 2.87x in Spark SQL and 1.87x in Redshift, with a maximum individual query improvement of over 68x in Spark SQL.

Across all engines, the average optimization time per query remains low (only *0.65 seconds*), underscoring Dingo's practicality even in latency-sensitive workloads.

3 RELATED WORK

Federated Query Optimization. There has been a significant effort on federated query optimization research [14, 19, 25] that aims at optimizing queries across diverse data sources. For instance, Garlic [16] 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. On the other hand, MuSQLE [12] and System-PV [17] introduce federated query optimization approaches that perform optimizations both in the external, and the federated query engines. Similar approaches are being followed in *polystores* [7, 18]. These approaches depend strongly on cost model implementation for the external systems. 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). To the best of our knowledge, there has been only a single approach by Liqi Xu et al. [27] on learned federated query optimization. However, this work does not consider splitting further the complement queries, ignoring potential query plans that could achieve better performance. Other research has investigated in-situ cross-database processing [9], which is orthogonal to our focus. We target systems that use a dedicated mediator for federated query processing. Finally, BRAD [11] explores cost- and performance-based optimization across cloud databases, but with a different scope. Rather than decomposing a single query across multiple systems, BRAD routes entire queries to a single best-fit engine and holistically optimizes infrastructure provisioning.

Composable Architectures. There has been extensive work on composable data system architectures across different components. For example, Apache Calcite [6] is a widely used framework for building query optimizers and has been adopted in several production systems. Velox [20], on the other hand, provides a high-performance implementation of low-level query operators and can serve as the execution backend for new systems. Similarly, Gluten [24] enables Java-based engines to offload execution to native libraries such as Velox. Furthermore, CompoDB [10] offers standardized interfaces for constructing modular database system architectures. Finally, there has also been significant work on data exchange interfaces across heterogeneous data sources, such as E-Scan [22] and XDBC [8].

4 LESSONS LEARNED AND VISION

Building a query optimizer is both challenging and rewarding. In this section, we briefly reflect on our experience developing Dingo and highlight the key takeaways.

4.1 Portability

We initially built Dingo on top of Spark SQL, assuming a query engine capable of JDBC connections to external systems. This setup enabled us to construct federated query plans and enforce subquery pushdown by routing parts of the plan to external engines. However, we quickly realized that not all execution engines support such functionality. To enhance portability and ensure compatibility with any engine that supports basic table federation, we adopted a simpler and more general approach: generating views in external systems for each subquery targeted for pushdown. We then rewrite the original federated query by replacing each pushed-down subquery with a reference to its corresponding view. Executing the rewritten query on the federated engine automatically triggers the execution of the subquery in the appropriate external system, without requiring tight integration or specialized connectors.

4.2 Slow Retrieval of External Join Results

In some execution engines, retrieving the result of a pushed-down subquery that includes a join can be prohibitively slow. For example, Spark SQL by default fetches the result of such a subquery using a single partition. When the intermediate result is large, this becomes a significant bottleneck in the query's critical path. A common workaround is to parallelize the retrieval by setting the `partitionColumn`, `lowerBound`, `upperBound`, and `numPartitions` options. This allows Spark to distribute the load across multiple workers. However, this approach shifts the burden to the external database, which must now execute `numPartitions` join queries, each restricted to a range of values on the `partitionColumn`. If the column's value distribution is skewed, this can lead to severe load imbalance across partitions and suboptimal parallelism.

4.3 Subquery Pushdown is Critical

Determining which parts of a query to group and push down is a critical decision. A well-chosen pushdown can accelerate query execution by orders of magnitude, while a poor choice may overwhelm the system or result in massive intermediate data that slows everything down. In some cases, pushing down a large or poorly filtered subquery may perform worse than simply fetching all base tables into the federated engine and executing the query there. Optimizers must carefully weigh the trade-offs when constructing federated plans.

4.4 Regression vs Classification

While Dingo currently relies on a regression-based learned cost model, our ongoing experimentation suggests that a classification-based approach may be more promising. Our existing model uses a single regressor to estimate subquery costs across different engines, with the target engine encoded as a feature in the query vector. However, we found that this setup often leads to poor generalization. In particular, estimating intermediate result sizes proved difficult, and the model frequently produced large errors. In contrast, our preliminary experiments indicate that a classifier can more reliably decide whether a given subquery **should be pushed down or not**. Instead of predicting absolute costs, the classifier performs a binary decision that aligns more directly with the optimizer's needs. Prior work has shown the effectiveness of pairwise plan comparison [5], but to our knowledge, this technique has not yet been explored in the context of federated query optimization. As part of future work, we plan to extend Dingo with support for classification-based decision-making along these lines.

5 CONCLUSIONS

In this paper, we presented our experience building Dingo, a learned federated query optimizer. Prior approaches to federated optimization often incurred significant development overhead due to the need for custom data wrappers and cost models tailored to each external system. Additionally, these solutions lacked generality, making integration with new engines difficult and time-consuming. Dingo addresses these challenges through a pluggable architecture that supports subquery pushdown via automatic view generation and query rewriting. By leveraging learned cost models, Dingo eliminates the communication overhead during optimization and removes the need for engine-specific wrappers or cost estimators. Our evaluation shows that Dingo can significantly accelerate real-world commercial systems such as Amazon Redshift and Spark SQL, achieving up to a 5x speedup on the Join Order Benchmark workload.

ACKNOWLEDGMENTS

This material is based upon work supported by the National Science Foundation under Award No. 2239326.

REFERENCES

[1] Michael Armbrust, Reynold S Xin, Cheng Lian, Yin Huai, Davies Liu, Joseph K Bradley, Xiangrui Meng, Tomer Kaftan, Michael J Franklin, Ali Ghodsi, et al. 2015. Spark sql: Relational data processing in spark. In *Proceedings of the 2015 ACM SIGMOD international conference on management of data*. 1383–1394.

[2] Nikos Armenatzoglou, Sanuj Basu, Naga Bhanoori, Mengchu Cai, Naresh Chainani, Kiran Chinta, Venkatraman Govindaraju, Todd J Green, Monish Gupta, Sebastian Hillig, et al. 2022. Amazon Redshift re-invented. In *Proceedings of the 2022 International Conference on Management of Data*. 2205–2217.

[3] Edmon Begoli, Jesús Camacho-Rodríguez, Julian Hyde, Michael J Mior, and Daniel Lemire. 2018. Apache calcite: A foundational framework for optimized query processing over heterogeneous data sources. In *Proceedings of the 2018 International Conference on Management of Data*. 221–230.

[4] Amol Deshpande and Joseph M Hellerstein. 2002. Decoupled query optimization for federated database systems. In *Proceedings 18th International Conference on Data Engineering*. IEEE, 716–727.

[5] Bailu Ding, Sudipto Das, Ryan Marcus, Wentao Wu, Surajit Chaudhuri, and Vivek R Narasayya. 2019. Ai meets ai: Leveraging query executions to improve index recommendations. In *Proceedings of the 2019 International Conference on Management of Data*. 1241–1258.

[6] Mark Dodds and Khuzaima Daudjee. 2025. Apache Ignite+ Calcite Composable Database System: Experimental Evaluation and Analysis. (2025).

[7] Jennie Duggan, Aaron J Elmore, Michael Stonebraker, Magda Balazinska, Bill Howe, Jeremy Kepner, Sam Madden, David Maier, Tim Mattson, and Stan Zdonik. 2015. The bigdawg polystore system. *ACM Sigmod Record* 44, 2 (2015), 11–16.

[8] Haralampos Gavriilidis, Kaustubh Beedkar, Matthias Boehm, and Volker Markl. 2025. Fast and Scalable Data Transfer Across Data Systems. *Proceedings of the ACM on Management of Data* 3, 3 (2025), 1–28.

[9] Haralampos Gavriilidis, Kaustubh Beedkar, Jorge-Arnulfo Quiané-Ruiz, and Volker Markl. 2023. In-situ cross-database query processing. In *2023 IEEE 39th International Conference on Data Engineering (ICDE)*. IEEE, 2794–2807.

[10] Haralampos Gavriilidis, Lennart Behme, Christian Munz, Varun Pandey, and Volker Markl. 2025. CompoDB: A Demonstration of Modular Data Systems in Practice. (2025).

[11] X Yu Geoffrey, Ziniu Wu, Ferdi Kossmann, Tianyu Li, Markos Markakis, Amadou Ngom, Samuel Madden, and Tim Kraska. 2024. Blueprinting the Cloud: Unifying and Automatically Optimizing Cloud Data Infrastructures with BRAD. *Proceedings of the VLDB Endowment* 17, 11 (2024), 3629–3643.

[12] Victor Giannakouris, Nikolaos Papailiou, Dimitrios Tsoumakos, and Nectarios Koziris. 2016. MuSQLE: Distributed SQL query execution over multiple engine environments. In *2016 IEEE International Conference on Big Data (Big Data)*. IEEE, 452–461.

[13] Victor Giannakouris and Immanuel Trummer. 2022. Building Learned Federated Query Optimizers. In *CEUR workshop proceedings*, Vol. 3186.

[14] Dennis Heimbigner and Dennis McLeod. 1985. A federated architecture for information management. *ACM Transactions on Information Systems (TOIS)* 3, 3 (1985), 253–278.

[15] Alekh Jindal and Jyoti Leeka. 2022. Query Optimizer as a Service: An Idea Whose Time Has Come! *ACM SIGMOD Record* 51, 3 (2022), 49–55.

[16] Vanja Josifovski, Peter Schwarz, Laura Haas, and Eileen Lin. 2002. Garlic: a new flavor of federated query processing for DB2. In *Proceedings of the 2002 ACM SIGMOD international conference on Management of data*. 524–532.

[17] Manos Karpathiotakis, Avrilia Floratou, Fatma Özcan, and Anastasia Ailamaki. 2017. No data left behind: real-time insights from a complex data ecosystem. In *Proceedings of the 2017 Symposium on Cloud Computing*. 108–120.

[18] Jeff LeFevre, Jagan Sankaranarayanan, Hakan Hacigumus, Junichi Tatemura, Neoklis Polyzotis, and Michael J Carey. 2014. MISO: souping up big data query processing with a multistore system. In *Proceedings of the 2014 ACM SIGMOD international conference on Management of data*. 1591–1602.

[19] Dennis McLeod and Dennis Heimbigner. 1980. A federated architecture for database systems. In *Proceedings of the May 19-22, 1980, national computer conference*. 283–289.

[20] Pedro Pedreira, Orri Erling, Masha Basmanova, Kevin Wilfong, Laith Sakka, Krishna Pai, Wei He, and Biswapesh Chattopadhyay. 2022. Velox: meta's unified execution engine. *Proceedings of the VLDB Endowment* 15, 12 (2022), 3372–3384.

[21] Mark Raasveldt and Hannes Mühleisen. 2019. Duckdb: an embeddable analytical database. In *Proceedings of the 2019 International Conference on Management of Data*. 1981–1984.

[22] Viktor Sanca and Anastasia Ailamaki. 2023. E-scan: Consuming contextual data with model plugins. In *Joint Workshops at 49th International Conference on Very Large Data Bases (VLDBW'23)*.

[23] Raghav Sethi, Martin Traverso, Dain Sundstrom, David Phillips, Wenlei Xie, Yutian Sun, Nezih Yegitbasi, Haozhun Jin, Eric Hwang, Nileema Shingte, et al. 2019. Presto: SQL on everything. In *2019 IEEE 35th International Conference on Data Engineering (ICDE)*. IEEE, 1802–1813.

[24] Akash Shankaran, George Gu, Weiting Chen, Binwei Yang, Chidamber Kulkarni, Mark Rambacher, Nesime Tatbul, and David E Cohen. 2023. The Gluten Open-Source Software Project: Modernizing Java-based Query Engines for the Lakehouse Era.. In *VLDB Workshops*.

[25] Amit P Sheth and James A Larson. 1990. Federated database systems for managing distributed, heterogeneous, and autonomous databases. *ACM Computing Surveys (CSUR)* 22, 3 (1990), 183–236.

[26] Xiaoying Wang, Weiyuan Wu, Jinze Wu, Yizhou Chen, Nick Zrymiak, Changbo Qu, Lampros Flokas, George Chow, Jiannan Wang, Tianzheng Wang, et al. 2022. ConnectorX: accelerating data loading from databases to dataframes. *Proceedings of the VLDB Endowment* 15, 11 (2022), 2994–3003.

[27] Liqi Xu, Richard L Cole, and Daniel Ting. 2019. Learning to optimize federated queries. In *Proceedings of the Second International Workshop on Exploiting Artificial Intelligence Techniques for Data Management*. 1–7.

[28] Yunjia Zhang, Yannis Chronis, Jignesh M Patel, and Theodoros Rekatsinas. [n.d.]. Can Transfer Learning be used to build a Query Optimizer? ([n. d.]).