Accio: Bolt-on Query Federation [Scalable Data Science]

Xiaoying Wang[†], Jiannan Wang[◇†], Tianzheng Wang[†], Yong Zhang[◇]
Simon Fraser University[†] Huawei Technologies[◇]
{xiaoying_wang, jnwang, tzwang}@sfu.ca yong.zhang3@huawei.com

ABSTRACT

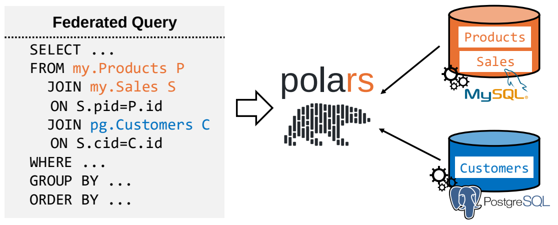
Data scientists today often need to analyze data from various places. This makes it necessary for corresponding engines to support query federation (i.e., the ability to perform SQL queries over data hosted in different sources). Although many systems come with federation capabilities, their implementations 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.

1 INTRODUCTION

In recent years, a variety of query engines have become popular among data scientists, such as Pandas [47], Spark [21], DuckDB [49], ClickHouse [6] (chDB [5]), Dask [50], Modin [48], DataFusion [33], Polars [12], 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, 10, 13]. 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.*

Under this background, there is a growing trend for DS engines to add query federation support. Some engines, including Pandas [47], Dask [50], Modin [48], DataFusion [33], and Polars [12], 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 [21],



```
SELECT ...
FROM my.Products P
  JOIN my.Sales S
   ON S.pid=P.id
  JOIN pg.Customers C
   ON S.cid=C.id
WHERE ...
GROUP BY ...
ORDER BY ...
```

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

DuckDB [49], and ClickHouse [6] (chDB [5]) come with native federation capability that allows users to directly issue federated queries. Nevertheless, we find that their current implementations miss optimizations for improving efficiency (see Section 2.1).

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 2.2). 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. 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, we point out that 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 and vice versa. We find that existing distributed/federated query optimization techniques [15, 40, 55] are either expensive or do not consider the interaction between join order and pushdown. To address this, we propose a new approach that effectively identifies an efficient strategy by iteratively evaluating the next most beneficial join, considering the join order using a heuristic algorithm (see Section 4). In terms of query partitioning, to adapt to various DS engines and data sources, Accio provides a flexible query partitioning interface that supports different partition schemes. It also seamlessly incorporates the effects into the join pushdown process (see Section 5).

It is important to 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 [14] and JOB [43] 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 [15], Apache Wayang [22]). 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. This paper makes the following contributions:

- We identify the challenges of enabling efficient query federation for DS engines in the *"built-in"* fashion and propose a new *"bolt-on"* approach that tackles the problem (Section 2, Section 3).
- We study the new join pushdown problem under the "bolt-on" setup and propose a new algorithm to tackle it (Section 4).
- We design a query partitioning mechanism that supports various partition schemes for different engines and sources (Section 5).
- We apply Accio to five DS engines and demonstrate that Accio can enable and accelerate query federation (Section 6).

2 BACKGROUND

In this paper, we study the problem of *enabling efficient query federation for DS engines*. As demonstrated in Figure 1, we target the *loosely-coupled* [37] federation setup. Specifically, given a target query engine (e.g., Polars) and *read* access to the 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. We focus on *analytical* workloads and *RDBMS* data sources since they are more challenging to support with much room for improvement.

2.1 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 [7].[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, which confirms the common need for this functionality, reinforcing the motivation of our work. Spark, DuckDB and ClickHouse provide native federation capabilities. Although DataFusion and Polars currently lack direct support for federated queries, there are ongoing discussions and suggested workarounds [4, 13].

However, we find these existing implementations focus primarily on enabling query federation, i.e., fetching remote tables and converting to the local data format, while considerably less efforts are devoted to optimizing the process. Since data transfer is generally considered as a crucial bottleneck in the federation setup [31], 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 [8, 9, 17]) 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 [17]. 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.

2.2 Built-in Approach

Although each engine could independently implement its missing optimizations, our review of their source code revealed 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 [46] 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 [39]. 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.

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

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

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.

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 = T.a AND S.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.a=T.a                  -- t₁(S₁)
q₂*: SELECT ... FROM S WHERE d < 10 AND ...             -- t₂(S₂)
                                  |- q₂⁰: ID < 1,000,000
                                  |- q₂¹: ID ∈ [1,000,000, 2,000,000)
                                  |- q₂²: ID ≥ 2,000,000
q': SELECT ... FROM t₁, t₂, U WHERE t₁.b= t₂.b AND t₂.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 intended to be 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*. This declarative workflow also allows Accio to enable *join pushdown* and *query partitioning* outside the target engine.

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

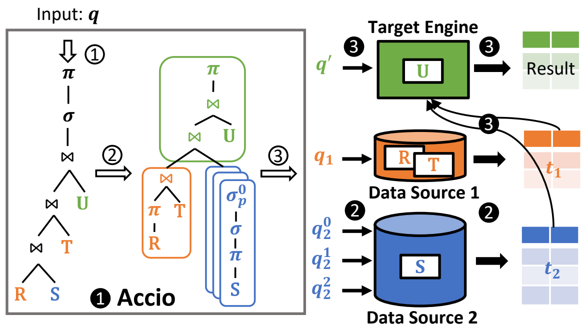


Figure 2: Example workflow. A federated query is ❶ fed into Accio, where it is ① parsed, ② optimized and ③ rewritten into a set of pushdown queries (q_1, q_2^*) and a local query (q'). Pushdown queries are ❷ issued to data sources, whose results serve as input to the local query ❸ executed by the DS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

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.

3.2 Tradeoff Discussion

Compared to the "built-in" solution discussed in Section 2.2, the "bolt-on" design comes with a trade-off.

Advantages. The advantages are twofold. 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) *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. Accio has two main 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 (e.g., Presto, Trino), Accio is only a query rewriter and cannot participate in the final query execution happening inside the federation engine (i.e., target DS engine), making it hard to support adaptive optimization techniques (e.g., dynamic filter [16] supported by Trino).

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] Using the example query of Figure 3, the input query joins four relations from two remote sites (denoted by different colors). 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.

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

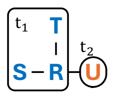
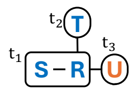
Plan	Graph Partition	Rewritten Query
P_1		SELECT ... FROM t_1, t_2 WHERE t_1.c=t_2.c $t_1(S_1)$: SELECT ... FROM R, S, T WHERE R.a=S.a AND R.b=T.b $t_2(S_2)$: SELECT ... FROM U
P_2		SELECT ... FROM t_1, t_2, t_3 WHERE t_1.b= t_2.b AND t_1.c=t_3.c $t_1(S_1)$: SELECT ... FROM R, S WHERE R.a=S.a $t_2(S_1)$: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

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. $\mathbf{E}' = \{(R_i, R_j) | (R_i, R_j) \in \mathbf{E} \wedge R_i \in \mathbf{V}' \wedge R_j \in \mathbf{V}'\} \wedge \forall R \in \mathbf{V}' : s(R) = S_k$.*

In Figure 3, t_* represent pushdown queries. Since the local query always joins the results of all pushdown queries along with local relations at the target engine, 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, $\cup_{G_i \in P} \mathbf{V_i} = \{R | R \in \mathbf{V} \wedge s(R) \neq S_0\} \wedge \forall G_i, G_j \in P : \mathbf{V_i} \cap \mathbf{V_j} = \emptyset$.*

P_* from Figure 3 are examples of rewrite plans. Finally, we define the join pushdown problem for the "bolt-on" setup:

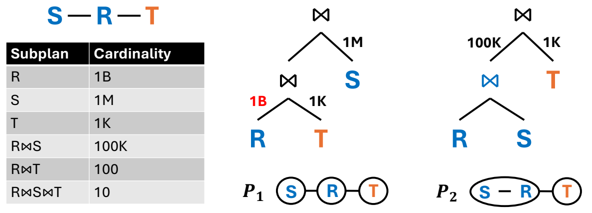
Definition 3 (Bolt-on Join Pushdown). *Given a federated query, the bolt-on join pushdown problem aims to find the valid rewrite plan with the minimum estimated cost.*

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 DBMS data sources (e.g., PostgreSQL, MySQL) on joining small/medium-sized data. Therefore, to find the best plan, one needs to traverse a large search space, which can be up to the Bell number $B(n)$ [57]. More detailed analysis can be found in our technical report [59].

2). No Physical Control. The rewrite plan can only determine the *logical* combinations of joins to be delegated, but cannot control any *physical* aspects of query execution, such as join ordering or the selection of join algorithms. For instance, 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 would 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 the global join 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 query optimization for distributed/federated DBMSs. To the best of our knowledge, we are the first to study this specific problem.



Subplan	Cardinality
R	1B
S	1M
T	1K
R⋈S	100K
R⋈T	100
R⋈S⋈T	10

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.

4.2 Adapting Optimization Techniques from Distributed/Federated Systems

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 [24, 40] extends centralized query optimization to distributed DBMS by introducing the concept of *interesting site*, which is a physical property similar to *interesting order*, indicating the locality that performs the operation. The new physical property can be applied to exhaustive enumeration algorithms (e.g., dynamic programming) to generate the distributed plan. A concrete example can be found in our technical report [59].

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 [40]. As demonstrated in Section 6.2, traversing 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* was first discussed in the context of parallel database systems [35], which assumes that the best parallel plan is a parallelization of the best sequential plan. This idea was then adopted for the query optimization of distributed and federated database systems [25, 55], 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. Trino [15] also adopts this approach.

The limitation of the two-phase approach is that it overlooks the interaction between join pushdown and join ordering. By determining the join order in the first phase without considering data movement costs and pushdown possibilities, this approach may result in expensive plans that stall, waiting for data transfer to complete. Figure 4 presents an illustrative example, where a fact table R is joined with a co-located dimension table S and T from another site. A reasonable join order would first join R and T like P_1, as it would greatly reduce the size of the intermediate data and thus the input of the second join. 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 join order.

4.3 Iterative Approach

The primary issue with existing solutions for distributed/federated systems 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. Therefore, we propose an alternative approach. The key idea is to iteratively identify and push down the most beneficial join, considering join order only for cost estimation using a heuristic algorithm.

The framework is illustrated by Algorithm 1, where joinOrdering(S, R) represents a join ordering algorithm that simulates ordering factors in R on site S. The function benefit is a criterion used to prioritize joins for pushdown, which will be discussed in detail later. First, we initialize the best plan by invoking a join ordering algorithm over all base relations, assuming each of them is fetched to local (line 1). Then, pairwise pushdown benefit scores are collected (line 2-5), and used to push down the joins (line 6-17). Specifically, at each iteration, the join with the highest benefit score is pushed down (line 7), and the new pushdown factor is ordered from scratch (line 8). Then, the benefit scores and join factors are updated (line 9-14). Finally, the new plan is ordered for cost estimation (line 16) if the benefit of the join is greater than 1 (line 15), indicating that the new pushdown does not hurt the performance. And, the best plan is updated if the new one has a smaller cost (line 17-18). The procedure is repeatedly invoked until there is no more join that can be pushed down (line 6).

Algorithm 1: `Iterative Framework`

```
Input: $R = \{R_1, ..., R_n\}$
1  $\hat{R} = R$; $P^* =$ joinOrdering$(S_0, R)$;
2  foreach $R_i \neq R_j \in R : s(R_i) = s(R_j) \neq S_0$ do
3      if $\neg$connected$(R_i, R_j)$ then
4          continue;
5      $B[\{R_i, R_j\}] =$ benefit$(R_i \bowtie R_j)$;
6  while $|B| > 0$ do
7      $\{R_i, R_j\} = \text{argmax}\, B[\{R_i, R_j\}]$;
8      $R' =$ joinOrdering$(s(R_i), \{r | r \in R \wedge (r \in R_i \vee r \in R_j)\})$;
9      $B = B \setminus \{\{R_a, R_b\} | R_a \in \{R_i, R_j\} \vee R_b \in \{R_i, R_j\}\}$;
10     foreach $R_k \in \hat{R} \setminus \{R_i, R_j\} : s(R_k) = s(R')$ do
11         if $\neg$connected$(R_k, R')$ then
12             continue;
13         $B[\{R_k, R'\}] =$ benefit$(R_k \bowtie R')$;
14     $\hat{R} = \hat{R} \setminus \{R_i, R_j\} \cup \{R'\}$;
15     if benefit$(R_i \bowtie R_j) > 1$ then
16         $P =$ joinOrdering$(S_0, \hat{R})$;
17         if $C(P) < C(P^*)$ then
18             $P^* = P$;
19 return $P^*$;
```

It is clear that using Algorithm 1, both P_1 and P_2 from Figure 4 would be generated (in the first and second iterations, respectively), and the one with a lower cost (most likely to be P_2) will be chosen. Algorithm 1 incrementally pushes down one join at each iteration and invokes the centralized join ordering for both global query and the corresponding pushdown query. Thus, let $O(X)$ be the complexity of the join ordering algorithm, the complexity of the iterative framework is $O(X * n)$. Next, we discuss the cost model and pushdown criterion used for the iterative framework in Accio.

Cost Model. Estimating the cost of a given plan is already non-trivial, not even mentioning that as a "bolt-on" rewriter, we know little about the physical executions. Thus, we adopt the calibration-based [39] approach to extend the C_{out} [45] 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 using C_{out} is that it only depends on the cardinality without the need of physical information. Thus, we can then easily abstract the performance of site S into a simple coefficient γ_S. Similarly, we use a parameter τ_S to model the overhead of fetching intermediate result R from site S to local (γ_S and τ_S are non-negative):

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

Join Pushdown Criterion. Using the above annotations, 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:*

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

Note that the implementation choices for the join ordering algorithm, cost model, and join pushdown criterion are mutually orthogonal, and are also orthogonal to the iterative framework.

Intuition & Analysis. Obviously, the iterative approach is heuristic and can lead to suboptimal results. In our report [59], 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, the pushdown benefit indicates the cost improvement of pushing down a single join considering both data movement overhead and performance variance between local and remote systems. Since both τ_{S_k} and γ_{S_k} are non-negative, Equation (1) can be expanded using the cost models:

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

Intuitively, 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. $\frac{|R_1|+|R_2|}{|R_1 \bowtie R_2|}$ heuristically combines these two factors using input and output cardinality as the numerator and denominator, respectively. $benefit(R_1 \bowtie R_2)$ further takes into consideration the execution (γ) and data transfer (τ) overhead of the corresponding data source.

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 [41, 62, 63].

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

Table 2: Dataset table grouping based on prior work [61].

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

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.

Once built, QueryPartitioner can be reused for different data sources through configuration. Alternatively, one could also implement customized partitioners for a specific setup. Note that as a "bolt-on" library, Accio focuses on providing the mechanism, rather than dictating the policy for query partitioning. Next, we describe one of our current implementations and leave it for future work to discover more alternatives.

A Concrete Example. We illustrate an example partitioner for the PostgreSQL data source used in our experiment. From a high level, we evenly partition the CTID column 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.

6 EVALUATION

In this section, we seek to understand: (1) How effective are the optimizations, i.e., join pushdown and query partitioning, provided by Accio? (2) How much improvement can Accio bring to the integrated DS engines? (3) How does the Accio-enhanced DS engine compare with standalone federation systems?

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 all five DS engines studied in Section 2: Spark [21] (v3.5.1), DuckDB [49] (v0.10.3), DataFusion [33] (v39.0.0), Polars [12] (v1.3.0) and ClickHouse [6]. In particular, we adopt the embedded version of ClickHouse, namely chDB [5] (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 [60] to fetch the results of the pushdown queries in the form of Arrow [3], and then convert them to their internal data format. We also slightly modify DuckDB's PostgreSQL scanner so that it can leverage multiple threads for arbitrary query partitioning. Note that these modifications are simple and non-intrusive, requiring less than 100 lines of code in most cases.

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**

Hardware and Platform. We use three dual-socket servers with two 20-core Intel Xeon Gold 6242R CPUs clocked at 3.10GHz. Each server has 80 hyperthreads and 375GB main memory and runs Arch Linux with kernel 6.3. 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 [34]. 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.

Datasets and Workloads. We use the Join Order Benchmark (JOB) [43], which includes 113 queries that join up to 17 tables on a snapshot of the real-world IMDb database. We also evaluate on the TPC-H [14] benchmark with various scale factors (1,10,50). We report the result of ten queries, ranging from joining two to eight relations. To construct a federated setup, we follow previous work [61] to divide the tables of each dataset into three groups as summarized 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 in the rest of this section), each of which places a different group on one data source and the rest on the other.

Implementation. Accio is implemented in Java leveraging Calcite [23], an open source framework that provides building blocks for query processing and optimization (e.g., SQL parser, validator). It exposes interfaces in Python, Java, Rust and C/C++ for easy adoption by various data science tools, and also 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). Cardinality estimation is a difficult problem itself in query optimization and is independent of what we study in this paper. Unless otherwise specified, we issue Explain statements to get estimated values from each PostgreSQL instance for data that reside at the same source and use simple and widely adopted heuristics [43] to estimate the selectivity of cross-site joins. As mentioned in Section 4, our cost model is calibration-based [46] which needs a few parameters. We manually tune these parameters and use a set of default values in our evaluation. Automatic tuning of these parameters is beyond the scope of this paper and interesting future work. We repeat each query five times after warm-up and report the averaged result measured in query latency.

6.2 Efficacy of Accio Optimizations

We evaluate the efficacy of the two optimizations: join pushdown and query partitioning provided by Accio. Due to space limitation, we only present the result on Spark, and put the results on other engines in the technical report [59].

Join Pushdown. We disable query partitioning and compare different join pushdown approaches. We evaluate the three cost-based methods described in Section 4, i.e., Exhaustive, TwoPhase and

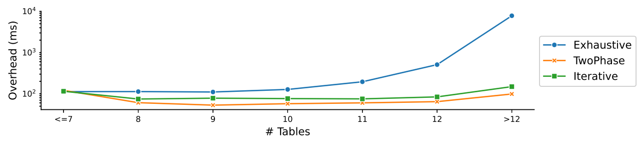


Figure 5: Maximum rewrite overhead of cost-based join pushdown approaches group by # join tables on JOB.

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 [30] 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. Note that this single instance is only for statistical use. The DS engine still needs to fetch data from different data sources for execution.

Table 3 shows the result of running all three table distributions of JOB with a 10Gbps bandwidth on Spark. Iterative shows the best overall performance, which speeds up the entire workload by 2.2×, 1.7× and 2.1× over NoPush, PushAll and TwoPhase, respectively. An interesting finding is that Exhaustive is approximately 20% slower than Iterative on average. We attribute this gap to two main factors. First, since Exhaustive explores the entire search space, it is more sensitive to inaccuracies in our cost estimation, such as errors in cardinality estimates, the cost model, and discrepancies between the logical joins 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.

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 [17], we partition the queries on its first projected numerical column, and manually tune the maximum number of partitions and set it to 4. Figure 6 compares the latency of executing queries from the two workloads on Spark with the 10Gbps bandwidth. We observe that 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 (see [59] for more results).

6.3 How much improvement can Accio bring in?

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, simulating the most straightforward approach users might employ to manually execute each federated query on these engines.

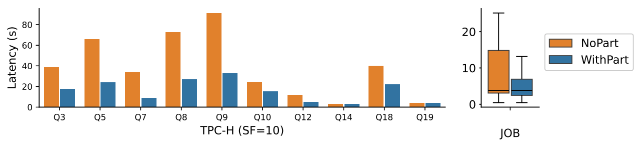


Figure 6: Impact of query partitioning on Spark (10Gbps).

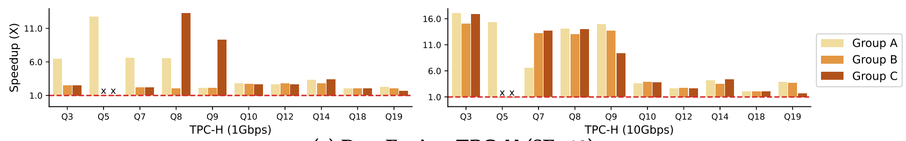
Figure 7a-Figure 7d present the results. The three bars for the same query in Figure 7a and Figure 7c denote the speedup for TPC-H queries on the three table distributions. A "×" is placed on Q5 in Figure 7a due to an OOM error raised by DataFusion, and on Q7 and Q19 in Figure 7c since Polars currently 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 [59]. 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.

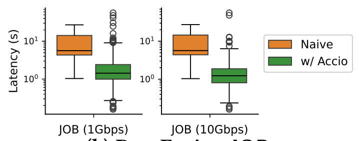
Figure 8c and Figure 8d present the results on DuckDB. We observe that Accio generally provides better performance under 1Gbps bandwidth but may be slower under 10Gbps. This is because DuckDB's native PostgreSQL scanner is highly efficient, especially with query partitioning already enabled. Furthermore, DuckDB itself outperforms PostgreSQL when the entire dataset fits into memory. As a result, when bandwidth is sufficient, fetching the base tables and allowing DuckDB to perform the joins locally tends to be more beneficial. However, Accio sometimes delegates 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.

6.4 Comparison with Standalone FDBMS

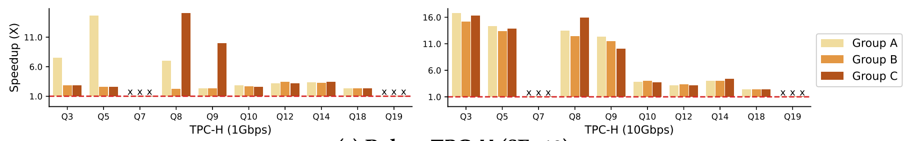
We compare the performance between Trino [15] (release 453), a fork from Presto [53], and Spark enhanced by Accio since both systems are scalable and built for big data analysis. To ensure a fair comparison, we deploy Trino on the same server and configure it to use at least the same amount of CPU and memory resources as Spark. To show the efficacy of Accio, 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 [59] 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 figure in the rightmost column shows the result on JOB, where native Spark



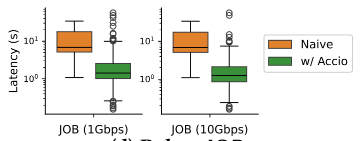
(a) DataFusion TPC-H (SF=10).



(b) DataFusion JOB.

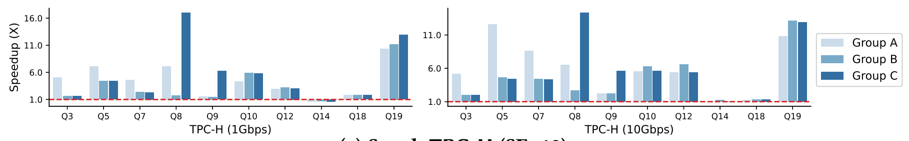


(c) Polars TPC-H (SF=10).

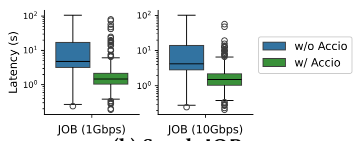


(d) Polars JOB.

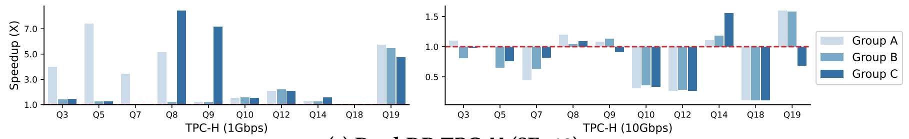
Figure 7: Accio on enabling query federation (red line: y=1; ×: cannot be executed by the Naive baseline).



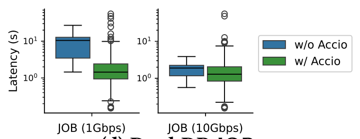
(a) Spark TPC-H (SF=10).



(b) Spark JOB.



(c) DuckDB TPC-H (SF=10).



(d) DuckDB JOB.

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

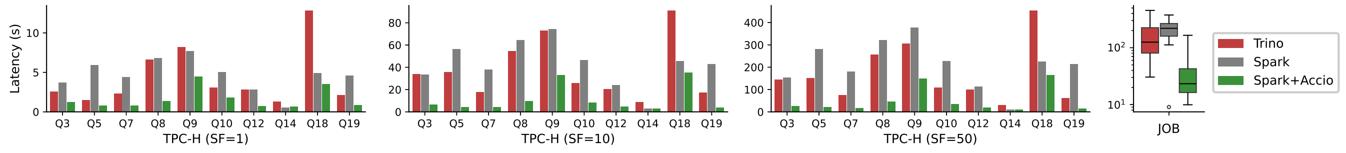


Figure 9: Trino v.s. Spark w/wo Accio (10Gbps).

is approximately 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 [19, 22, 42] to our technical report [59].

7 RELATED WORK

We discuss related works as a complement to the existing query federation support of DS engines investigated in Section 2.

Standalone Federation Systems. Federated DBMSs [54] aim to coordinate a collection of autonomous DBMSs. Garlic [24, 51] integrates different data sources using a mediator-wrapper architecture, which is widely adopted later. More recently, ployglot and cross-platform data systems [18–21, 26, 28, 32, 36–38, 42, 53, 58] further renewed interest in query federation. In particular, Presto [53] and its derivative products, including Trino [15], AWS Athena [2] and Huawei Hetu [11], 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 [61] uses a machine learning model to dynamically choose the best federation engine for each query using the cost estimation from each instance. XDB [31] 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.

Distributed/Federated Query Optimization. The join push-down optimization shares similarities with previous work in distributed and federated query optimization [39, 46], where algorithms were proposed from different angles, such as plan enumeration [27, 40, 56], inter-site join scheduling [29] and cross-database communication reduction [25]. However, Accio assumes different setups. Unlike existing techniques, Accio is a "bolt-on" library that is not coupled with a global optimizer or a specific execution engine, resulting in less information available, less control over execution, and a higher requirement on extensibility. Moreover, Accio works on a modern federation setup, where the performance of the participating systems may vary drastically and high network bandwidth is available. It makes query partition crucial for performance, which did not receive much attention in prior work.

8 CONCLUSION

In this paper, we studied the problem of "bolt-on" query federation. We identified the issues of supporting federated queries inside existing supports and proposed to decouple this feature into Accio, a middleware library designed to facilitate efficient query federation for various DS engines. We integrated Accio with five popular engines and conducted extensive experiments, showing that Accio can easily "bolt-on" to these systems while delivering high performance, which is also robust under various setups.

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. chDB. https://clickhouse.com/docs/en/chdb.

[6] 2024. ClickHouse. https://github.com/ClickHouse/ClickHouse.

[7] 2024. Database-like ops benchmark. https://duckdblabs.github.io/db-benchmark/.

[8] 2024. DuckDB Postgres extension. https://github.com/duckdb/postgres_scanner.

[9] 2024. DuckDB SQLite extension. https://github.com/duckdb/sqlite_scanner.

[10] 2024. Federated Querying using DuckDB on blockchain data. https://kowalskidefi.medium.com/federated-querying-using-duckdb-x-postgres-on-blockchain-data-5391518601ee.

[11] 2024. Huawei Hetu Cyberverse. https://www.huaweicloud.com/product/live/dspace.html.

[12] 2024. Polars: Fast multi-threaded DataFrame library in Rust and Python. https://github.com/pola-rs/polars.

[13] 2024. scan_database feature. https://github.com/pola-rs/polars/issues/9091.

[14] 2024. TPC-H Homepage. http://www.tpc.org/tpch.

[15] 2024. Trino, a query engine that runs at ludicrous speed. https://trino.io/.

[16] 2024. Trino Dynamic Filtering. https://trino.io/docs/current/admin/dynamic-filtering.html.

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

[18] 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

[19] 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

[20] 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

[21] 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

[22] 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

[23] 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

[24] 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

[25] 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

[26] 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

[27] Weimin Du, Ming-Chien Shan, and Umeshwar Dayal. 1995. Reducing Multidatabase Query Response Time by Tree Balancing. In *Proceedings of the 1995 ACM SIGMOD International Conference on Management of Data, San Jose, California, USA, May 22-25, 1995*, Michael J. Carey and Donovan A. Schneider (Eds.). ACM Press, 293–303. https://doi.org/10.1145/223784.223846

[28] 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

[29] Cem Evrendilek, Asuman Dogac, Sena Nural, and Fatma Ozcan. 1997. Multidatabase Query Optimization. *Distributed Parallel Databases* 5, 1 (1997), 77–113. https://doi.org/10.1023/A:1008674905987

[30] 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

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

[32] 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

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

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

[35] 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

[36] 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

[37] 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

[38] 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

[39] 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

[40] 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

[41] 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.

[42] 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

[43] 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

[44] Guido Moerkotte and Thomas Neumann. 2006. Analysis of Two Existing and One New Dynamic Programming Algorithm for the Generation of Optimal Bushy Join Trees without Cross Products. In *Proceedings of the 32nd International Conference on Very Large Data Bases, Seoul, Korea, September 12-15, 2006*, Umeshwar Dayal, Kyu-Young Whang, David B. Lomet, Gustavo Alonso, Guy M. Lohman, Martin L. Kersten, Sang Kyun Cha, and Young-Kuk Kim (Eds.). ACM, 930–941. http://dl.acm.org/citation.cfm?id=1164207

[45] 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

[46] 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

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

[48] 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
[49] Mark Raasveldt and Hannes Muehleisen. [n.d.]. *DuckDB.* https://github.com/duckdb/duckdb
[50] Matthew Rocklin. 2015. Dask: Parallel computation with blocked algorithms and task scheduling. In *Proceedings of the 14th python in science conference.* Citeseer.
[51] 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
[52] Patricia G. Selinger, Morton M. Astrahan, Donald D. Chamberlin, Raymond A. Lorie, and Thomas G. Price. 1979. Access Path Selection in a Relational Database Management System. In *Proceedings of the 1979 ACM SIGMOD International Conference on Management of Data, Boston, Massachusetts, USA, May 30 - June 1*, Philip A. Bernstein (Ed.). ACM, 23–34. https://doi.org/10.1145/582095.582099
[53] 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
[54] 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
[55] 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
[56] D. K. Subramanian and K. Subramanian. 1998. Query Optimization in Multidatabase Systems. *Distributed Parallel Databases* 6, 2 (1998), 183–210. https://doi.org/10.1023/A:1008691331104
[57] 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
[58] 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
[59] 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.
[60] 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
[61] 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
[62] 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.
[63] 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

A MORE TECHNICAL DETAILS

A.1 Search Space of Join Pushdown

Since the tables in the pushdown queries are all connected, we exclude any pushdown query that requires a cross product. Additionally, we ensure that the evaluation of any in-place join condition is not delayed. For instance, consider a query with three join factors from the same remote site with each pair of factors connected by a join condition. Any pushdown query that contains two of the three joins must also contains the third.

If we treat each local factor V as a subgraph $(\{V\}, \emptyset)$, each valid plan pushdown plan can be seen as a connected graph partition with the constraint that every edge within a partition must be able to be pushed to the same remote site that supports join. Thus, depending on the data distribution and the query graph, the number of valid plan of a query with n join factors can vary from 1 (e.g., all factors are located on a different engine) to the Bell number $B(n)$ [57] (e.g., a clique query [44] with all data are on the same remote engine).

A.2 Example of Exhaustive Approach

Algorithm 2 shows the exhaustive approach we implement for evaluation. It is an implementation over DPsize [52], a wildly used dynamic programming approach to find the optimal join order in centralized systems. The main modification here is to track multiple sites where the join could be carried out in the DP table. The adoption in Algorithm 2 is tailored for the federation setup we discussed in Section 2, where only read access to remote sites is needed.

Algorithm 2: `Exhaustive Approach`

```
   Input: $R = \{R_1, ..., R_n\}$
1  foreach $R_i \in R$ do
2      $dpTable[\{R_i\}, s(R_i)] = R_i$;
3      if $s(R_i) \neq S_0$ then
4          $dpTable[\{R_i\}, S_0] = fetch(R_i)$;
5  for $1 < s \leq n$ do
6      foreach $\mathbf{R_i}, \mathbf{R_j} \subseteq R : |R_i| + |R_j| = s$ do
7          if $R_i \cap R_j \neq \emptyset \vee \neg$conncted$(R_i, R_j)$ then
8              continue;
9          if $s(R_i) = s(R_j) \neq S_0$ then
10             addJoinPlan$(R_i, R_j, s(R_i))$;
11         addJoinPlan$(R_i, R_j, S_0)$;
12 return $dpTable[R, S_0]$;
13
14 addJoinPlan$(R_i, R_j, S)$
15 $P = dpTable[R_i, S] \bowtie_S dpTable[R_j, S]$;
16 if $C(P) < C(dpTable[R_i \cup R_j, S])$ then
17     $dpTable[R_i \cup R_j, S] = P$;
18 if $S \neq S_0$ then
19     $P' = fetch(P)$;
20     if $C(P') < C(dpTable[R_i \cup R_j, S_0])$ then
21         $dpTable[R_i \cup R_j, S_0] = P'$;
```

Complexity. The time complexity of the general exhaustive approach is $O(s^3 * 3^n)$ where n and s denotes the number of join factors and the number of sites involved [40]. However, since each table has only one copy and data can only transfer from remote to local engine, at most two sites (i.e., the local and the remote site if the intermediate result is not already sent to local) are "interesting" for every sub-problem of dynamic programming. Therefore, the complexity of Algorithm 2 is $O(3^n)$, which is exponential and not able to control the overhead of rewrite to a negligible level.

A.3 Detailed Analysis of Iterative Approach

We reason for the optimality of *pushdown benefit* for star queries with restrictions. We start with a query Q as follows:

- Q is a star query. W.l.o.g., we let R_0 be the fact table and $R_1, ..., R_n$ as n dimension tables.
- All the join conditions are independent (i.e., $\forall R_i (i > 0)$, there is a fixed join selectivity sel_i).

Since Q is a star query, all plans are linear with R_0 within the first join, and $R_1, ..., R_n$ can be joined in arbitrary order. Any valid pushdown plan P can be represented in the form of $fetch(R_0 J_0) \bowtie fetch(J_1)$, where J_0 and J_1 are two join sequences and $\forall R_i \in J_0 : s(R_i) = s(R_0)$. $fetch(J_1)$ is defined as fetching each factor in J_1 respectively and then joining at local in the corresponding order. Both J_0 and J_1 can be empty.

Let J be a join sequence with k factors $R'_1, ..., R'_k$ in order with selectivity $sel'_1, ...sel'_n$ respectively. We define a function $g(J, l) = \prod_{i=1}^{l} |R'_i| sel'_i (l \leq k)$. Then for an arbitrary pushdown plan $P = fetch(R_0 J_0) \bowtie fetch(J_1)$, we have:

$$\begin{aligned} C(P) = \quad & \gamma_{s(R_0)} \cdot |R_0| \cdot \textstyle\sum_{i=1}^{|J_0|} g(J_0, i) \\ & + |R_0| \cdot g(J, |J_0|) \cdot \textstyle\sum_{i=1}^{|J_1|} g(J_1, i) \\ & + \tau_{s(R_0)} \cdot |R_0| \cdot g(J, |J_0|) + \textstyle\sum_{R_i \in J_1} |R_i| \cdot \tau_{s(R_i)}, \end{aligned} \tag{2}$$

where the three lines compute the cost of conducting joins at remote engine, conducting joins locally, and data transfer respectively.

We define $\tilde{\mathbf{R}}$ as the set of dimension tables that are co-located with R_0 (i.e., $\tilde{\mathbf{R}} = \{R_i | i \in \{1, ..., n\} \wedge s(R_i) = s(R_0)\}$). We also define $\prec$ as a binary function given two dimension tables such that:

$$R_i \prec R_j = \begin{cases} \text{true}, & |R_i| \cdot sel_i < |R_j| \cdot sel_j \wedge |R_i| > |R_j|; \\ \text{false}, & \text{otherwise}. \end{cases}$$

Next, we show that if $\prec$ is true between any pair of dimension tables of Q, prioritizing joins with pushdown benefit like Algorithm 1 could enumerate the best pushdown plan as a candidate.

Lemma 1. *Let $P^* = fetch(R_0 J_0) \bowtie fetch(J_1)$ be the best pushdown plan (i.e., has minimum estimated cost). For $\forall R_i, R_j \in \tilde{\mathbf{R}} (R_i \neq R_j)$, the following holds: $R_i \prec R_j \Rightarrow R_i \in J_0 \vee R_j \in J_1$.*

Proof. We prove this by contradiction. Suppose $\exists R_i, R_j \in \hat{\mathbf{R}} (R_i \neq R_j)$ s.t. $R_i \prec R_j$ and $R_j \in J_0 \wedge R_i \in J_1$ (i.e., P^* pushdown $\bowtie_j$ but not $\bowtie_i$). We can construct another plan $P' = fetch(R_0 J'_0) \bowtie fetch(J'_1)$ from P^* by swapping the positions of R_i and R_j. Next, we show that P' is a better pushdown plan than P^*.

W.l.o.g., let the position of R_j in J_0 is $p (1 \leq p \leq |J_0|)$, and R_i in J_1 is $q (1 \leq q \leq |J_1|)$. By Equation 2, we have[4]:

$$\begin{aligned} C(P^*) - C(P') = \quad & \gamma_{s(R_0)} \cdot |R_0| \cdot \textstyle\sum_{i=p}^{|J_0|} (g(J_0, i) - g(J'_0, i)) \\ & + |R_0| \cdot \textstyle\sum_{i=1}^{q-1} g(J_1, i) \cdot (g(J_0, |J_0|) - g(J'_0, |J_0|)) \\ & + \tau_{s(R_0)} \cdot (|R_0|((g(J_0, |J_0|) - g(J'_0, |J_0|)) + |R_i| - |R_j|) \end{aligned}$$

Since $|R_i| \cdot sel_i < |R_j| \cdot sel_j$, we have $\forall i = p, ..., |J_0| : g(J_0, i) - g(J'_0, i) > 0$. Also, since $|R_i| > |R_j|$, we have $C(P^*) - C(P') > 0$. Thus, P' is a better than P^*. Contradiction. □

Lemma 1 indicates that the best pushdown plan will never pushdown $\bowtie_j$ but not $\bowtie_i$ if $R_i \prec R_j$ is true. In such cases, we can say that $R_i (\bowtie_i)$ has a higher pushdown priority than $R_j (\bowtie_j)$.

[4]If $i < q$, $g(J_1, i) = g(J'_1, i)$, otherwise $g(J_0, |J_0|) g(J_1, i) = g(J'_0, |J_0|) g(J'_1, i)$.

Lemma 2. *If $\forall R_i, R_j \in \tilde{\mathbf{R}} (R_i \neq R_j)$, either $R_i \prec R_j$ or $R_j \prec R_i$ is true, then given any prefix $X = R_0 J' (J' \subseteq \tilde{\mathbf{R}})$, we have $R_i \prec R_j \Leftrightarrow benefit(X \bowtie R_i) > benefit(X \bowtie R_j)$.*

Proof. It is clear that if $R_i \prec R_j$ is true, $b(X \bowtie R_i) = \frac{|X|+|R_i|}{|X| \cdot |R_i| \cdot sel_i} > \frac{|X|+|R_j|}{|X| \cdot |R_j| \cdot sel_j} = b(|X| \bowtie |R_j|)$, thus $benefit(X \bowtie R_i) > benefit(X \bowtie R_j)$ ($\Rightarrow$).

On the other hand, if $benefit(X \bowtie R_i) > benefit(X \bowtie R_j)$, then $b(X \bowtie R_i) > b(X \bowtie R_j)$. And suppose $R_j \prec R_i$ is true, $b(X \bowtie R_i) < b(X \bowtie R_j)$ (using $\Rightarrow$). Contradiction. Therefore, $R_j \prec R_i$ is false. And since $R_i \prec R_j \vee R_j \prec R_i$ is true, $R_i \prec R_j$ must be true. ($\Leftarrow$) □

Lemma 2 shows the isomorphism between $\prec$ and pushdown benefit when there is a pushdown priority defined between any pair of joins.

Theorem 1. *Let $P^* = fetch(R_0 J_0) \bowtie fetch(J_1)$ be the best pushdown plan for Q. If $\forall R_i, R_j \in \tilde{\mathbf{R}} (R_i \neq R_j)$, either $R_i \prec R_j$ or $R_j \prec R_i$ is true, then Algorithm 1 finds the best pushdown plan P^* if the join ordering method adopted chooses the best centralized join order, which is also followed by each involved system.*

Proof. For $\forall R_i, R_j \in \hat{\mathbf{R}}$ s.t. $R_i \in J_0 \wedge R_j \in J_1$, we know that $R_j \prec R_i$ is false by the contrapositive of Lemma 1. And since $R_i \prec R_j \vee R_j \prec R_i$ is true, $R_i \prec R_j$ must be true. Let X be the prefix at any iteration step in Algorithm 1, by Lemma 2, we know that $benefit(X \bowtie R_i) > benefit(X \bowtie R_j)$. In other words, all relations in J_0 have higher pushdown benefit score than any relation in J_1.

Since Algorithm 1 picks the relation with maximums benefit score at each iteration, at one of the iteration, it will merge all relations in J_0 with R_0 as a single node X. And since all joins in $J_0 (resp. J_1)$ are conducted by the remote (resp. local) engine, J_0, J_1 must has the best centralized join order. Therefore, if the join ordering method can find the best centralized order, P^* can be found. □

Intuition for General Queries. $R_i \prec R_j$ can be seen as a condition of comparing both input (i.e., $|R_i|, |R_j|$) and output (i.e., $|R_i| \cdot sel_i, |R_j| \cdot sel_j$) cardinality of two candidate joins. While the former relates to data transfer cost if the join is not pushed down, the latter corresponds to the join cost as well as data transfer cost if the join is pushed down. Indeed, the condition of $\prec$ is restrictive and real-world workload are not all star queries. Intuitively, $b(R_i \bowtie R_j) = \frac{|R_i|+|R_j|}{|R_i \bowtie R_j|}$ heuristically combines the two factors of $\prec$ altogether, with inputs' cardinality as numerator and output cardinality of denominator. While $benefit(R_i \bowtie R_j) = b(R_i \bowtie R_j) \cdot \frac{\tau_S}{\tau_S + \gamma_S} + \frac{1}{\tau_S + \gamma_S}$ further takes the execution and data transfer overhead of the corresponding query engine into consideration, prioritizing joins that can benefit more from pushdown.

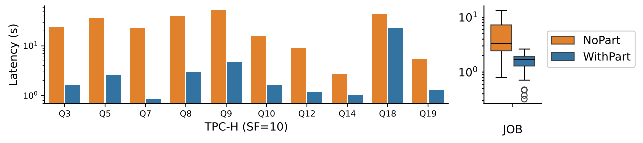
B MORE EVALUATION RESULTS

B.1 Join Pushdown (on More DS Engines)

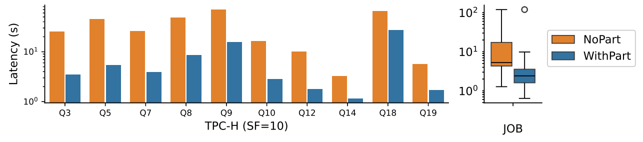
Table 4 presents the average query latency of all JOB queries (three table distributions) using different pushdown approaches same with Section 6.2 on the remaining four engines. It is clear that Iterative consistently outperforms other baseline approaches across all tested DS engines.

Table 4: Comparison of join pushdown approaches on average latency using Spark (10Gbps, JOB, unit: seconds).

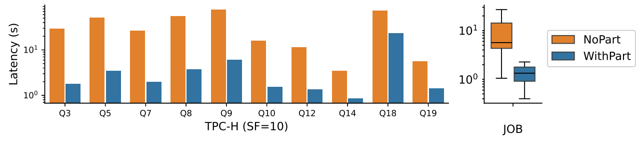
	NoPush	PushAll	Exhaustive	TwoPhase	Iterative
DuckDB	4.997	4.883	3.232	5.201	**2.935**
chDB	9.117	5.144	4.120	9.132	**3.849**
DataFusion	10.22	5.479	6.817	10.23	**4.029**
Polars	12.76	6.055	7.170	12.88	**4.590**



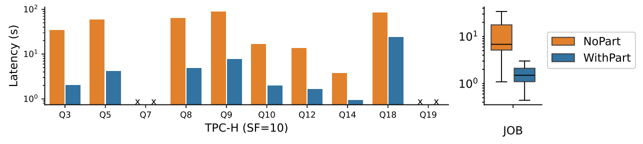
(a) DuckDB.



(b) chDB.



(c) DataFusion.



(d) Polars.

Figure 10: With v.s. without query partition given 10Gbps bandwidth (×: not supported by Polars).

B.2 Query Partition (on More DS Engines)

Figure 10 compares the latency of executing queries from the two workloads on DuckDB, chDB, DataFusion and Polars under 10Gbps network bandwidth (other engines show similar results). We adopt the example query partitioner introduced in Section 5 for all these four engines, and set the maximum number of partitions to 16 for chDB and 32 for the remaining engines, respectively. We put a "×" on Q7 and Q19 since they include disjunctive join conditions, which is not supported by Polars for now. We can see that by enabling query partition, single query's performance can be accelerated significantly by around 3× to 16× on TPC-H, the overall latency of JOB is about 3× to 8× faster.

B.3 chDB v.s. chDB + Accio

The impact of Accio to chDB is shown in Figure 12a and Figure 12b. In particular, some queries cannot be completed in two minutes using native chDB, for which we use two minutes to compute the speedup (also annotate with "+" for TPC-H queries). For the remaining queries that finish within two minutes, we observe that chDB +Accio can outperform chDB by up to 42×. In addition to the two optimizations introduced by Accio, the substantial performance improvement can also be attributed to the join reordering, which occurs as a side effect of the query rewriting process in Accio. This is particularly significant, as chDB does not natively support join reordering and directly follows the order specified in the declarative input query during execution.

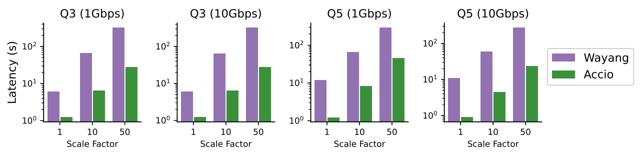


Figure 11: Accio v.s. Wayang over Spark + PostgreSQL.

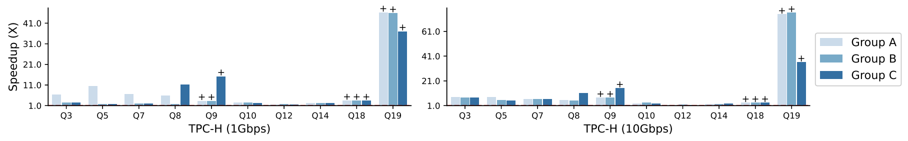
B.4 Trino v.s. Spark + Accio (More Table Distributions)

We put the comparison results over table distributions B and C to Appendix B.5. Similar to group A shown in Section 6.4, Accio consistently enables Spark to outperform Trino, whereas the native federation support in Spark sometimes results in slower performance.

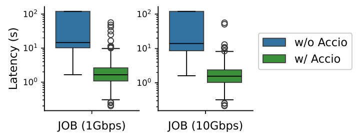
B.5 Apache Wayang v.s. Spark + Accio

We compare Accio with Apache Wayang (v0.7.1) [19, 22, 42], a unified data processing framework that integrates and orchestrates multiple data platforms. Specifically, we let Accio and Wayang run the exact same PostgreSQL + Spark cross-platform. We deploy Wayang on the same server with Spark, and enable an additional Java platform as it is sometimes more efficient. All tables are placed in a single PostgreSQL data source because Wayang currently only supports one instance per platform. And since Wayang currently lacks a ready-to-use declarative interface, we manually construct Q3 and Q5 of TPC-H using its Scala API.

The result of running Q3 and Q5 of TPC-H is illustrated in Figure 11. The x-axis indicates the scale factor of the dataset. It is evident that Accio can leverage the capabilities of the PostgreSQL + Spark cross-platform more efficiently (up to 10×). The main reason is that Wayang directly generates RDD plans for Spark execution instead of declarative queries like Accio does, missing the optimizations done by SparkSQL [21]. However, we also want to note that here we only compare Accio and Wayang for handling relational queries, while the latter also supports other kinds of cross-platform task such as running machine learning algorithms.

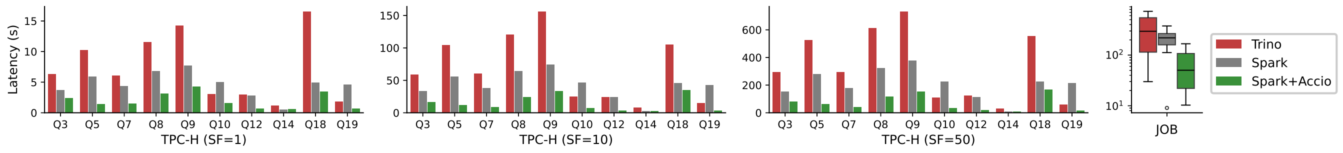


(a) chDB TPC-H (SF=10).

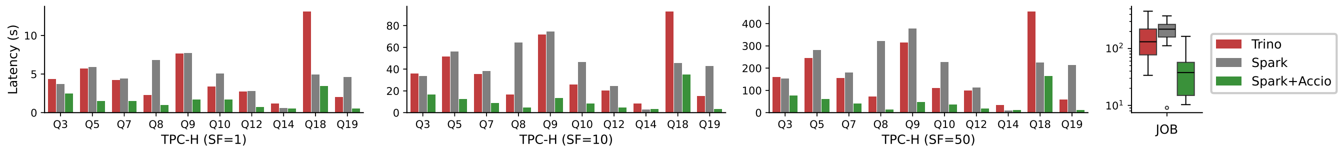


(b) chDB JOB.

Figure 12: Performance improvement after integrating with Accio (+: timeout in two minutes by native chDB).



(a) Group B.



(b) Group C.

Figure 13: Trino v.s. Spark w/wo Accio (10Gbps).