Accio: Bolt-on Query Federation

Xiaoying Wang
xiaoying_wang@sfu.ca
Simon Fraser University

Jiannan Wang
jnwang@sfu.ca
Huawei Technologies,
Simon Fraser University

Tianzheng Wang
tzwang@sfu.ca
Simon Fraser University

Yong Zhang
yong.zhang3@huawei.com
Huawei Technologies

ABSTRACT

Data scientists today often need to analyze data from various places. This makes it necessary for corresponding engines to support query federation (i.e., the ability to perform SQL queries over data hosted in different sources). Although many systems come with federation capabilities, their implementations are tightly coupled with the core engine design. This not only increases complexity and reduces portability across engines, but also often leads to performance issues by missing optimization opportunities. This paper proposes Accio, a new "bolt-on" approach to query federation. Accio is a middleware library that decouples query federation from the target system. It enables two key optimizations—join pushdown and query partitioning—via a declarative interface that can be easily leveraged by different engines. Our experience of adapting five popular data science query engines shows that Accio can outperform existing approaches by orders of magnitude in various scenarios without the need for any intrusive changes or extra maintenance.

PVLDB Reference Format:
Xiaoying Wang, Jiannan Wang, Tianzheng Wang, and Yong Zhang. Accio: Bolt-on Query Federation. PVLDB, 18(7): 2126 - 2135, 2025.
doi:10.14778/3734839.3734849

PVLDB Artifact Availability:
The source code, data, and/or other artifacts have been made available at https://github.com/sfu-db/accio.

1 INTRODUCTION

In recent years, a variety of query engines have become popular among data scientists, such as Pandas [58], Spark [28], DuckDB [60], ClickHouse [8] (chDB [7]), Dask [62], Modin [59], DataFusion [42], Polars [16], etc. These engines are typically open-source and provide an intuitive SQL+dataframe Python interface. In this paper, we categorize these systems as *DS engines (data science query engines)*, as they are popular for handling data science tasks such as exploratory data analysis, data integration, feature engineering, and building ML pipelines. Since the data required to perform these tasks often reside in different data sources, data collection and analysis from various sources (i.e., query federation) becomes a common necessity [1, 4, 13, 17]. Consider an exploratory data analysis example in Figure 1:

Example 1. *A data scientist working for an E-Commerce company is investigating the cause of a sales decline in the previous month. Using Polars, she analyzes the purchase history alongside user profiles, requiring joining tables stored in separate DBMSs maintained by the sales and CRM departments, respectively. Since she only has read access to these databases and the entire data are too large to be cached locally, she needs to craft a series of ad hoc federated queries, adjusting filters and/or aggregations iteratively to explore different aspects of the data and gradually narrow down the possible causes.*

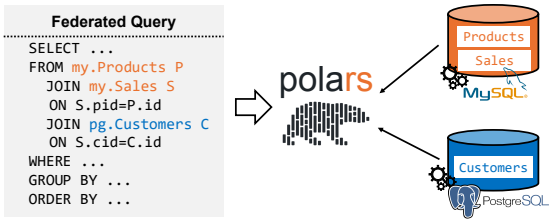


Figure 1: Example of using a data science query engine (e.g., Polars) to query data on PostgreSQL and MySQL.

Under this background, there is a growing trend for DS engines to add query federation support. Some engines, including Pandas [58], Dask [62], Modin [59], DataFusion [42], and Polars [16], offer mechanisms for accessing external data sources. However, since they lack direct support for federated queries (e.g., example in Figure 1), users must first manually fetch the data needed from each source and then join them on the DS engine. Others, such as Spark [28], DuckDB [60], and ClickHouse [8] (chDB [7]) come with native federation capability that allows users to directly issue federated queries. However, their current implementations miss optimizations for improving efficiency. Specifically, none of these systems automatically pushes down joins (e.g., the join between Products and Sales in Figure 1), and most of them fetch remote data through a single SQL endpoint only (see Section 2).

To enable efficient query federation for all DS engines, a straightforward solution would be to have each engine develop its own federation capability individually, which we refer to as the *"built-in"* approach in this paper. However, this "built-in" approach requires intrusive modifications to the core engines (e.g., query optimizer) and involves considerable engineering efforts (see Section 2). Worse, the efforts made for one engine are not portable to another, since they are tightly coupled with the engine internals (e.g., data representation, optimization framework).

In this paper, we propose Accio, a new *"bolt-on"* approach to equip a given target DS engine for efficient query federation support through a pluggable design, which can be easily adopted by various engines without touching their core modules. While Accio directly addresses the limitations of the "built-in" approach, its "bolt-on" nature also introduces new challenges.

Firstly, how to make Accio *easy to integrate* with various target DS engines? To achieve this, Accio functions as an external query rewriter, decomposing the input federated query into multiple small ones. Each of these rewritten queries is declarative and is executed by only one of the involved systems (either a data source or the local target engine), allowing direct use of the existing SQL interfaces.

Proceedings of the VLDB Endowment, Vol. 18, No. 7 ISSN 2150-8097.
doi:10.14778/3734839.3734849

In the experiment, we find that it is easy to integrate Accio with each evaluated engine (e.g., less than 50 lines of Python code).

Secondly, how to *enable high query-federation performance* for various target DS engines? We focus on two optimizations in Accio: *join pushdown* and *query partitioning*, which are either not implemented or only partially supported in popular DS engines. Both optimizations mitigate the critical data-transfer bottleneck. While the former reduces data transfer by delegating joins to data sources, the latter accelerates data fetching by parallelizing it. Furthermore, they are applicable to various DS engines and data sources via Accio's declarative interface (see Section 3.1).

However, supporting these optimizations is not easy, particularly for a "bolt-on" library like Accio. Specifically, the join pushdown problem in Accio differs from that in traditional distributed/federated systems due to its declarative interface. This interface provides no direct control over the physical execution, such as the join order in each participating system, which in turn affects the quality of the join pushdown plan. Thus, existing distributed/federated optimization techniques, such as exhaustively enumerating all join orders considering locality [34, 50], or identifying pushdown opportunities only based on the best join order [21, 66], are either overly expensive or lead to poor performance (see Section 4.2). To address this, we propose a new approach that effectively identifies an efficient strategy by iteratively evaluating the next most beneficial join (see Section 4.3). As for query partitioning, to adapt to various DS engines and data sources, Accio provides a flexible interface that supports different strategies. Its impact is also seamlessly incorporated into the join pushdown process (see Section 5).

Note that we do not claim Accio can fully replace the "built-in" approach. Since the latter has more access to the target engine and more control over execution, it can potentially apply a wider range of optimizations than Accio is capable of (see Section 3.2). However, we argue that Accio remains valuable for its simplicity and portability. Moreover, it already delivers state-of-the-art performance. We evaluate Accio with five popular DS engines on TPC-H [20] and JOB [54] benchmarks (see Section 6). The results demonstrate that Accio can substantially accelerate the target DS engine in general and remains robust across different setups. We also compare Accio-enhanced DS engines against standalone federation systems (e.g., Trino [21], Apache Wayang [29]). We find that with Accio, the target DS engine consistently outperforms these systems, all while avoiding the need for users to set up an additional service.

2 BACKGROUND

Scope of the Paper. We study the problem of *enabling efficient query federation for DS engines*. We target the *loosely-coupled* [47] federation setup. Specifically, given a target DS engine (e.g., Spark, Polars) and *read* access to heterogeneous data sources (e.g., PostgreSQL, MySQL), all of which provide a *SQL interface*, our goal is to *enable* and *accelerate* the execution of federated queries that are issued to the target engine. Each base table referenced in the query resides in either a data source or the target engine. We focus on *analytical* workloads and *RDBMS* data sources since they are more challenging and have much room for improvement.

Query Federation Support in DS Engines. We study the query federation support in five DS engines: Spark, DuckDB, ClickHouse, DataFusion and Polars, which show top performance in database-like tools for data science benchmark [10].[1]

[1]The picked engines are ranked top among those written in Java, C/C++ and Rust.

Unsurprisingly, all engines support federation to some extent, reinforcing the motivation of our work. However, the existing implementations focus on enabling query federation, while considerably less efforts are devoted to optimizing the process. Since data transfer is generally considered as a crucial bottleneck in the federation setup [40], we focus our attention on two features: *join pushdown* and *query partitioning*. While the former reduces the size of the data movement by delegating join operations to the data sources, the latter parallelizes data fetching by splitting a query into multiple small ones (e.g., divide a column into bins and add each bin as a predicate to the query [11, 12, 23]) and issuing them concurrently. Table 1 summarizes the support of them in each engine.

We can see that optimizations that are crucial for federation efficiency are generally not or only partially supported. None of the engines automatically pushes down joins to the RDBMS data sources. As for query partitioning, SparkSQL requires the user to manually specify a partition column, the range of the column, and the partition number [23]. DuckDB partitions queries in only two of its RDBMS extensions (PostgreSQL and SQLite). Others do not have the support. As we show in Section 6, enabling these missing optimizations can result in significant performance improvements.

Built-in Approach. Although each engine can independently implement its missing optimizations, we argue that the efforts required for such a "built-in" approach are not trivial.

Firstly, intrusive changes are needed. For example, join pushdown is a challenging problem that must be evaluated with the awareness of physical properties like table locality and join ordering (more details in Section 4). However, these DS engines either perform join ordering over logical plans (e.g., SparkSQL, DuckDB) or do not reorder joins at all (e.g., ClickHouse, DataFusion, Polars). Therefore, enabling cost-based join pushdown requires significant modifications to the core optimization logic of these engines.

Secondly, the engineering efforts required are substantial. These engines generally adopt the mediator-wrapper architecture [57] to enable query federation, where a wrapper is implemented for each data source. To enable the above optimizations, information such as statistics and supported partitioning scheme of each source is required, which these engines currently lack. Therefore, the wrappers need to be more complicated than the existing ones, which are already fairly complex [49]. Furthermore, a wrapper of one engine is generally not reusable by another since it is usually tightly coupled with the engine internals, such as the specific data representation.

3 ACCIO: BOLT-ON QUERY FEDERATION

To address the above issues, we propose Accio, a "bolt-on" library that can enable efficient query federation for DS engines.

3.1 Workflow

Our goal is to develop a reusable library that can be integrated with various target engines to enable and/or enhance their federation capabilities. To achieve this, Accio functions as an external query rewriter that relies solely on the existing SQL interface of the systems involved. Specifically, it decomposes a federated query into multiple small ones, each corresponding to a portion of the overall execution. These queries are then processed by the appropriate system, whether it is the target engine or one of the data sources. Next, we use an example in Figure 2 to explain how Accio works. The black and white circles denote the overall execution steps and the internal query rewrite phases within Accio, respectively.

Table 1: Optimization supported for RDBMS data sources.[2]

	Spark	DuckDB	ClickHouse	DataFusion	Polars
Join Pushdown	×	×	×	×	×
Query Partitioning	✓	✓	×	×	×

Let q be a federated query issued against the target engine S_0 (green), which also queries tables maintained by two data sources S_1 (orange) and S_2 (blue):

```
q: SELECT ... FROM S1.R, S2.S, S1.T, S0.U
   WHERE R.a = S.a AND R.b = T.b AND T.c = U.c AND S.d < 10
```

❶ **Rewrite.** The input query q is first rewritten by Accio, which ① parses q to an initial query plan, whose leaf nodes (scan operators) are annotated as the remote sites that store the data. Accio then ② performs a series of optimizations. It first applies conventional rewrite rules such as projection and filter pushdown, and then runs the new join pushdown algorithm (more details in Section 4.3) while considering query partitioning opportunities (more details in Section 5). Finally, it ③ traverses the plan tree and converts it into a *rewrite plan* that consists of multiple declarative queries:

```
q1: SELECT ... FROM R, T WHERE R.b=T.b                    -- t1(S1)
q2*: SELECT ... FROM S WHERE d < 10 AND ...               -- t2(S2)
                        |- q2^0: ID < 1,000,000
                        |- q2^1: ID ∈ [1,000,000, 2,000,000)
                        |- q2^2: ID ≥ 2,000,000
q': SELECT ... FROM t1, t2, U WHERE t1.a=t2.a AND t1.c=U.c
```

Among these queries, q_1 and q_2^* are the *pushdown queries* for sites S_1 and S_2 respectively, and q' is the *local query* for the target engine.

❷ **Registration.** To use the *rewrite plan* generated by Accio and derive the final result, *pushdown queries* q_1 and q_2^* are sent to the corresponding data sources (S_1, S_2) with their results registered in the target engine (t_1/t_2 for q_1/q_2^*). These result tables do not have to be materialized now and can be created as views.

❸ **Execution.** Finally, the local query q' is executed by the target engine, using both the local table (U) and the results of the pushdown queries (t_1, t_2) as input. Data in t_1 and t_2 can be obtained from each data source through native table functions or extracted using third-party tools into a common data format (e.g., Apache Arrow [3]) and then ingested into the target engine.

In summary, Accio takes as input a federated query and outputs a *rewrite plan* consisting of multiple *pushdown queries* and a *local query*. Each *pushdown query* is executed by one of the data sources using only its local tables, with the results fetched into the target engine, where they are then used as inputs for the *local query*. Thus, it only requires read access to each data source. Additionally, this declarative workflow allows Accio to enable *join pushdown* and *query partitioning* outside the target engine. Using the above example, the join between R and T is pushed down to S_1 through q_1, and the table S in S_2 can be fetched in parallel by issuing q_2^0, q_2^1 and q_2^2 through concurrent connections. Through this workflow, only the necessary information is retrieved from each data source with the flexibility to enable parallel data transfer, thereby alleviating the bottleneck of moving large volumes of data through network.

3.2 Tradeoff Discussion

Compared to the "built-in" solution discussed in Section 2, the "bolt-on" design comes with a tradeoff.

Advantages. 1) *Simple and non-intrusive.* It greatly reduces complexity to enable efficient query federation. Instead of modifying the core logic of the query engine, the target engine only needs to invoke Accio and run the rewritten queries correspondingly.

[2]Until versions used in Section 6. Only consider automatic pushdown and partitioning.

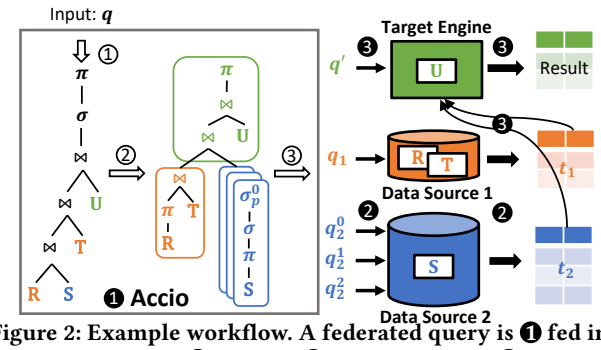


Figure 2: Example workflow. A federated query is ❶ fed into Accio, where it is ① parsed, ② optimized and ③ rewritten into a set of pushdown queries (q_1, q_2^*) and a local query (q'). Pushdown queries are ❷ issued to data sources, whose results serve as input to the local query ❸ executed by the DS engine.

2) *Reusable.* The efforts become shareable among different query engines through a system-agnostic interface (i.e., declarative SQL). This helps avoid reinventing the wheel and makes it easy to benchmark and evaluate different systems.

Limitations. 1) *Limited plan quality.* As we will illustrate in Section 4, even the exhaustive approach cannot guarantee the optimality of the rewrite plan due to the lack of information and control of the final execution. 2) *Limited runtime control.* Unlike federation systems with execution engines (e.g., Presto), Accio is only a rewriter and cannot participate in the final query execution, making it hard to support adaptive optimizations that dynamically adjust the execution based on intermediate results [22] or realtime resource availability [75] (more discussions in Section 8).

Despite the above limitations, we argue that Accio is still valuable due to its simplicity and the savings of engineering efforts. In addition, as we will show in Section 6, Accio can already be used to facilitate state-of-the-art federation performance, even with the imperfections in optimization and execution described above.

4 JOIN PUSHDOWN

Given a federated query, join pushdown decides where to perform each join.[3] For example, in Figure 3, the input query joins four relations from two remote sites (denoted by different colors), and P_1 and P_2 represent two valid solutions that push $R \bowtie S \bowtie T$ and $R \bowtie S$ to remote data sources, respectively. The two remaining feasible decisions are to only push down $R \bowtie T$ and not push down any join. $R \bowtie U$ can only be executed locally on the target engine.

In this paper, we only consider pushing down inner joins. Non-inner joins as well as projections and filters are pushed into the input relations as much as possible beforehand.

4.1 Problem Definition

Let $\mathbf{R} = \{R_0, R_1, R_2, ...\}$ be a set of relations, each residing at a site in $\mathbf{S} = \{S_0, S_1, S_2, ...\}$. A function $s : \mathbf{R} \rightarrow \mathbf{S}$ maps a relation $R \in \mathbf{R}$ to its location $S \in \mathbf{S}$. Without loss of generality, we let S_0 indicate the local target engine, and $S_1, S_2, ...$ represent remote data sources. A federated query can be seen as a graph $G = (\mathbf{V}, \mathbf{E})$, where $\mathbf{V}$ is the set of joined relations, and edge $(R_i, R_j) \in \mathbf{E}$ if there is a join condition between R_i and R_j. We have the following definitions:

Definition 1 (Pushdown Query). *A valid pushdown query is a connected subgraph $G' = (\mathbf{V}', \mathbf{E}')$ of G if there exists a remote site $S_k (k \neq 0)$ that supports join operation, s.t. $\mathbf{E}' = \{(R_i, R_j) | (R_i, R_j) \in \mathbf{E} \wedge R_i \in \mathbf{V}' \wedge R_j \in \mathbf{V}'\} \wedge \forall R \in \mathbf{V}' : s(R) = S_k$.*

[3]Not to be confused with "join pushdown" in centralized systems, where an operation is performed before others on the same system.

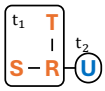
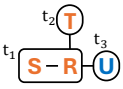
Plan	Graph Partition	Rewrite Plan
P_1		SELECT ... FROM t_1, t_2 WHERE t_1.c=t_2.c $t_1(S_1)$: SELECT ... FROM R, S, T WHERE R.a=S.a AND R.b=T.b $t_2(S_2)$: SELECT ... FROM U
P_2		SELECT ... FROM t_1, t_2, t_3 WHERE t_1.b= t_2.b AND t_1.c=t_3.c $t_1(S_1)$: SELECT ... FROM R, S WHERE R.a=S.a $t_2(S_1)$: SELECT ... FROM T $t_3(S_2)$: SELECT ... FROM U

Figure 3: Two valid rewrites for federated query: `SELECT ... FROM S1.R,S1.S,S1.T,S2.U WHERE R.a=S.a AND R.b=T.b AND R.c=U.c.`

In Figure 3, each subgraph represents a pushdown query. Since the local query always joins the results of all pushdown queries along with local relations (if any), we can define a rewrite plan as:

Definition 2 (Rewrite Plan). *A valid rewrite plan P is defined by a set of valid and non-overlapping pushdown queries $\{G_1, G_2, ...\}$ that cover all the remote relations. That is, $\cup_{G_i \in P} \mathbf{V_i} = \{R | R \in \mathbf{V} \wedge s(R) \neq S_0\} \wedge \forall G_i, G_j \in P : \mathbf{V_i} \cap \mathbf{V_j} = \emptyset$.*

Figure 3 demonstrates two examples of valid rewrite plans. The above definitions (i.e., pushdown query, rewrite plan) also correspond to the output of Accio elaborated in Section 3.1. Finally, we define the join pushdown problem for the "bolt-on" setup:

Definition 3 (Bolt-on Join Pushdown). *Given a federated query, the bolt-on join pushdown problem aims to find the valid rewrite plan with the minimum estimated cost.*

Challenges. Bolt-on join pushdown has the following challenges:

1). Large Search Space. Unlike projection and filter, it is not always favorable to push down joins. This is because DS engines usually adopt a columnar-vectorized design, which is optimized for analytical workload and therefore can be much faster than legacy data sources (e.g., PostgreSQL) on small/medium-sized data. To find the best plan, one needs to traverse a large search space, which can be up to the Bell number $B(n)$ [67] (see [69] for more details).

2). No Physical Control. The rewrite plan cannot control the physical execution of the query, such as join ordering or join algorithm selection. For example, the local engine, rather than Accio, decides how to physically join t_1, t_2 and t_3 for P_2 from Figure 3. Worse, these physical properties affect the actual cost of a rewrite plan and vice versa. Taking join order for example, P_1 in Figure 3 might be better than P_2 with order $(t_1 \bowtie t_2) \bowtie t_3$, but inferior to $(t_1 \bowtie t_3) \bowtie t_2$. And order $((R \bowtie S) \bowtie U) \bowtie T$ is feasible for P_2, but not for P_1.

The second challenge distinguishes the join pushdown problem in our "bolt-on" setup from existing distributed/federated DBMSs. To the best of our knowledge, we are the first to study this problem.

4.2 Adapting from Distributed/Federated DBMS

Although different, one can always adopt existing optimizations for distributed/federated DBMSs and convert the result physical plan into SQL queries for "bolt-on" federation. We discuss two well-known approaches here and identify their issues.

Exhaustive Approach. Previous work [34, 50] introduced the concept of *interesting site*, which is a physical property indicating the locality that performs the operation. The new property can be applied to exhaustive join enumeration algorithms (e.g., dynamic programming) to generate a distributed plan. A concrete example can be found in our technical report [69].

The primary drawback of the exhaustive approach lies in the computational overhead. Specifically, the time complexity is $O(s^3 * 3^n)$, where n and s denote the number of join factors and the number of sites involved [50]. As demonstrated in Section 6.2, traversing

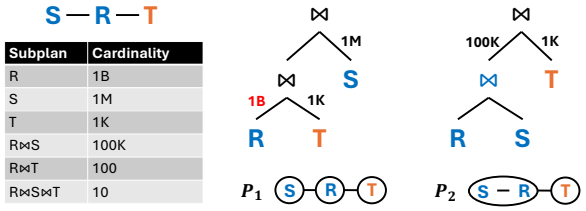


Figure 4: Example of the best plan found by two-phase approach (P_1) v.s. iterative approach (P_2). Number in the plan tree denotes the size of data transfer. Colors denote sites.

the entire search space can take seconds. Moreover, unlike in distributed DBMS, the exhaustive approach in the "bolt-on" setup cannot guarantee to find the optimal plan even if the statistics are correct, due to the lack of control over the actual physical execution.

Two-phase Approach. The *two-phase hypothesis* [44] was adopted for the query optimization of distributed/federated DBMSs [21, 36, 66], which first generates the best join order assuming a centralized setup and then finds the best decomposition of the plan by scheduling each fragment to a specific site.

The issue of this approach is that it overlooks the interaction between join pushdown and ordering. By determining the join order in the first phase without considering data movement costs and pushdown possibilities, it may result in expensive plans that stall, waiting for data transfer to complete. Figure 4 shows an example, where a fact table R is joined with a co-located dimension table S and T from another site. P_1 has a reasonable join order as it greatly reduces the size of the intermediate data. However, this order makes join pushdown impossible, resulting in a suboptimal plan that has to move the large fact table through the network. In this case, P_2 is preferable, despite having a suboptimal global join order.

4.3 Iterative Approach

The primary issue with the above solutions is that they tend to focus more on join ordering than on pushdown, which is inefficient for our setup because the order cannot be fully controlled in the "bolt-on" fashion. Thus, we propose an alternative approach, which iteratively pushes down the most beneficial join, considering join order only for cost estimation using a heuristic algorithm.

Algorithm 1: `Iterative Framework`

Input: $R = \{R_1, ..., R_n\}$
$\hat{R} = R$; $P^* =$ `joinOrdering`(S_0, R);
foreach $R_i \neq R_j \in R : s(R_i) = s(R_j) \neq S_0$ **do**
  if ¬`connected`(R_i, R_j) **then**
    continue;
  $B[\{R_i, R_j\}] =$ `benefit`$(R_i \bowtie R_j)$;
while $|B| > 0$ **do**
  $\{R_i, R_j\} = \text{argmax}\, B[\{R_i, R_j\}]$;
  $R' =$ `joinOrdering`$(s(R_i), \{r | r \in R \wedge (r \in R_i \vee r \in R_j)\})$;
  $B = B \setminus \{\{R_a, R_b\} | R_a \in \{R_i, R_j\} \vee R_b \in \{R_i, R_j\}\}$;
  foreach $R_k \in \hat{R} \setminus \{R_i, R_j\} : s(R_k) = s(R')$ **do**
    if ¬`connected`(R_k, R') **then**
      continue;
    $B[\{R_k, R'\}] =$ `benefit`$(R_k \bowtie R')$;
  $\hat{R} = \hat{R} \setminus \{R_i, R_j\} \cup \{R'\}$;
  if `benefit`$(R_i \bowtie R_j) > 1$ **then**
    $P =$ `joinOrdering`$(S_0, \hat{R})$;
    if $C(P) < C(P^*)$ **then**
      $P^* = P$;
return P^*;

The framework is illustrated by Algorithm 1, where joinOrdering(S, R) refers to a join ordering algorithm that simulates the process of ordering factors within R on site S by first fetching all factors to S if they are not already there. The function benefit is a criterion used to prioritize joins for pushdown, which will be

discussed in detail later. First, we generate a candidate plan, which fetches all base relations to local for integration, as the initial best plan P^* (line 1). Then, pairwise factors are validated for pushdown eligibility, and each valid pair is added to the pushdown candidate set B with a benefit score (lines 2–5). Next, we iteratively generate one candidate plan at a time in a greedy fashion. Specifically, at each iteration, the pair $\{R_i, R_j\}$ with the highest benefit score is selected (line 7), and a factor R' is generated by pushing down all the joins among the base factors within R_i and R_j to the corresponding remote site (line 8). Pushdown candidates are then updated by removing those that overlap with R_i or R_j (line 9) and adding new ones formed by joining the remaining factors with R' if applicable (lines 10–13). The remaining factor set $\hat{R}$ is also updated by replacing R_i and R_j with R' (line 14). If the benefit of the selected join is greater than 1 (line 15), indicating a potential performance improvement, a new candidate plan is generated by joining the remaining factors locally (line 16). The best plan is updated if this new candidate has a smaller estimated cost (lines 17-18). The procedure continues until no more joins can be pushed down (line 6).

It is clear that using Algorithm 1, both P_1 and P_2 from Figure 4 would be generated (at line 1 and in the first iteration, respectively), and the one with a lower estimated cost will be chosen. Algorithm 1 incrementally pushes down one join at each iteration and invokes the join ordering algorithm for both the new pushdown query and the global query. Thus, let $O(X)$ be the complexity of the join ordering algorithm, the complexity of the iterative framework is $O(X * n)$, where n denotes the number of join factors.

Cost Model. We adopt the calibration-based [49] approach to extend the C_{out} [56] cost function:

$$C(R) = \begin{cases} 0, & R \text{ is a base relation;} \\ |R| \cdot \gamma_S + C(R_1) + C(R_2), & R = R_1 \bowtie_S R_2, \end{cases}$$

where γ_S represents the relative cost of executing the same join at site S over the local engine (i.e., $\gamma_{S_0} = 1$). The advantage of C_{out} is that it only depends on the cardinality without the need of physical information. Similarly, we use a parameter τ_S to model the overhead of fetching intermediate result R from site S to local:

$$C(fetch(R)) = |R| \cdot \tau_S + C(R).$$

Thus, our cost model C simply abstracts the relative join execution performance and the data transfer overhead of each remote site S into two *non-negative* coefficients: γ_S and τ_S respectively.

Join Pushdown Criterion. Using the annotations of the above cost functions, we can define the join pushdown criterion as follows:

Definition 4 (Pushdown Benefit). *Let $C_{in} = C(R_1) + C(R_2)$, we derive the benefit of pushing down a specific join $R_1 \bowtie R_2$ to S_k ($s(R_1) = s(R_2) = S_k \neq S_0$) as:*

$$benefit(R_1 \bowtie R_2) = \frac{C(fetch(R_1) \bowtie_{S_0} fetch(R_2)) - C_{in}}{C(fetch(R_1 \bowtie_{S_k} R_2)) - C_{in}}. \quad (1)$$

Intuition & Analysis. Obviously, the iterative approach is heuristic and can lead to suboptimal results. In [69], we formally prove that it can find the best rewrite plan for some star queries with certain restrictions. Here, we discuss some general intuitions.

From a high level, by deducting the total cost C_{in} associated with the inputs of a join, the pushdown benefit isolates the impact from pushing down this single join, considering its input/output cardinality, data movement overhead and the performance variance between local and remote systems. Specifically, Equation (1) can be expanded using C as:

$$benefit(R_1 \bowtie R_2) = \frac{|R_1| + |R_2|}{|R_1 \bowtie R_2|} \cdot \frac{\tau_{S_k}}{\tau_{S_k} + \gamma_{S_k}} + \frac{1}{\tau_{S_k} + \gamma_{S_k}}.$$

Intuitively, for a specific remote site, a join becomes more favorable for pushdown when its input is large and the output is small. While the former relates to the data transfer cost if it is not pushed, the latter corresponds to the join cost and data transfer cost if it is pushed down. The term $\frac{|R_1|+|R_2|}{|R_1 \bowtie R_2|}$ heuristically combines these two factors. The coefficients γ_{S_k} and τ_{S_k} further adjust the benefit for the remote site S_k according to its relative execution and data transfer overheads compared to others within the federation setup.

5 QUERY PARTITIONING

We show how Accio 1) accommodates various partition schemes and 2) incorporates query partitioning opportunities into the cost-based rewrite process. We assume *static* datasets here, but our approach can be easily extended to dynamic ones as long as a consistent snapshot of each data source is available, which has been widely studied in prior work [51, 72, 76].

Accio exposes the following interface for query partitioning:

```
interface QueryPartitioner {
  fn get_scheme(q) -> scheme;
  fn estimate(q, scheme) -> cost;
  fn partition(q, scheme) -> partitions;
}
```

The interface is seamlessly integrated into the rewrite process. Specifically, during the optimization phase (② in Figure 2), the function get_scheme is invoked for each pushdown query q enumerated by Algorithm 1. The output scheme structure contains information that defines how to partition the pushdown query, such as the partition column and the number of partitions, and is attached to q. If there is a valid partition scheme, the estimate method is then used to overwrite the estimated cost described in Section 4 for fetching q to local. Finally, to generate the rewrite plan (③ in Figure 2), the partition function converts the pushdown query into the corresponding format, such as a list of partitioned queries that can be executed directly through multiple connections.

Implementation & Extensibility. We describe an implementation used in our experiment for PostgreSQL data source.

A Concrete Example. From a high level, we evenly partition the CTID field, a system column exists in every PostgreSQL table that denotes the physical location of the tuple, for large pushdown queries that contain only one base table. Specifically, given a pushdown query, get_scheme first inquires its cardinality and checks the number of base tables involved. If the cardinality is smaller than 100K or there are multiple tables, we return null, indicating no partitioning. Otherwise, we derive the number of partitions n_p as $\min(|R|/100K, 32)$. Intuitively, if the table is larger, we split the query into more partitions with a maximum number of 32. The constant values here (i.e., $100K$, 32) are configurable. estimate then adjusts the cost of fetching the query with a new multiplier $1/n_p$. As for do_partition, it places a predicate on top (e.g., σ_p^* in Figure 2), which evenly filters the CTID column of the base table, and then converts the plans into declarative queries in PostgreSQL dialect.

Through implementing the QueryPartitioner interface, Accio can be extended to support a variety of partitioning strategies. For example, rather than using the CTID column, partitioning can

Table 2: Dataset table grouping based on prior work [71].

Dataset	Group 1	Group 2	Group 3
JOB	at, cn, ct, k, kt, mc, mk	cc, cct, lt, t, mi, mii, ml	an, ci, chn, c, it, pi, rt, n
TPC-H	part, partsupp	lineitem, orders	customer, supplier, nation, region

also be performed on a column from the queried table. One could also exploit the specific underlying data layout. For instance, if the base table is already partitioned in the data source, one can directly access distinct partitions in each output query. Additionally, partitioning can be applied not only to single-table queries, but also to queries involving join operations, such as by splitting the largest table among all join tables or partitioning each table based on their corresponding join keys. Due to space constraints, more detailed examples are deferred to the technical report [69].

Tradeoff Discussion. There is a tradeoff between the generality and effectiveness of partitioning strategies. This is because the effectiveness of a partitioning strategy is influenced by two critical factors, both of which depend on the specific characteristics and layout of the underlying queried data. 1) *Partition evenness.* Uneven partitioning can result in stragglers that transfer disproportionately large amounts of data, nullifying the benefits of parallelism. The ability of a partitioning scheme to achieve even splits, however, depends on the specific layout/distribution of the queried data, which varies across different datasets and queries. 2) *Introduced overhead.* Partitioning may introduce additional overhead at the data source since the queries issued to generate partitions may share the same costly subplan (e.g., full-table scan). Whether this overhead is small enough to justify the partitioning depends on the physical storage layout of the data. For instance, certain layouts (e.g., partitioned or indexed) may naturally align with a given partitioning scheme, minimizing overhead, while others may incur significant costs.

Therefore, as a "bolt-on" library that may be adopted in various federation setup, Accio focuses on providing the extensible mechanism rather than dictating the specific strategy used. We leave it for future work to explore more partitioning strategies and develop methods to select the best one for each specific scenario.

6 EVALUATION

6.1 Experimental Setup

Source code, workloads, and configurations used in our experiments are publicly available at https://github.com/sfu-db/accio.

DS Engines. We evaluate Accio on five DS engines: Spark [28] (v3.5.1), DuckDB [60] (v0.10.3), DataFusion [42] (v39.0.0), Polars [16] (v1.3.0) and ClickHouse [8]. We adopt the embedded version of ClickHouse, namely chDB [7] (v1.3.0), as it provides a more friendly dataframe API in Python. Since enabling join pushdown requires a data fetching mechanism capable of issuing arbitrary queries to RDBMS data sources, which chDB, DataFusion, and Polars currently lack, we use ConnectorX [70] to fetch the results of the pushdown queries in the form of Arrow [3], and then convert them to their internal data format. We slightly modify DuckDB's PostgreSQL scanner so that it can leverage multiple threads for arbitrary query partitioning. These modifications are simple and non-intrusive, requiring less than 100 lines of code in most cases.

Hardware and Platform. We use three servers with two 20-core Intel Xeon Gold 6242R CPUs clocked at 3.10GHz, each has 80 hyperthreads and 375GB main memory. We deploy the target DS engine on one server and two PostgreSQL (v16.2) instances as data sources on two other servers, respectively, and test under 10Gbps network and 1Gbps network bandwidth controlled by netem [43]. Unless otherwise specified, we configure all tested engines to use 32 cores and up to 128GB of memory, and the two PostgreSQL instances with a maximum of 16 parallel workers and 128GB of shared buffers.

Table 3: Comparison of join pushdown approaches on Spark (10Gbps, JOB, unit: seconds).

	NoPush	PushAll	Exhaustive	TwoPhase	Iterative
Max	**31.32**	210.99	80.80	207.81	73.65
99th	27.95	70.14	39.56	26.67	**25.42**
90th	21.61	12.35	9.985	19.91	**6.003**
50th	3.225	1.595	1.745	2.865	**1.570**
Avg	7.782	6.095	4.152	7.443	**3.492**

Datasets and Workloads. We use the Join Order Benchmark (JOB) [54], and TPC-H [20] benchmark with various scale factors (1,10,50) for evaluation. We use all 113 queries from JOB and report the result of ten queries, ranging from joining two to eight relations, for TPC-H. We follow previous work [71] to divide the tables of each dataset into three groups as in Table 2. Each group contains 6%, 31% and 63% (13%, 85% and 2%) of the total size of JOB (TPC-H) dataset respectively. For each dataset, we construct three different table distributions (indicated by A, B, C), each of which places a different group on one data source and the rest on the other. Note that we place all tables on remote data sources to make the queries more challenging (i.e., a larger exploration space for join pushdown and query partitioning), even though Accio supports queries with base tables residing in the local target engine as well.

Implementation. Accio is implemented in Java leveraging Calcite [30]. It exposes interfaces in Python, Java, Rust and C/C++ for integration, and provides wrappers that can execute the rewrite plans on the five engines above directly. The implementation of the wrapper is simple and straightforward (less than 50 lines of code each). Unless otherwise specified, we issue Explain statements to get estimated cardinalates from each PostgreSQL instance for data that reside at the same source and use simple and widely adopted heuristics [54] for cross-site joins. We manually tune the hyperparameters in the cost model using grid search and derive a set of default values for evaluation. Automatic tuning of these parameters is beyond the scope of this paper. Each query is repeated five times after warm-up and the averaged result is reported.

6.2 Efficacy of Accio Optimizations

Join Pushdown. We evaluate three cost-based methods described in Section 4, i.e., Exhaustive, TwoPhase and Iterative, along with two rule-based baselines: NoPush and PushAll. While the former fetches all the base tables (with projection and filter pushdown), the latter pushes all non-cross-product joins as long as they can be executed at one of the remote sites. We adopt the widely used GOO [39] algorithm as the heuristic join ordering method for both TwoPhase and Iterative. Exhaustive and TwoPhase utilize the same cost model with Iterative as discussed in Section 4.3. To mitigate the possible impact of the adopted cardinality estimation strategy, in this experiment, a single PostgreSQL instance with all the data ingested is inquired for cardinality estimates. Query partitioning is disabled in this comparison.

Table 3 shows the max, 99th and 90th percentile, median and average latency of all queries from the three table distributions of JOB with a 10Gbps bandwidth on Spark. Iterative shows the best performance, which outperforms NoPush, PushAll and TwoPhase by 2.2×, 1.7× and 2.1×, respectively. An interesting finding is that Exhaustive is approximately 20% slower than Iterative on average. We think it is due to two reasons. First, since Exhaustive explores the entire search space, it is more sensitive to inaccuracies in cost

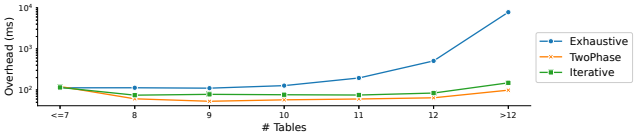


Figure 5: Maximum rewrite overhead of cost-based join pushdown approaches group by # join tables on JOB.

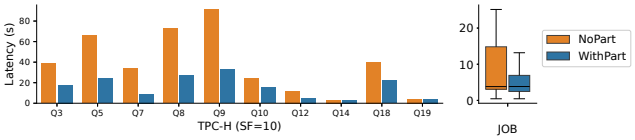


Figure 6: Impact of query partitioning on Spark (10Gbps).

estimation, such as errors in cardinality estimates, cost model, and discrepancies between the join order assumed by Accio and the final physical plan executed. Second, as we will demonstrate next, the overhead of Exhaustive becomes increasingly significant as the number of tables rises beyond a certain threshold. The remaining results can be found in [69]. In summary, Iterative consistently shows the best overall performance on all engines, whose average latency is around 1.1×, 1.1×, 1.4× and 1.3× faster than the second best approach on DuckDB, chDB, DataFusion and Polars, respectively.

Figure 5 shows the rewrite overhead of the three cost-based join pushdown methods. Exhaustive scales exponentially with more tables joined in the query (up to ~8 seconds). Both TwoPhase and Iterative keeps the overhead within around 100ms, which is negligible considering the data transfer overhead and the execution time of analytical queries under the federation setup.

Query partitioning. We disable join pushdown and show the efficacy of query partitioning alone (i.e., fetching all base tables with projection and filter pushdown to local for join). Since no join is pushed, different table distributions illustrate similar results. Thus, we only show the result of distribution A for each workload. In order to leverage Spark's native query partitioning mechanism [23], we partition the queries on its first projected numerical column and set the number of partitions to four. Figure 6 shows the latency of the two workloads on Spark with 10Gbps bandwidth. We observe that query partitioning effectively improves the performance. In particular, by enabling query partitioning, the performance of a single query can be accelerated by up to 3.7× on TPC-H, and the accumulated latency of JOB is around 60% faster. The results on other four engines exhibit similar patterns, as detailed in [69].

6.3 Performance Comparison

Enable Query Federation. We demonstrate that Accio enables efficient query federation for DataFusion and Polars. Since neither of these engines provides native federation support, we compare the performance of the Accio-enabled solution against a Naive baseline. The baseline fetches all necessary tables (with projection and filter pushdown) without partitioning and joins them locally.

The three bars A-C in Figure 7a and Figure 7c denote the speedup on the three table distributions for TPC-H queries. A "×" is placed on Q5 in Figure 7a due to an OOM error raised by DataFusion, and on Q7 and Q19 in Figure 7c since Polars currently does not support disjunctive join conditions. For the remaining TPC-H queries, Accio consistently outperforms the Naive baseline for both engines, achieving up to 14× and 17× speedup under 1Gbps and 10Gbps bandwidth conditions, respectively. Figure 7b and Figure 7d show the time distribution of all JOB queries from the three table distributions. The average improvement is 3.7× and 5.3× (4.4× and 6.1×) on DataFusion (Polars) under 1Gbps and 10Gbps, respectively.

Improve Query Federation. We take Spark, DuckDB and chDB, and compare their performance without (i.e., using their existing implementations) and with Accio. The result for chDB can be found in [69]. Figure 8a shows the performance speed up after integrating Accio to Spark on TPC-H. It is clear that Accio brings significant improvement under both network conditions. In fact, it is beneficial on almost all queries of TPC-H with up to 16× speedup. For JOB (Figure 8b), we observe that Accio steadily improves the overall performance by more than 4× under both network conditions.

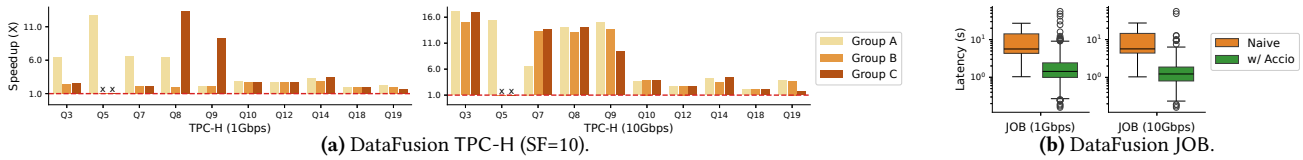
Figure 8c and Figure 8d present the results on DuckDB. We observe that Accio generally provides better performance under 1Gbps bandwidth but may be slower under 10Gbps. This is because DuckDB's native PostgreSQL scanner is highly efficient with query partitioning already enabled. Furthermore, DuckDB outperforms PostgreSQL when data fit in memory. As a result, with high bandwidth, fetching base tables and performing joins on DuckDB tends to be more beneficial, while Accio may delegate some joins to remote sites, leading to a less optimal plan. Despite this, Accio is still capable of finishing every query in a reasonable time frame (≤20 seconds) and being faster for some queries. When bandwidth is limited, the pushdown decisions made by Accio become helpful, as the primary bottleneck shifts back to the data transfer. In such cases, Accio helps DuckDB exhibit less degradation.

Compare with Standalone FDBMS. We compare the performance between Trino [21] (release 453), a fork from Presto [64], and Spark enhanced by Accio since both systems are scalable and built for big data analysis. Trino is deployed on the same server and is configured to use at least the same amount of CPU and memory resources as Spark. We report the result of Trino, Spark and Spark + Accio. The first three figures of Figure 9 present the result of running TPC-H with scale factor varying from 1 to 50 over table distribution A (see [69] for B and C) given 10Gbps bandwidth. We can see that although there is no clear winner between Trino and native Spark, Accio can help Spark beat Trino significantly and consistently by up to 8×. The last figure shows the result on JOB, where native Spark is about twice as slow as Trino to complete all queries, which, in turn, is more than twice as slow as Spark + Accio. Due to space limitation, we leave the comparison with Apache Wayang [26, 29, 52] to our technical report [69].

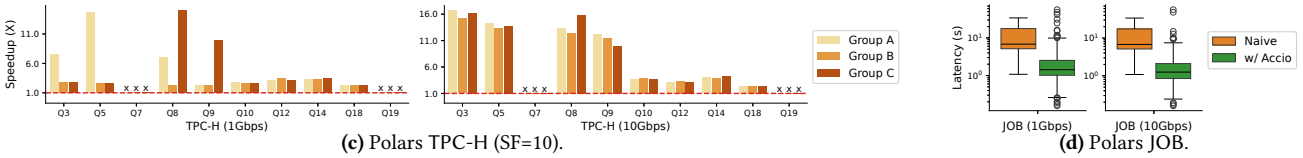
7 RELATED WORK

Standalone Federation Systems. Federated DBMSs [65] aim to coordinate a collection of autonomous DBMSs. Garlic [34, 63] integrates different data sources using a mediator-wrapper architecture, which is widely adopted later. More recently, ploygot and cross-platform data systems [24, 26–28, 37, 38, 41, 45, 47, 48, 52, 64, 68] further renewed interest in query federation. In particular, Presto [64] and its derivative products, including Trino [21], AWS Athena [2] and Huawei Hetu [14], have become popular for supporting querying multiple data sources with a unified SQL interface. Unlike these systems that offer federation functionality directly through their own engines, Accio is designed as a library that enables and enhances federation capabilities of various given query engines.

In-situ Query Federation. Tableau [71] uses a machine learning model to dynamically choose the best federation engine for each query using the cost estimation from each instance. XDB [40] delegates the entire query to existing DBMSs leveraging similar query rewrite mechanism as Accio. These previous works address scenarios where no fixed federation engine is designated. That is, they study the problem of selecting the best engine(s) for federation across all systems involved, assuming pre-established interconnections. In contrast, Accio aims to enable efficient query federation for a specific target engine with fewer constraints.

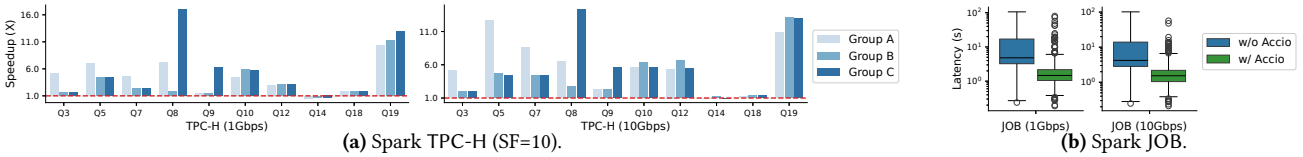


(a) DataFusion TPC-H (SF=10). **(b)** DataFusion JOB.

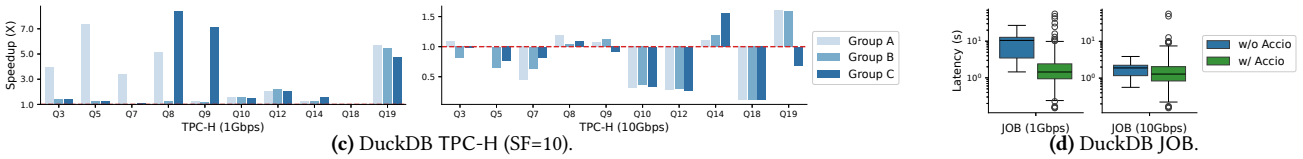


(c) Polars TPC-H (SF=10). **(d)** Polars JOB.

Figure 7: Accio on enabling query federation (red line: y=1; ×: cannot be executed by the Naive baseline).



(a) Spark TPC-H (SF=10). **(b)** Spark JOB.



(c) DuckDB TPC-H (SF=10). **(d)** DuckDB JOB.

Figure 8: Accio on improving query federation (red line: y=1).

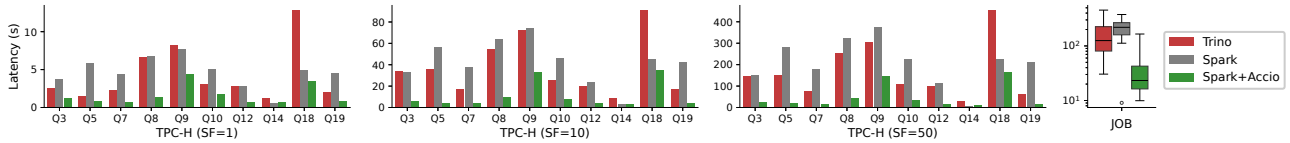


Figure 9: Trino v.s. Spark w/wo Accio (10Gbps).

Virtual Data Integration Systems. Accio also shares similarities with virtual data integration systems [55], particularly in the overall query processing workflow. Like Accio, mediator-wrapper-based virtual data integration systems [25, 31–33, 46, 53, 61] also decompose an input query into multiple subqueries that are sent to different data sources, with the results retrieved and integrated locally. However, unlike Accio, these systems are essentially standalone platforms with dedicated engines as part of the mediator for integration execution. They are designed for general data integration scenarios, emphasizing handling data heterogeneity using semantic technologies such as RDF and SPARQL. In contrast, Accio is a "bolt-on" library that aims to enable efficient federation capabilities for various relational query engines. Thus, we focus on techniques (e.g., join pushdown, query partitioning) that are applicable to general relational engines via the SQL interface.

8 DISCUSSION AND FUTURE WORK

Rule-based Rewrite. The optimization phase (② in Figure 2) of Accio applies a series of transformation rules utilizing the extensible Calcite framework [30]. These rules, fired in multiple stages, include both provided ones (e.g., unnesting correlated subqueries [6], predicate pushdown [5]) and newly implemented ones (e.g., join pushdown, query partitioning). Likewise, Accio can be easily extended to incorporate additional transformations in a *rule-based* manner, such as constant folding [15, 18], common subexpression elimination [9, 19] and group-by pushdown [35, 73, 74].

Cost-based Group-by Pushdown. Enabling group-by pushdown within the Iterative Framework of Accio in a cost-based manner is not trivial, as exhaustively applying all possible transformations considerably enlarges the search space [35]. Inspired by previous work [35], one possible solution is to push down a group-by operation only if it results in a cheaper *pushdown query* locally. This approach is lightweight and also provides guarantees regarding the quality of the resulting plan. More detailed discussions can be found in the technical report [69]. We leave the exploration of more sophisticated designs for future work.

Adaptive Pushdown. Joins pushed down by Accio can be counterproductive when the hyperparameters (i.e., γ, τ from Section 4.3) cannot reflect the realtime resource availability of the remote sites. Adjusting these parameters through monitoring could mitigate this issue but not resolve it, as resource utilization can fluctuate between planning and runtime. [75] proposes to adaptively push back requests when the storage layer is under heavy load. It requires modifications to both data sources and the target engine, which lies beyond the scope of a "bolt-on" library like Accio. However, Accio can be used to reduce the efforts needed to support it. For example, Accio can be extended to generate a fallback plan that avoids pushing down joins to data sources identified as being under heavy load at runtime, eliminating the need for re-optimization within the target engine. We leave this as an interesting direction for future work and defer more detailed discussions to [69].

9 CONCLUSION

We studied the problem of "bolt-on" query federation. We identified the issues of supporting federated queries in existing solutions and proposed to decouple this feature into Accio, a middleware library designed to enable efficient query federation for various DS engines. We integrated Accio with five DS engines and conducted extensive experiments, showing that Accio can easily "bolt-on" to these systems while delivering high and robust performance.

REFERENCES

[1] 2023. Federated Query using Spark. https://medium.com/globant/federated-query-using-spark-a32ad9152e77.

[2] 2024. Amazon Athena. https://docs.aws.amazon.com/athena/latest/ug/what-is.html.

[3] 2024. Apache Arrow. https://arrow.apache.org/.

[4] 2024. Are different database systems going to be supported data sources? https://github.com/apache/datafusion/issues/1048.

[5] 2024. Calcite CoreRules. https://calcite.apache.org/javadocAggregate/org/apache/calcite/rel/rules/CoreRules.html.

[6] 2024. Calcite RelDecorrelator. https://calcite.apache.org/javadocAggregate/org/apache/calcite/sql2rel/RelDecorrelator.html.

[7] 2024. chDB. https://clickhouse.com/docs/en/chdb.

[8] 2024. ClickHouse. https://github.com/ClickHouse/ClickHouse.

[9] 2024. Common Subexpression Elimination Explained. https://learn.microsoft.com/en-us/sql/analytics-platform-system/common-sub-expression-elimination?view=aps-pdw-2016-au7.

[10] 2024. Database-like ops benchmark. https://duckdblabs.github.io/db-benchmark/.

[11] 2024. DuckDB Postgres extension. https://github.com/duckdb/postgres_scanner.

[12] 2024. DuckDB SQLite extension. https://github.com/duckdb/sqlite_scanner.

[13] 2024. Federated Querying using DuckDB on blockchain data. https://kowalskidefi.medium.com/federated-querying-using-duckdb-x-postgres-on-blockchain-data-5391518601ee.

[14] 2024. Huawei Hetu Cyberverse. https://www.huaweicloud.com/product/live/dspace.html.

[15] 2024. MySQL Constant-Folding Optimization. https://dev.mysql.com/doc/refman/8.4/en/constant-folding-optimization.html.

[16] 2024. Polars: Fast multi-threaded DataFrame library in Rust and Python. https://github.com/pola-rs/polars.

[17] 2024. scan_database feature. https://github.com/pola-rs/polars/issues/9091.

[18] 2024. SQLServer Constant Folding and Expression Evaluation. https://learn.microsoft.com/en-us/sql/relational-databases/query-processing-architecture-guide?view=sql-server-ver16#constant-folding-and-expression-evaluation.

[19] 2024. Subexpression Elimination In Code-Generated Expression Evaluation (Common Expression Reuse). https://books.japila.pl/spark-sql-internals/subexpression-elimination/.

[20] 2024. TPC-H Homepage. http://www.tpc.org/tpch.

[21] 2024. Trino, a query engine that runs at ludicrous speed. https://trino.io/.

[22] 2024. Trino Dynamic Filtering. https://trino.io/docs/current/admin/dynamic-filtering.html.

[23] Apache Spark 3.4.0. 2024. JDBC To Other Databases. https://spark.apache.org/docs/latest/sql-data-sources-jdbc.html.

[24] Azza Abouzeid, Kamil Bajda-Pawlikowski, Daniel J. Abadi, Alexander Rasin, and Avi Silberschatz. 2009. HadoopDB: An Architectural Hybrid of MapReduce and DBMS Technologies for Analytical Workloads. *Proc. VLDB Endow.* 2, 1 (2009), 922–933. https://doi.org/10.14778/1687627.1687731

[25] AA Adeyelu and EO Anyebe. 2018. Implementing an Improved Mediator Wrapper Paradigm for Heterogeneous Database Integration. *Nigerian Annals of Pure and Applied Sciences* 1 (2018), 224–235.

[26] Divy Agrawal, Sanjay Chawla, Bertty Contreras-Rojas, Ahmed K. Elmagarmid, Yasser Idris, Zoi Kaoudi, Sebastian Kruse, Ji Lucas, Essam Mansour, Mourad Ouzzani, Paolo Papotti, Jorge-Arnulfo Quiané-Ruiz, Nan Tang, Saravanan Thirumuruganathan, and Anis Troudi. 2018. RHEEM: Enabling Cross-Platform Data Processing - May The Big Data Be With You! -. *Proc. VLDB Endow.* 11, 11 (2018), 1414–1427. https://doi.org/10.14778/3236187.3236195

[27] Rana Alotaibi, Damian Bursztyn, Alin Deutsch, Ioana Manolescu, and Stamatis Zampetakis. 2019. Towards Scalable Hybrid Stores: Constraint-Based Rewriting to the Rescue. In *Proceedings of the 2019 International Conference on Management of Data, SIGMOD Conference 2019, Amsterdam, The Netherlands, June 30 - July 5, 2019*, Peter A. Boncz, Stefan Manegold, Anastasia Ailamaki, Amol Deshpande, and Tim Kraska (Eds.). ACM, 1660–1677. https://doi.org/10.1145/3299869.3319895

[28] Michael Armbrust, Reynold S. Xin, Cheng Lian, Yin Huai, Davies Liu, Joseph K. Bradley, Xiangrui Meng, Tomer Kaftan, Michael J. Franklin, Ali Ghodsi, and Matei Zaharia. 2015. Spark SQL: Relational Data Processing in Spark. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data, Melbourne, Victoria, Australia, May 31 - June 4, 2015*, Timos K. Sellis, Susan B. Davidson, and Zachary G. Ives (Eds.). ACM, 1383–1394. https://doi.org/10.1145/2723372.2742797

[29] Kaustubh Beedkar, Bertty Contreras-Rojas, Haralampos Gavriilidis, Zoi Kaoudi, Volker Markl, Rodrigo Pardo-Meza, and Jorge-Arnulfo Quiané-Ruiz. 2023. Apache Wayang: A Unified Data Analytics Framework. *SIGMOD Rec.* 52, 3 (2023), 30–35. https://doi.org/10.1145/3631504.3631510

[30] 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, SIGMOD Conference 2018, Houston, TX, USA, June 10-15, 2018*, Gautam Das, Christopher M. Jermaine, and Philip A. Bernstein (Eds.). ACM, 221–230. https://doi.org/10.1145/3183713.3190662

[31] Konstantina Bereta, George Papadakis, and Manolis Koubarakis. 2020. OBDA for the Web: Creating Virtual RDF Graphs On Top of Web Data Sources. *CoRR* abs/2005.11264 (2020). arXiv:2005.11264 https://arxiv.org/abs/2005.11264

[32] Konstantina Bereta, George Stamoulis, and Manolis Koubarakis. 2018. Ontology-Based Data Access and Visualization of Big Vector and Raster Data. In *2018 IEEE International Geoscience and Remote Sensing Symposium, IGARSS 2018, Valencia, Spain, July 22-27, 2018*. IEEE, 407–410. https://doi.org/10.1109/IGARSS.2018.8518255

[33] Diego Calvanese, Benjamin Cogrel, Sarah Komla-Ebri, Roman Kontchakov, Davide Lanti, Martin Rezk, Mariano Rodriguez-Muro, and Guohui Xiao. 2017. Ontop: Answering SPARQL queries over relational databases. *Semantic Web* 8, 3 (2017), 471–487. https://doi.org/10.3233/SW-160217

[34] Michael J. Carey, Laura M. Haas, Peter M. Schwarz, Manish Arya, William F. Cody, Ronald Fagin, Myron Flickner, Allen Luniewski, Wayne Niblack, Dragutin Petkovic, Joachim Thomas, John H. Williams, and Edward L. Wimmers. 1995. Towards Heterogeneous Multimedia Information Systems: The Garlic Approach. In *Proceedings RIDE-DOM '95, Fifth International Workshop on Research Issues in Data Engineering - Distributed Object Management, Taipei, Taiwan, March 6-7, 1995*, Omran A. Bukhres, M. Tamer Özsu, and Ming-Chien Shan (Eds.). IEEE Computer Society, 124–131. https://doi.org/10.1109/RIDE.1995.378736

[35] Surajit Chaudhuri and Kyuseok Shim. 1994. Including Group-By in Query Optimization. In *VLDB'94, Proceedings of 20th International Conference on Very Large Data Bases, September 12-15, 1994, Santiago de Chile, Chile*, Jorge B. Bocca, Matthias Jarke, and Carlo Zaniolo (Eds.). Morgan Kaufmann, 354–366. http://www.vldb.org/conf/1994/P354.PDF

[36] Amol Deshpande and Joseph M. Hellerstein. 2002. Decoupled Query Optimization for Federated Database Systems. In *Proceedings of the 18th International Conference on Data Engineering, San Jose, CA, USA, February 26 - March 1, 2002*, Rakesh Agrawal and Klaus R. Dittrich (Eds.). IEEE Computer Society, 716–727. https://doi.org/10.1109/ICDE.2002.994788

[37] David J. DeWitt, Alan Halverson, Rimma V. Nehme, Srinath Shankar, Josep Aguilar-Saborit, Artin Avanes, Miro Flasza, and Jim Gramling. 2013. Split query processing in polybase. In *Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2013, New York, NY, USA, June 22-27, 2013*, Kenneth A. Ross, Divesh Srivastava, and Dimitris Papadias (Eds.). ACM, 1255–1266. https://doi.org/10.1145/2463676.2463709

[38] Jennie Duggan, Aaron J. Elmore, Michael Stonebraker, Magdalena Balazinska, Bill Howe, Jeremy Kepner, Sam Madden, David Maier, Tim Mattson, and Stanley B. Zdonik. 2015. The BigDAWG Polystore System. *SIGMOD Rec.* 44, 2 (2015), 11–16. https://doi.org/10.1145/2814710.2814713

[39] Leonidas Fegaras. 1998. A New Heuristic for Optimizing Large Queries. In *Database and Expert Systems Applications, 9th International Conference, DEXA '98, Vienna, Austria, August 24-28, 1998, Proceedings (Lecture Notes in Computer Science)*, Gerald Quirchmayr, Erich Schweighofer, and Trevor J. M. Bench-Capon (Eds.), Vol. 1460. Springer, 726–735. https://doi.org/10.1007/BFB0054528

[40] Haralampos Gavriilidis, Kaustubh Beedkar, Jorge-Arnulfo Quiané-Ruiz, and Volker Markl. 2023. In-Situ Cross-Database Query Processing. In *ICDE*.

[41] Ionel Gog, Malte Schwarzkopf, Natacha Crooks, Matthew P. Grosvenor, Allen Clement, and Steven Hand. 2015. Musketeer: all for one, one for all in data processing systems. In *Proceedings of the Tenth European Conference on Computer Systems, EuroSys 2015, Bordeaux, France, April 21-24, 2015*, Laurent Réveillère, Tim Harris, and Maurice Herlihy (Eds.). ACM, 2:1–2:16. https://doi.org/10.1145/2741948.2741968

[42] Andy Grove. [n.d.]. *Apache DataFusion.* https://github.com/apache/arrow-datafusion

[43] S. Hemminger. April 2005. Network Emulation with NetEm. Open Source Development Lab.

[44] Wei Hong and Michael Stonebraker. 1993. Optimization of Parallel Query Execution Plans in XPRS. *Distributed Parallel Databases* 1, 1 (1993), 9–32. https://doi.org/10.1007/BF01277518

[45] Zoi Kaoudi and Jorge-Arnulfo Quiané-Ruiz. 2022. Unified Data Analytics: State-of-the-art and Open Problems. *Proc. VLDB Endow.* 15, 12 (2022), 3778–3781. https://www.vldb.org/pvldb/vol15/p3778-kaoudi.pdf

[46] Evgeny Kharlamov, Theofilos Mailis, Gulnar Mehdi, Christian Neuenstadt, Özgür L. Özçep, Mikhail Roshchin, Nina Solomakhina, Ahmet Soylu, Christoforos Svingos, Sebastian Brandt, Martin Giese, Yannis E. Ioannidis, Steffen Lamparter, Ralf Möller, Yannis Kotidis, and Arild Waaler. 2017. Semantic access to streaming and static data at Siemens. *J. Web Semant.* 44 (2017), 54–74. https://doi.org/10.1016/J.WEBSEM.2017.02.001

[47] Felix Kiehn, Mareike Schmidt, Daniel Glake, Fabian Panse, Wolfram Wingerath, Benjamin Wollmer, Martin Poppinga, and Norbert Ritter. 2022. Polyglot Data Management: State of the Art & Open Challenges. *Proc. VLDB Endow.* 15, 12 (2022), 3750–3753. https://www.vldb.org/pvldb/vol15/p3750-panse.pdf

[48] Boyan Kolev, Patrick Valduriez, Carlyna Bondiombouy, Ricardo Jiménez-Peris, Raquel Pau, and José Pereira. 2016. CloudMdsQL: querying heterogeneous cloud data stores with a common language. *Distributed Parallel Databases* 34, 4 (2016), 463–503. https://doi.org/10.1007/s10619-015-7185-y

[49] Donald Kossmann. 2000. The State of the art in distributed query processing. *ACM Comput. Surv.* 32, 4 (2000), 422–469. https://doi.org/10.1145/371578.371598

[50] Donald Kossmann and Konrad Stocker. 2000. Iterative dynamic programming: a new class of query optimization algorithms. *ACM Trans. Database Syst.* 25, 1 (2000), 43–82. https://doi.org/10.1145/352958.352982

[51] Peter Kraft, Qian Li, Xinjing Zhou, Peter Bailis, Michael Stonebraker, Matei Zaharia, and Xiangyao Yu. 2023. Epoxy: ACID Transactions Across Diverse Data Stores. *Proceedings of the VLDB Endowment* 16, 11 (2023), 2742–2754.

[52] Sebastian Kruse, Zoi Kaoudi, Bertty Contreras-Rojas, Sanjay Chawla, Felix Naumann, and Jorge-Arnulfo Quiané-Ruiz. 2020. RHEEMix in the data jungle: a cost-based optimizer for cross-platform systems. *VLDB J.* 29, 6 (2020), 1287–1310. https://doi.org/10.1007/s00778-020-00612-x

[53] Andreas Langegger, Wolfram Wöß, and Martin Blöchl. 2008. A Semantic Web Middleware for Virtual Data Integration on the Web. In *The Semantic Web: Research and Applications, 5th European Semantic Web Conference, ESWC 2008, Tenerife, Canary Islands, Spain, June 1-5, 2008, Proceedings (Lecture Notes in Computer Science)*, Sean Bechhofer, Manfred Hauswirth, Jörg Hoffmann, and Manolis Koubarakis (Eds.), Vol. 5021. Springer, 493–507. https://doi.org/10.1007/978-3-540-68234-9_37

[54] Viktor Leis, Andrey Gubichev, Atanas Mirchev, Peter A. Boncz, Alfons Kemper, and Thomas Neumann. 2015. How Good Are Query Optimizers, Really? *Proc. VLDB Endow.* 9, 3 (2015), 204–215. https://doi.org/10.14778/2850583.2850594

[55] Maroua Masmoudi, Sana Ben Abdallah Ben Lamine, Mohamed-Hedi Karray, Bernard Archimède, and Hajer Baazaoui Zghal. 2024. Semantic Data Integration and Querying: A Survey and Challenges. *ACM Comput. Surv.* 56, 8 (2024), 209:1–209:35. https://doi.org/10.1145/3653317

[56] Thomas Neumann. 2009. Query simplification: graceful degradation for join-order optimization. In *Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2009, Providence, Rhode Island, USA, June 29 - July 2, 2009*, Ugur Çetintemel, Stanley B. Zdonik, Donald Kossmann, and Nesime Tatbul (Eds.). ACM, 403–414. https://doi.org/10.1145/1559845.1559889

[57] M. Tamer Özsu and Patrick Valduriez. 2020. *Principles of Distributed Database Systems, 4th Edition.* Springer. https://doi.org/10.1007/978-3-030-26253-2

[58] The pandas development team. 2020. *pandas-dev/pandas: Pandas.* https://doi.org/10.5281/zenodo.3509134

[59] Devin Petersohn, William W. Ma, Doris Jung Lin Lee, Stephen Macke, Doris Xin, Xiangxi Mo, Joseph Gonzalez, Joseph M. Hellerstein, Anthony D. Joseph, and Aditya G. Parameswaran. 2020. Towards Scalable Dataframe Systems. *Proc. VLDB Endow.* 13, 11 (2020), 2033–2046. http://www.vldb.org/pvldb/vol13/p2033-petersohn.pdf

[60] Mark Raasveldt and Hannes Muehleisen. [n.d.]. *DuckDB.* https://github.com/duckdb/duckdb

[61] Manuel A. Regueiro, José Ramon Rios Viqueira, Christoph Stasch, and José A. Taboada. 2017. Semantic mediation of observation datasets through Sensor Observation Services. *Future Gener. Comput. Syst.* 67 (2017), 47–56. https://doi.org/10.1016/J.FUTURE.2016.08.013

[62] Matthew Rocklin. 2015. Dask: Parallel computation with blocked algorithms and task scheduling. In *Proceedings of the 14th python in science conference.* Citeseer.

[63] Mary Tork Roth and Peter M. Schwarz. 1997. Don't Scrap It, Wrap It! A Wrapper Architecture for Legacy Data Sources. In *VLDB'97, Proceedings of 23rd International Conference on Very Large Data Bases, August 25-29, 1997, Athens, Greece*, Matthias Jarke, Michael J. Carey, Klaus R. Dittrich, Frederick H. Lochovsky, Pericles Loucopoulos, and Manfred A. Jeusfeld (Eds.). Morgan Kaufmann, 266–275. http://www.vldb.org/conf/1997/P266.PDF

[64] Raghav Sethi, Martin Traverso, Dain Sundstrom, David Phillips, Wenlei Xie, Yutian Sun, Nezih Yegitbasi, Haozhun Jin, Eric Hwang, Nileema Shingte, and Christopher Berner. 2019. Presto: SQL on Everything. In *35th IEEE International Conference on Data Engineering, ICDE 2019, Macao, China, April 8-11, 2019.* IEEE, 1802–1813. https://doi.org/10.1109/ICDE.2019.00196

[65] Amit P. Sheth and James A. Larson. 1990. Federated Database Systems for Managing Distributed, Heterogeneous, and Autonomous Databases. *ACM Comput. Surv.* 22, 3 (1990), 183–236. https://doi.org/10.1145/96602.96604

[66] Michael Stonebraker, Paul M. Aoki, Witold Litwin, Avi Pfeffer, Adam Sah, Jeff Sidell, Carl Staelin, and Andrew Yu. 1996. Mariposa: A Wide-Area Distributed Database System. *VLDB J.* 5, 1 (1996), 48–63. https://doi.org/10.1007/S007780050015

[67] Andrew Vince. 2017. Counting connected sets and connected partitions of a graph. *Australas. J Comb.* 67 (2017), 281–293. http://ajc.maths.uq.edu.au/pdf/67/ajc_v67_p281.pdf

[68] Jingjing Wang, Tobin Baker, Magdalena Balazinska, Daniel Halperin, Brandon Haynes, Bill Howe, Dylan Hutchison, Shrainik Jain, Ryan Maas, Parmita Mehta, Dominik Moritz, Brandon Myers, Jennifer Ortiz, Dan Suciu, Andrew Whitaker, and Shengliang Xu. 2017. The Myria Big Data Management and Analytics System and Cloud Services. In *8th Biennial Conference on Innovative Data Systems Research, CIDR 2017, Chaminade, CA, USA, January 8-11, 2017, Online Proceedings.* www.cidrdb.org. http://cidrdb.org/cidr2017/papers/p37-wang-cidr17.pdf

[69] Xiaoying Wang, Jiannan Wang, Tianzheng Wang, and Yong Zhang. 2024. [Technical Report] Accio: Bolt-on Query Federation. https://github.com/sfu-db/accio/blob/main/accio_technical_report.pdf.

[70] Xiaoying Wang, Weiyuan Wu, Jinze Wu, Yizhou Chen, Nick Zrymiak, Changbo Qu, Lampros Flokas, George Chow, Jiannan Wang, Tianzheng Wang, Eugene Wu, and Qingqing Zhou. 2022. ConnectorX: Accelerating Data Loading From Databases to Dataframes. *Proc. VLDB Endow.* 15, 11 (2022), 2994–3003. https://www.vldb.org/pvldb/vol15/p2994-wang.pdf

[71] 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, aiDM@SIGMOD 2019, Amsterdam, The Netherlands, July 5, 2019*, Rajesh Bordawekar and Oded Shmueli (Eds.). ACM, 2:1–2:7. https://doi.org/10.1145/3329859.3329873

[72] Hiroyuki Yamada, Toshihiro Suzuki, Yuji Ito, and Jun Nemoto. 2023. ScalarDB: Universal Transaction Manager for Polystores. *Proceedings of the VLDB Endowment* 16, 12 (2023), 3768–3780.

[73] W Yan and Per-Ake Larson. 1995. *Interchanging the order of grouping and join.* Technical Report. Citeseer.

[74] Weipeng P. Yan and Per-Åke Larson. 1994. Performing Group-By before Join. In *Proceedings of the Tenth International Conference on Data Engineering, February 14-18, 1994, Houston, Texas, USA.* IEEE Computer Society, 89–100. https://doi.org/10.1109/ICDE.1994.283001

[75] Yifei Yang, Xiangyao Yu, Marco Serafini, Ashraf Aboulnaga, and Michael Stonebraker. 2024. FlexpushdownDB: rethinking computation pushdown for cloud OLAP DBMSs. *VLDB J.* 33, 5 (2024), 1643–1670. https://doi.org/10.1007/S00778-024-00867-8

[76] Jianqiu Zhang, Kaisong Huang, Tianzheng Wang, and King Lv. 2022. Skeena: Efficient and Consistent Cross-Engine Transactions. In *SIGMOD '22: International Conference on Management of Data, Philadelphia, PA, USA, June 12 - 17, 2022*, Zachary G. Ives, Angela Bonifati, and Amr El Abbadi (Eds.). ACM, 34–48. https://doi.org/10.1145/3514221.3526171