In-Situ Cross-Database Query Processing

Haralampos Gavriilidis[◇] Kaustubh Beedkar[◇] Jorge-Arnulfo Quiané-Ruiz[*] Volker Markl[◇#]
Technische Universität Berlin[◇] *IT University of Copenhagen*[*] DFKI GmbH[#]
{gavriilidis, kaustubh.beedkar, volker.markl}@tu-berlin.de joqu@itu.dk

***Abstract*—Today's organizations utilize a plethora of heterogeneous and autonomous DBMSes, many of those being spread across different geo-locations. It is therefore crucial to have effective and efficient cross-database query processing capabilities. We present XDB, an efficient middleware system that runs cross-database analytics over existing DBMSes. In contrast to traditional query processing systems, XDB does not rely on any mediating execution engine to perform cross-database operations (e.g., joining data from two DBMSes). It delegates an entire query execution including cross-database operations to underlying DBMSes. At its core, it comprises an optimizer and a delegation engine: the optimizer rewrites cross-database queries into a delegation plan, which captures the semantics as well as the mechanics of a fully decentralized query execution; the delegation engine then deploys the plan to the underlying DBMSes via their declarative interfaces. Our experimental study based on the TPC-H benchmark data shows that XDB outperforms state-of-the-art systems (Garlic and Presto) by up to 6× in terms of runtime and up to 3 orders of magnitude in terms of data transfer.**

I. INTRODUCTION

Leveraging all intra- and inter-organizational data is crucial for data-driven decision-making in many domains [1], [2]. For example, consider a pandemic scenario where doctors work with data scientists to explore the effectiveness of different vaccine types. Data scientists need to query data from different organizations (data silos), e.g., municipality offices, vaccination centers, and local health centers, each maintaining their own database management system (DBMS).

Yet, producing valuable insights in such scenarios requires effective and efficient ways of combining data across data silos. A common approach is to first consolidate all available data into a centralized repository (e.g., a data warehouse) through ETL pipelines [3], [4]. Nevertheless, implementing ETL pipelines is tedious and error-prone, and does not allow performing ad-hoc queries on fresh data [5]. Furthermore, centralizing intra- or inter-organizational data is often not feasible in the first place, e.g., in our pandemic scenario. Thus, we require systems for efficient *cross-database query processing*, where ad-hoc queries seamlessly combine data from different DBMSes.

Why Not Existing Decentralized Approaches? Decentralized DBMSes offer a rather suitable alternative to ETL's centralized approach. However, modern distributed data paradigms, such as parallel and distributed database systems (DDBMS) [6]–[10], P2P databases (PDBMS) [11]–[13], and federated databases (FDBMS) [14]–[16], are either ineffective or inefficient for processing cross-database queries. On the one hand, DDBMS and PDBMS are ineffective: DDBMSes are tailored towards homogeneous environments (same vendor, schema, among others), make strong assumptions about data partitioning across physical nodes, and require complete control over individual DBMSes for query optimization and execution. PDBMSes—although relaxing assumptions with regard to homogeneity and autonomy—focus mainly on discoverability and query routing, and lack support for complex queries.On the other hand, FDBMSes, including recent work on polystores, and cross-platform systems [17]–[22], are suitable for cross-database query processing but are inefficient as they employ the Mediator-Wrapper (MW) approach [23]–[25].

[*] Work done while author was at Technische Universität Berlin

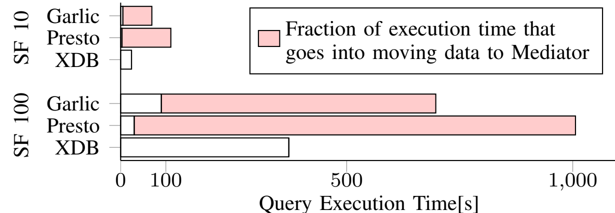


Fig. 1: Cross-database query processing approaches.

Inefficiency in the MW Approach. From a user's perspective, MW-based systems are desirable as they offer a single query interface and hide the complexity of interacting with multiple DBMSes. However, from a system perspective the MW approach is rather inefficient: MW-based systems first decompose a cross-database query into multiple subqueries that target individual DBMSes, then execute the resulting subqueries on the corresponding DBMS, and finally fetch the subquery results to perform remaining (cross-database) query operations. It is this "centralized" processing of cross-database operations that: (i) requires maintaining and providing resources for an additional execution engine (the mediator), which also requires additional administration expertise; and (ii) leads to expensive and unnecessary data movement as the mediator centralizes all (intermediate) data. As a remedy, modern systems [26]–[28] employ massively parallel processing and techniques to scale the mediator out and up [29], [30]. Yet, they still suffer from high maintainability and data movement costs imposed by the MW approach.

To demonstrate the above limitations, we benchmarked two MW-based systems for TPC-H Q3 on distributed tables (experiment details in Section VI-A). We examine the performance of a single-node Garlic-like system [31], and Presto [26], a scale-out federated query engine. Figure 1 shows the results for two scale factors (SF). Overall, we observe that the actual execution time (white bar) accounts for roughly 15% of the total execution time for Garlic and 3% for Presto. Both systems spent most of the execution time on moving data to the mediator (red bar).

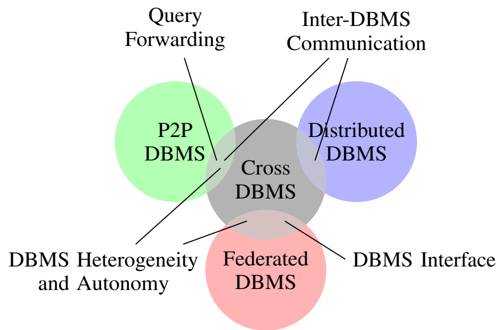


Fig. 2: Distributed data processing characteristics in Cross-DBMS that are common with current approaches.

A New Approach. We introduce a new distributed data processing paradigm, which we refer to as *in-situ cross-database* query processing (Cross-DBMS, for short). [1] We advocate for keeping the user-side simplicity of FDBMS (to interface with heterogeneous and autonomous DBMSes) while leveraging the advantages of decentralized query processing techniques, such as in DDBMS and PDBMS, i.e., processing data in-place within DBMSes, without a mediating execution engine. Cross-DBMS shares several characteristics of current decentralized DBMSes as shown in Figure 2. Cross-DBMS leverages the capabilities of underlying DBMSes for query processing, including cross-database operations. At the same time, it achieves performance close to the "actual" execution times of our baselines (Figure 1). Thus, Cross-DBMS, renders an effective solution for cross-database query processing: It can deal with DBMS heterogeneity and autonomy; It does not require maintaining an additional execution engine; It incurs reduced data movement costs by moving data directly between underlying DBMSes.

Achieving in-situ cross-database query processing is challenging with regard to both optimization and execution. First, when optimizing cross-database queries we have to holistically consider (i) operator ordering, (ii) operator placement, and (iii) how intermediate data should be moved between DBMSes. Second, our proposed processing model requires coordinating underlying DBMSes to decentrally execute queries. Although existing DBMS interoperability allows querying external data, it is not sufficient for inter-operating multiple DBMSes in tandem to achieve a fully decentralized execution.

We propose XDB, a middleware system that enables efficient in-situ cross-database query processing over existing DBMSes. To the best of our knowledge, XDB is the first system to tackle all the above challenges. Figure 1 shows that XDB's in-situ processing can indeed outperform MW-based systems. Overall, XDB achieves performance improvements by employing a cross-database optimizer that produces query delegation plans. The core idea behind a delegation plan is to offload data processing and inter-DBMS communication instructions to the underlying DBMSes. Following these instructions, DBMSes collaboratively execute user queries in a fully decentralized fashion by building inter-DBMS pipelines that transfer relevant data (subquery results) among them.

[1] Our notion of in-situ query processing differs from that studied in literature [32]–[34]; see Section VII.

In summary, after introducing state-of-the-art and its limitations in Section II, we make the following contributions:
Section III: We propose a novel *in-situ cross-database query processing* (Cross-DBMS) approach that decentrally executes cross-database queries through a lightweight middleware.
Section IV: We introduce our cross-database optimizer and the concept of a *delegation plan*, which captures the semantics of decentralized query execution as well as the mechanics of inter-DBMS communication.
Section V: We introduce our cross-database execution mechanism and our *delegation* approach that is based on query rewriting and transforming delegation plans into DBMS-specific instructions to enable executing cross-database queries "in-situ" across multiple DBMSes in a pipelined fashion.
Section VI: We evaluate XDB against baseline systems (Garlic, Presto, and ScleraDB) for cross-database query processing. Our results show that XDB outperforms baseline systems by up to 6$\times$ w.r.t. query execution time and up to 3 orders of magnitude w.r.t. amount of data transferred during query execution.

We then discuss additional related work in Section VII and conclude in Section VIII.

II. Cross-database Query Processing

We now detail our a motivating scenario to better illustrate the need for *cross-database query processing* and discuss why current approaches fall short.

A. Motivating Scenario

Consider again our motivating scenario from Section I, which generalizes to several real-world scenarios where data resides in different DBMSes. These DBMSes (potentially geo-distributed) are managed by different departments (or organizations), which are willing to share their information [1], [2]. More concretely, consider the case of the Municipal Office (MO) of an imaginary city called Credo. MO has several independent departments: a citizens' department that stores and manages citizen's information; a health department that manages citizens' health records and COVID-19-related information (e.g., antibody measurements); and a newly established vaccination center to manage vaccine-related information. Each department uses a DBMS (potentially from different vendors). Assume that the citizens' department uses `CDB`, the health department uses `HDB`, and the vaccination center uses `VDB` (as shown in Table I).

After the initial rollout of COVID-19 vaccines, the MO's chief health officer (CHO) would like to analyze the effectiveness of different vaccine types. For this, she would like to measure COVID-19 antibodies (U/ml) in different age groups for people over 20 years. The analytical query shown in Figure 3 precisely fulfills her information needs.[2] Although the query looks simple, it hides the intricacy that executing it requires the CHO to combine data from three different, heterogeneous, and autonomous DBMSes (across departments), which is far from trivial. We refer to such kind of queries as *cross-database queries*.

[2] In this paper, we assume that tables across DBMSes can be joined without transforming join attributes. We consider other cases as important future work.

TABLE I: DBMSes with their corresponding schema

DBMS	Table(s)	Schema
CDB	Citizen	(id, name, age, address)
VDB	Vaccines	(id, name, type, manufacturer)
	Vaccination	(c_id, v_id, date)
HDB	Measurements	(id, c_id, date, u_ml)

```
SELECT v.type, AVG(m.u_ml),
    case when c.age between 20 and 30 then '20-30'
         when c.age between 30 and 40 then '30-40'
         ...
    end as 'age_group'
FROM CDB.Citizen c, VDB.Vaccines v, VDB.Vaccination vn, HDB.
     Measurements m
WHERE c.id = vn.c_id AND c.id = m.c_id
    AND v.id = vn.v_id AND c.age > 20
GROUP BY age_group, v.type
```

Fig. 3: Example cross-database query.

Supporting the above cross-database query efficiently is particularly challenging in such a highly autonomous scenario:
DBMS Heterogeneity. Departments are free to select their underlying DBMS from any vendor that fits their needs. For instance, in our motivating scenario, CDB may be a PostgreSQL database, while VDB may be a MariaDB database. Hence, we need to process cross-database queries atop DBMSes that are *heterogeneous* with regard to SQL dialects, local optimizers, physical operator implementations, and internal cost models.
Storage Autonomy. DBMSes across departments are typically operated and managed by different independent entities. Thus, the data schemata within different DBMSes are designed independently of each other. In such scenarios, data in one DBMS is often not replicated or sharded across others.
Execution Autonomy. In many scenarios, underlying DBMSes are either legacy systems hosted on-premise or modern DBMSes running as managed services on the cloud. Hence, the only way to communicate with DBMSes is via their declarative SQL interface, as it may not be possible to equip physical (or virtual) nodes with additional software, e.g., a local query processor. As a result, when receiving user queries, DBMSes decide themselves about the choice of physical operators and their order, as well as other aspects of query execution.

B. State-of-the-Art & Drawbacks

We categorize existing approaches for processing data stored across different DBMSes into Parallel and Distributed DBMS (DDBMS; e.g., [6]–[10]), P2P DBMS (PDBMS; e.g., [11]–[13]), and Federated DBMS (FDBMS; e.g., [14]–[16], [26], [27]). These systems, however, are either ineffective or inefficient for processing cross-database queries, shown in Table II.

DDBMSes focus on scaling out storage and compute, and are composed of homogeneous DBMSes (i.e., from the same vendor). Furthermore, they withhold storage and execution autonomy – data is replicated or sharded across DBMSes and individual DBMSes expose their physical operators, which are used by the DDBMS for query optimization. PDBMSes, in contrast, mainly focus on data discoverability and query routing over individual DBMSes that are completely decentralized and autonomous. To enhance scalability for serving requests and to ensure fault tolerance, PDBMSes replicate data among

TABLE II: In-situ Cross-DBMS characteristics.

Characteristics	DDBMS	PDBMS	FDBMS	XDB
DBMS Heterogeneity	✗	✓	✓	✓
Storage Autonomy	✗	–[a]	✓	✓
Execution Autonomy	✗	✓	✓	✓
No additional QP engine	✓	✗	✗	✓
Inter-DBMS interactions	✓	–[b]	✗	✓

[a] data at times is replicated (e.g., [11]). [b] Requires additional software.

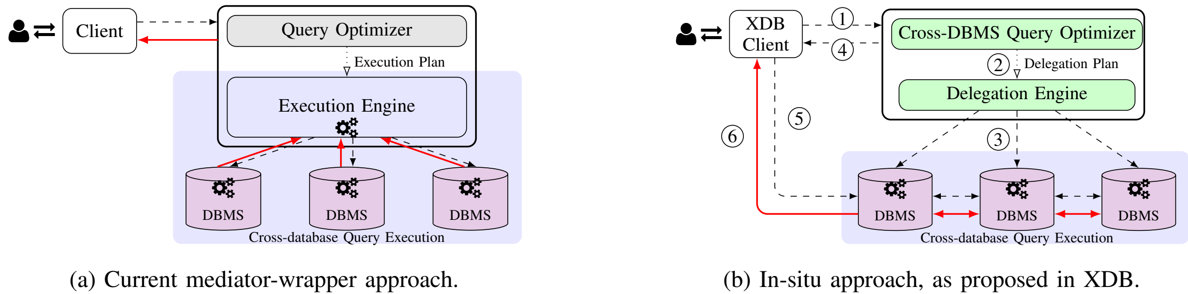
nodes, and hence do not fulfill the requirement for storage autonomy. Moreover, to join a PDBMS, individual DBMSes require additional software to be installed on the participating physical nodes, i.e., DHTs and query processors. Finally, most PDBMS approaches focus on simple queries, such as lookups, and do not consider complex relational queries.

FDBMSes, on the other side, are effective but inefficient for processing cross-database queries. They are based on a so-called Mediator-Wrapper (MW) architecture [14], [23], [26]–[28], [31]. Figure 4a gives a high-level overview of how queries are executed in an MW fashion, where, typically, wrappers (also known as connectors or adapters) provide access to data stored in DBMSes. Overall, a mediator first decomposes a query into a set of *local* and *cross-database* (global) *operations*. Local operations correspond to fragments of the query that require inputs from a single DBMS and can be natively performed within the DBMS. For instance, in our example cross-database query, the `selection` operation filtering all citizens with an age over 20 can be locally executed within the CDB DBMS. Likewise, cross-database operations are those that require inputs from multiple DBMSes. The mediator takes care of these cross-database operations, such as the `join` operation, in our query example, which joins data from the CDB and VDB DBMSes. The mediator sends local operations to the underlying DBMSes and gathers the intermediate results to perform the cross-database operations.

Although MW-based systems have been largely successful, they have some inherent drawbacks due to the mediator's "centralized" approach for cross-database operations. First, the MW architecture incurs a high performance overhead as illustrated in Figure 1. In the context of Credo's CHO, the MO will thus have extra costs when running an MW-based system, e.g., Presto [26] or SparkSQL [27]. Besides the performance overhead, the MO will also have to provision additional computing resources for the mediator. The reader might think that a cloud-based MW system [35] can solve MO's problem. However, this is not entirely true, as a cloud-based MW system would also lead to a high monetary cost due to its expensive data transfers (the red lines in Figure 4a). As a result, MW-based systems incur high maintenance and data movement cost next to high processing overhead.

III. XDB: A Cross-Database System

We propose XDB, a middleware for cross-database query processing that, in contrast to current MW-based systems, is not equipped with an execution engine. Instead, it employs an efficient *in-situ* cross-database query processing mechanism to entirely delegate query execution to underlying DBMSes.



(a) Current mediator-wrapper approach.

(b) In-situ approach, as proposed in XDB.

Fig. 4: Cross-database query processing approaches.

Figure 4b illustrates the in-situ cross-database query processing approach used by XDB.[3] From an end user's perspective, XDB offers similar functionality to current MW-based systems: It takes a declarative (SQL) cross-database query and returns its results; It offers a unified view of underlying distributed data via Global-as-a-View mappings [36]—We assume, without any loss of generality, that the global schema is a union of local schemas. From a system's perspective, in contrast to MW-based systems, XDB enables the underlying DBMSes to execute a cross-database query entirely without the need for a mediating execution engine.

At its core, XDB comprises two components: a cross-database query optimizer and a delegation engine. Given a query (step ①), the optimizer translates it into a *delegation plan* (step ②), which is a directed acyclic graph (DAG): nodes are tasks containing algebraic expressions assigned to a particular DBMS, and edges are task dependencies, i.e., data movement between DBMSes. More concretely, XDB translates a query into a sequence of operations that underlying DBMSes can execute, leading to a fully decentralized execution. In essence, a delegation plan encapsulates query semantics comprising local and cross-database operations as well as mechanics of data movement between DBMSes. To do so, the optimizer performs rule-based and cost-based optimization, such as operator pushdown and placement, to minimize the overall execution cost. We discuss the optimization details in Section IV.

The delegation engine, then, is responsible for deploying the delegation plan on the individual DBMSes to achieve in-situ cross-database query execution. For this, we exploit a DBMS's existing support for the SQL/MED standard [37], which is implemented in many modern systems [38]–[43], to enable interoperability among DBMSes. In more detail, the delegation engine uses the DBMS connectors to translate and execute DBMS-specific instructions corresponding to the algebraic expressions found in a delegation plan. Notice that these instructions are `DDL` statements, which do not execute the query but only "prepare" the underlying DBMSes for in-situ cross-database query execution via SQL/MED. These `DDLs` create short-lived relations and do not assume access to modify the existing schema. We discuss delegation in Section V.

Finally, XDB sends an XDB query, which is a DBMS-specific `SELECT` statement, back to the client (step ④). The client, then, executes the XDB query on the specified DBMS (step ⑤) to receive the results (step ⑥). One of the salient aspects of our work is rewriting a user query to an XDB query using our optimizer and delegation engine. It is this XDB query that actually triggers the in-situ cross-database query execution without having XDB partake at all in the execution.

The above query processing characteristics in XDB (also recall Figure 2) are reminiscent of those studied in the context of DDBMS and PDBMS [23]: the difference lies in the way we realize them for a (mediator-less) decentralized execution environment comprising heterogeneous and autonomous DBMSes.

[3]For brevity, we omit other standard components, e.g., parser and catalog.

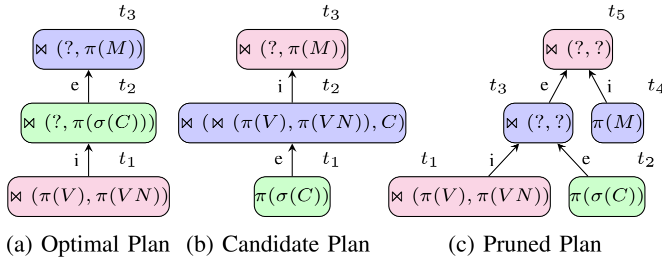
IV. Cross-Database Query Optimization

The cross-database query optimizer gets a query as input and outputs a delegation plan by passing the query through three components: First, the *Logical Optimizer* applies traditional optimization techniques, such as join ordering and selection/projection pushdown; Then, the *Plan Annotator* decides the operator placement and data movement; Finally, the *Plan Finalizer* fuses multiple operators into tasks, which represent the execution units that are performed by the underlying databases. In what follows, we first introduce a delegation plan as an intermediate representation that captures the semantics of decentralized cross-database query execution. We, then, detail our optimization process that finds an "optimal" delegation plan.

A. Delegation Plan

A salient aspect of XDB is that it delegates the entire execution to underlying DBMSes by sending them DBMS-specific instructions via their declarative interfaces. It is therefore desirable to have a delegation plan as an intermediate query plan representation. A delegation plan facilitates efficient decentralized query execution without having fine-grained control over physical plan operators and their order of execution.

In more detail, a *delegation plan* is a set of operations that describes how a query must be executed on the underlying DBMSes. Formally, we denote a delegation plan by a directed graph $G = (T, E)$, where T is a set of tasks and E denotes the set of edges as dataflow operations between tasks. Furthermore, a *task* $t \in T$ is a tuple (r, a) where r is an algebraic expression corresponding to a (sub) query and a is an annotation that prescribes the DBMS that must evaluate the expression. For expository reasons, we often use the notation $a : r$ to denote a task. For example, Figure 5 shows three delegation plans that correspond to the example query of Section II-A. The plan in Figure 5a comprises three tasks (for now ignore the edge labels): t_1, t_2, and t_3. Here, task t_1, $\texttt{VDB}{:}\bowtie(\pi(\texttt{V}), \pi(\texttt{VN}))$, specifies a join operation between the (projected) relations V and VN.



(a) Optimal Plan (b) Candidate Plan (c) Pruned Plan

Fig. 5: Delegation plans for the query of Section II-A, with DBMS annotations: VDB, CDB, HDB.

Task t_2, CDB:$\bowtie (?, \pi(\sigma(\texttt{C})))$, then joins the output of t_1 with the (filtered and projected) relation C. Finally, task t_3, HDB:$\bowtie (?, \pi(\texttt{M}))$, joins the output of t_2 with (projected) relation M. Note that we use the symbol "?" as a placeholder for a relation that is the result of evaluating a task's expression executed at another DBMS. For instance, the placeholder in task t_2 (and in t_3) denotes the join output from t_1 (t_2, respectively).

We now turn our attention to the dataflow operations between tasks. In particular, we focus on inter-DBMS tasks that are executed on different DBMSes. Recall that we rely on a DBMS's implementation of the SQL/MED standard to move data between DBMSes. We propose two ways to move data between DBMSes during query execution. One way is to *implicitly* move data between two DBMSes, i.e., by pipelining the output of the first task to the second. Another way is to *explicitly* move data by materializing the output of the first task (as a relation) on the DBMS executing the second task. We will discuss in Section V how to use SQL/MED for such data movements.

In the context of dataflow operations in a delegation plan, we denote by $t_1 \xrightarrow{i} t_2$ the output of task t_1 that is implicitly moved (i.e., pipelined) to task t_2, and by $t_1 \xrightarrow{e} t_2$ the data movement that is explicit (i.e., materialized). Continuing our running example, in Figure 5a, data is moved implicitly between tasks t_1 and t_2 while it is explicitly moved between tasks t_2 and t_3. The choice between these two kinds of data movement can significantly impact the query execution time. While an implicit dataflow operation allows parallelizing two dependent tasks, an explicit dataflow operation may lead to DBMS-specific optimizations. For instance, a DBMS could parallelize an operation or employ an efficient hash join by creating a hash table on the smaller table. Furthermore, when deploying tasks to the underlying DBMSes, dataflow operations help to define the dependencies between tasks. This allows parallelizing delegation and execution tasks, i.e., when two independent tasks are found.

Besides these two data movement options, the different alternatives to group operations into a single task and to prescribe a DBMS per group lead to a complex search space of possible delegation plans. Figures 5b and 5c illustrate two (of many) alternative delegation plans for our example query. For instance, in Figure 5b, task t_1 computes a relation $\pi(\sigma(\texttt{C}))$, whose output is explicitly (e) moved to task t_2, which then computes the two-way join $\bowtie (\bowtie (\pi(\texttt{V}), \pi(\texttt{VN})), \texttt{C})$. The output of this join is then moved implicitly (i) to task t_3 to compute the final join $\bowtie (?, \pi(\texttt{M}))$, where ? denotes the intermediate relation. As we will discuss later, XDB's optimizer never considers a plan like the one in Figure 5c. XDB prunes plans with cross-database operations that happen on a different DBMS than this of the inputs. In this particular pruned plan, t_3 would be placed on HDB even though its inputs t_1 and t_2 reside on VDB and CDB.

B. Optimization Process

We now discuss our optimization process to find an "optimal" delegation plan, which, when deployed, leads to a decentralized query execution with the lowest estimated cost. Achieving this is challenging because traditional approaches for deriving optimal execution plans cannot be readily used for deriving optimal delegation plans in XDB: DBMSes and compute nodes are black-boxes to XDB, which itself is not equipped with an execution engine. The optimizer, therefore, cannot reason about the characteristics of physical plan operators and their cost across different DBMSes (both at the local and at the cross-database level). Moreover, as the execution of operations is delegated to the underlying DBMSes via (already existing) declarative interfaces, the optimizer has little control over the order of operations within a task. To make things worse, combining operations into a task, prescribing a DBMS for a task, and the choice of dataflow operations between tasks makes the solution space very large even for simple queries.

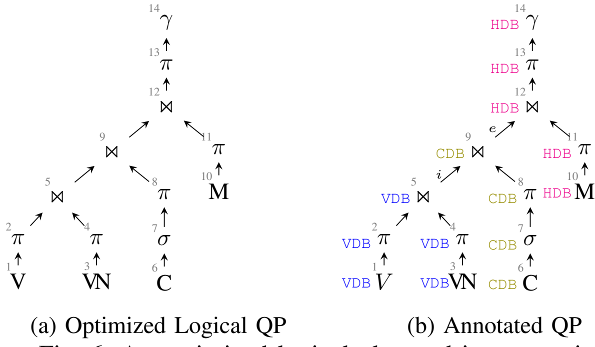
We tackle the above challenges by employing a three-step optimization process that involves 1) logical optimization, 2) plan annotation, and 3) plan finalization.

1) Logical Optimization: To cope with black-box DBMSes, we first optimize queries at the logical level. We determine the right order of plan operators, which is crucial for reducing the overall data movement between DBMSes. We use textbook optimization techniques, e.g., query rewrites (e.g., selection and projection pushdowns) and join-order optimization that overall reduce intermediate data [44], [45].[4] Figure 6a shows an optimized logical plan for our example query of Section II-A.

2) Plan Annotation: Given an optimized logical plan, we now determine the placement of operators (annotations) along with the type of dataflow operation between operators. These annotations allow us to group multiple operations into a task and determine dataflow operations between tasks.

Notations. Before delving into the specifics of the annotation process, we first introduce some necessary notations. We denote by $\mathcal{O}$ the set of all plan operators in the optimized logical plan. For two operators $o, o' \in \mathcal{O}$, $o' \rightarrow o$ denotes that the output of operator o' is directly consumed by operator o. For example, in Figure 6a, we have $o_1 \rightarrow o_2$ and $o_5 \rightarrow o_9$. For a unary operator o let o_l be its input operator (i.e., we have $o_l \rightarrow o$) and for a binary operator o let o_l and o_r be its left and right input operators respectively (i.e., we have $o_l \rightarrow o$ and $o_r \rightarrow o$). While we use the same notation for both unary and binary operators, it is always clear from the context to which one we refer to. Furthermore, let $\mathcal{A}$ be the set of annotations that denote underlying DBMSes. In our motivating example of

[4] In this paper, we only consider left-deep trees. Our optimization and execution paradigm (Section V), however, are agnostic to the plan shape. Our preliminary experiments (omitted due to space constraints) show that the degree of parallelism gained by bushy plans increases the performance, and hence we plan to investigate this further and extend the optimizer in future work.



(a) Optimized Logical QP (b) Annotated QP

Fig. 6: An optimized logical plan and its annotation.

Section II-A, we have, for instance, $\mathcal{A}_{ex} = \{ \texttt{VDB}, \texttt{HDB}, \texttt{CDB} \}$. We also denote as $\mathcal{A}(o)$ the annotation for operator o and as $\mathcal{A}(\mathcal{O}' \subseteq \mathcal{O}) = \{ \mathcal{A}(o) \mid o \in \mathcal{O}' \}$. We denote by $\mathcal{A}(o \to o') \in \{ i, e \}$ as an annotation for the edge where i and e correspond to implicit and explicit dataflow operations.

Annotation Process. Our goal is to determine an annotation $\mathcal{A}(o)$ for all operators $o \in \mathcal{O}$ and an annotation $\mathcal{A}(o' \to o)$ for each edge $o' \to o$ in the optimized logical plan. The overall idea is to perform a bottom-up traversal that propagates the annotation found in the leaf nodes through unary operators, until a cross-database operator is found. Then, we decide the placement and movement types for the inputs of a cross-database operator by costing the available options. In particular, we annotate the plan by applying one of the following rules:

Rule 1. For a leaf (tablescan) operator o, $\mathcal{A}(o) = a$, where a is an annotation corresponding to the DBMS on which the table resides. Intuitively, we annotate leaf nodes (table scans) with the DBMS where the table resides.

Rule 2. For a unary operator o, $\mathcal{A}(o) = \mathcal{A}(o_l)$ and $\mathcal{A}(o_l \to o) = i$. Essentially, we annotate each unary operator with the annotation of its input (child operator).

Rule 3. For a binary operator o where $\mathcal{A}(o_l) = \mathcal{A}(o_r)$, $\mathcal{A}(o) = \mathcal{A}(o_l)$ and $\mathcal{A}(o_l \to o) = \mathcal{A}(o_r \to o) = i$. In other terms, we annotate each binary operator having two inputs with the same annotation with this same annotation.

Rule 4. For a binary operator o where $\mathcal{A}(o_l) \neq \mathcal{A}(o_r)$, $\mathcal{A}(o) = a$, $\mathcal{A}(o_l \to o) = a_l$, and $\mathcal{A}(o_r \to o) = a_r$, where a, a_l, and a_r are obtained by solving the following optimization problem:

$$\underset{a \in \mathcal{A}; a_l, a_r \in \{ i,e \}}{\text{argmin}} \texttt{cost}(o,a) + \texttt{cost}(o_l \xrightarrow{a_l} o, a) + \texttt{cost}(o_r \xrightarrow{a_r} o, a) \quad (1)$$

where $\texttt{cost}(o,a)$ is the cost of executing the operator at the DBMS a and $\texttt{cost}(o' \xrightarrow{x} o, a) =$

$$\texttt{moveCost}(o', \mathcal{A}(o'), a) \qquad \text{if } x = i \quad (2)$$

$$\texttt{moveCost}(o', \mathcal{A}(o'), a) + \texttt{scanCost}(o', a) \qquad \text{if } x = e \quad (3)$$

is the cost of moving the output of operator o' from DBMS $\mathcal{A}(o')$ to a via dataflow operation $x \in \{ i, e \}$. Intuitively, this means that we annotate a binary operator having two inputs with different annotations, with that annotation that yields the cheapest cost. Note that for an explicit movement, we also consider the cost of scanning the relation (corresponding to the query rooted at operator o') at DBMS a as explicit movement requires materializing intermediate data (recall Section IV-A).

During the annotation process, we traverse the optimized logical plan using a depth-first post-order traversal and apply one of the rules above at each node. Figure 6b illustrates the output of the annotation process for our running example. While rules 1–3 are straightforward in their application, Rule 4 requires solving the optimization problem given in Equation 1. For example, consider the optimized logical plan in Figure 6a. We start with post-order traversal from node 1 and annotate the sub-plan rooted at node 5 with VDB using Rules 1–3. Likewise, using Rules 1 and 2, we annotate all nodes and edges until node 8 with CDB (see Figure 6b): data movement between operators with the same annotation is always implicit. For node 9, we decide the operator placement along with data flow operations such that it minimizes the execution cost as defined in Equation 1. For instance, we determine $\mathcal{A}(o_9) = \texttt{CDB}$ and $\mathcal{A}(o_8 \to o_9) = i$. Continuing our annotation process, we annotate nodes 10 and 11 with HDB, using Rule 1–2, and use Equation 1 to determine $\mathcal{A}(12) = \texttt{HDB}$ and $\mathcal{A}(o_9 \to o_{12}) = e$. The remaining nodes are then annotated with HDB.

A key challenge during the annotation process is efficiently solving Equation 1. While it may seem trivial at a first glance, it incurs a high cost to evaluate alternatives, considering our distributed and autonomous DBMS setup. Recall that one of the challenges in cross-database optimization is coping with black-box DBMSes. To this end, we follow a "consulting" approach during plan annotation to determine the cost of executing a operator at a certain DBMS and the cost of moving data between two DBMSes. More specifically, we probe the underlying DBMSes during plan annotation through our DBMS connectors, which provide costing functions by wrapping EXPLAIN-like statements. This approach is similar to the one proposed for the Garlic system in [46]. The main difference is that unlike Garlic, partial plans in XDB comprise cross-database operations.[5] Furthermore, to compute all the costs using Equation 1 will end up requiring $O(|\mathcal{A}| \cdot |\mathcal{O}|)$ communication rounds in the worst case. To reduce this high cost of evaluating alternatives, we replace $\mathcal{A}$ with $\mathcal{A}(\{ o_l, o_r \})$ in Equation 1, i.e., for each cross-database join, we only consider the two input annotations as potential options. This simplification is based on the observation that moving two relations R and S respectively from DBMSes a_R and a_S into a third DBMS a_T has a higher data transfer cost than moving either R to a_S (or S to a_R) as $|R| + |S| > \max(|R|, |S|)$.

Additionally, we note that replacing $\mathcal{A}$ with $\mathcal{A}(\{ o_l, o_r \})$ in Equation 1 assumes that all DBMSes are inter-connected. Other network topologies can be supported by constraining the possible values of set $\mathcal{A}$ depending on the network.

3) Plan Finalization: As a last step, we proceed to group multiple operators into tasks. Our approach aims to minimize the number of tasks by grouping successive operators with the same annotation into one task. A small number of tasks requires less communication during plan delegation, and furthermore allows underlying DBMSes to locally optimize queries. To create the final tasks, we traverse the annotated plan using

[5] Our approach requires that cost estimates from different DBMSes have the same unit. In this paper, we follow a simple calibration approach [47]–[49] to align costs across DBMSes. While this worked well for our experiments, we consider aligning or learning [50] different cost models as future work.

a modified depth-first post-order traversal in which for each node we compare its annotation to its parent's. Whenever the annotation differs or when the node is the root node, we create a task by grouping all operators below the tree. For example, in Figure 6b, we group nodes 1–5 into task t_1, nodes 6–9 into task t_2, and nodes 10–14 into task t_3. During the traversal, when transitioning to a node with a different annotation, we create a dummy operator as a new child of the currently visited node. This dummy operator is then treated as a placeholder for the input relation. For example, finalizing the plan in Figure 6b leads to the delegation plan shown in Figure 5a, where the "?" in task t_2 indicates the input for the join and projections on task t_1.

Algorithm 1 Delegating a query to produce its XDB query

Require: Delegation Plan $\mathcal{G}$ **Ensure:** XDB Query $\mathcal{Q}$

1: $t \leftarrow \mathcal{G}$.GETROOT()
2: $\mathcal{Q} \leftarrow$ PROCESSTASK(t)
3:
4: PROCESSTASK(t)
5: **for all** $t' \in t$.GETCHILDREN() **do**
6: $R_v \leftarrow$ PROCESSTASK(t')
7: $R'_v \leftarrow$ CREATEFOREIGNTABLE($R_v, t.a$)
8: **if** $t' \xrightarrow{e} t$ **then**
9: $R_m \leftarrow$ CREATELOCALTABLE($R'_v, t.a$)
10: replace $?_t$ in $t.r$ with R_m
11: **else** replace $?_t$ in $t.r$ with R'_v
12: $R_v \leftarrow$ CREATEVIRTUALTABLE($t.r, t.a$)
13: **return** R_v

V. Cross-database Query Execution

We now discuss the mechanics of cross-database delegation. During the delegation phase (Section V-A) XDB rewrites a delegation plan into DBMS-specific instructions. After deploying these instructions, we enter the execution phase (Section V-B) that triggers an in-situ cross-database query execution.

A. Delegation Phase

The delegation phase "prepares" the underlying DBMSes for executing a given delegation plan. Recall that each node (task) $t = (r, a)$ in a delegation plan encapsulates the algebraic instruction r that the DBMS a must execute, and each edge $t_1 \xrightarrow{x} t_2,\ x \in \{i, e\}$ denotes the movement type as either implicit (i) or explicit (e). The delegation engine, for each task and its dependency, sends DBMS-specific instructions to underlying DBMSes, which transforms a delegation plan into a cascade of views chained with foreign tables. The evaluation of these views leads to an in-situ cross-database query execution. Before unfolding the specifics, let us discuss the key techniques that form the basis of our approach.

Naive Execution. Consider the following example that involves joining two tables residing on two different DBMSes. Assume that we have tables `R(x,y)` on DBMS a_R and `S(x,z)` on a_S. Also consider that for the query `select * from R, S where R.x=S.x`, we have the delegation plan $xdp = (a_R : \texttt{R}) \rightarrow (a_S : \bowtie (?, \texttt{S}))$ (for now, ignore the specific movement types). Intuitively, executing this plan on the underlying DBMSes requires first executing the query `select * from R` on a_R, then moving (exporting and importing) the result, say `R'(x,y)`, from a_R to a_S, and finally executing the query `select * from R', S where R'.x=S.x` on a_S.

Although the above approach is straightforward, it is worth noting that it requires a "mediator" to explicitly coordinate the execution. More specifically, the mediator has to take care of both executing the two queries and moving data between DBMSes. Such an approach is not only cumbersome to implement, but also inefficient because exporting and importing data leads to expensive data movement costs. Moreover, such an approach defeats the purpose of having an in-situ query execution. Therefore, the challenge resides in delegating query execution, including moving intermediate query results between DBMSes *during execution*, without having any mediating entity partaking in the execution.

Leveraging SQL/MED. To efficiently execute cross-database queries in a black-box environment, we exploit SQL/MED for inter-DBMS communication during cross-database query execution. SQL/MED (Management of External Data) is a part of the SQL standard that deals with how one DBMS can integrate its data with data stored "outside" of it [37]. A core component of SQL/MED is the *wrapper interface* that enables viewing external data locally as *foreign tables*. A DBMS supporting the wrapper interface allows executing a SQL query that references both local and foreign tables. On a high level, the DBMS first decomposes such a query into local and remote fragments. Then, the DBMS initiates the execution of each remote fragment on the remote DBMSes, fetches the results, and finalizes the execution.

We now illustrate how we can effectively use the concept of foreign tables to efficiently move (intermediate) data between systems during query execution, all without having to coordinate executing queries on different systems. Reconsider our above example $xdp = (a_R : \texttt{R}) \rightarrow (a_S : \bowtie (?, \texttt{S}))$ (again, ignore the specific movement types). Exploiting the concept of foreign tables, we first create a foreign table `R'` on a_S that points to table `R` on a_R. Executing the query `select * from R', S` on a_S will automatically trigger an execution (via a_S's foreign wrapper) of the query `select * from R` on a_R. Then, a_S fetches the output of this latter query over from a_R and completes the execution. This approach captures the gist of our delegation approach. It neither requires coordinating the execution of the query across DBMSes a_R and a_S, nor it requires a mediator to move intermediate data between a_R and a_S. In essence, creating a foreign table `R'` and executing a local join `S` $\bowtie$ `R'` on a_S is all we need to de-centrally execute query xdp. Note that SQL/MED only provides a building block that facilitates communication between two DBMSes. In what follows, we introduce techniques to leverage SQL/MED to decentrally execute queries over multiple DBMSes.

Preventing Undesirable Executions. Using foreign tables to execute delegation plans, where each task's algebraic instructions involve additional operations (such as projections or filters), requires particular attention. This is because DBMS vendor-specific implementation of the SQL/MED standard may have undesirable effects. To illustrate such an undesirable case, consider again our above example but now with a slightly

DDL 1 @VDB

```
CREATE VIEW VVN AS SELECT v.type, vn.c_id
FROM Vaccines v, Vaccination vn
WHERE v.id = vn.v_id
```

DDL 2-1 @CDB

```
CREATE FOREIGN TABLE VVN(type, c_id) SERVER VDB;
```

DDL 2-2 @CDB

```
CREATE VIEW CVVN AS SELECT c.id, v.type,
case when c.age between 20 and30 then'20-30'
     ... end as 'age_group'
FROM VVN v, Citizen c
WHERE c.id = vn.c_id AND c.age > 20
```

Fig. 7: DDL statements for delegation plan of Figure 5a.

different delegation plan $xdp' = (a_R : \pi_x(\sigma_p(\text{R}))) \to (a_S : \bowtie (?, \text{S}))$, where p is some predicate. Following the above approach of using foreign table R′ and executing a local query $\text{S} \bowtie \pi_x(\sigma_p(\text{R}'))$ on a_S may lead to an undesirable execution: the projection and filter could be executed on a_S. This can happen as wrappers across DBMS have different capabilities in terms of "pushing down" operations to a remote DBMS. As a consequence, we might end up executing xdp' as follows: $(a_R : \text{R}) \to (a_S : \pi_x(\sigma_p(\bowtie (?, \text{S}))))$.

To avoid the above situation and not rely on vendor-specific implementation of the wrappers, we additionally make use of virtual relations when using foreign tables. Continuing the above example, rather than first creating the foreign table R′ on a_S, we first create a virtual relation $\text{R}_v \equiv \pi_x(\sigma_p(\text{R}))$ on a_R, then a foreign table R'_v on a_S, and execute the query $\text{S} \bowtie \text{R}'_v$ locally on a_S. Such an approach allows us to preserve the semantics of xdp. As a result, we can execute queries as specified by their delegation plans.

Enforcing Explicit Data Movements. Using the concept of foreign tables as explained so far leads to pipelining the output of a task to its parent and therefore corresponds to our notion of implicit data movement between tasks. To explicitly move data, we materialize intermediate data by creating an additional relation on the local DBMS. For example, for $xdp' = (a_R : \pi_x(\sigma_p(\text{R}))) \xrightarrow{e} (a_S : \bowtie (?, \text{S}))$, we additionally create a (physical) relation $\text{R}_m \equiv \text{R}'_v$ on a_S and execute the local query $\text{S} \bowtie \text{R}_m$, whereas before R'_v is a foreign table pointing to the virtual relation $\text{R}_v \equiv \pi_x(\sigma_p(\text{R}))$ on a_R. It is worth noting here that moving data between DBMSes is still carried out natively by foreign wrappers.

Deploying a Delegation Plan. Algorithm 1 generalizes all the above techniques to delegation plans with three or more DBMSes. Here, for a task t, we denote by $t.r$ and $t.a$ the algebraic instruction and the annotation, respectively. We process each task in a given delegation plan $\mathcal{G}$ using a depth-first traversal, in which for each task t we create a virtual relation R_v on DBMS $t.a$ (line 12). For example, recall the delegation plan from Figure 5a. For this delegation plan, we first create a virtual table on VDB for the task t_1. For instance, the CREATEVIRTUALTABLE(...) function on line 12 translates to executing the DDL 1 shown in Figure 7 to create the view R_v =VNN. During the delegation process, while recursing to the parent task, we first create a foreign table R'_v on DBMS $t.a$ (line 7). In our running example, we create a foreign table on CDB that points to R_v. For example, the

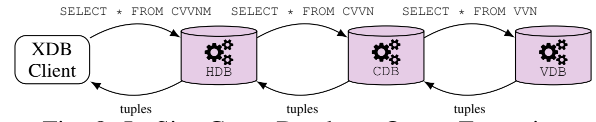


Fig. 8: In-Situ Cross-Database Query Execution.

CREATEFOREIGNTABLE(...) function translates to executing the DDL 2-1 in Figure 7 to create foreign table VNN. Moreover, as the movement from VDB to CDB is implicit, we replace the placeholder in the task's instruction by the foreign table (i.e., we have the instruction $\bowtie (\text{VNN}, \pi(\sigma(\text{C})))$; line 11), and create a virtual table CVNN on CDB by executing the DDL 2-2 (line 12). Finally, we follow the same procedure for the root task, where we first create the foreign table at HDB and then create a local table (via the CREATELOCALTABLE(...); line 9) as data is moved by an explicit movement (recall Figure 5a).

Having executed the DBMS-specific DDL statements on the respective DBMSes, the Delegator has successfully deployed the delegation plan. This means that the DBMSes are now primed for in-situ cross-database query execution. The final step (Algorithm 1; line 2) of the delegation phase is returning the XDB query to the client, which triggers the actual execution. The XDB query is a simple `SELECT * FROM <table>` query, where `table` is the view corresponding to the root task of the delegation plan. In our running example, the XDB query is `SELECT * FROM CVVNM` on HDB. Executing this query triggers the in-situ cross-database execution pipeline.

B. Execution Phase

After submitting a user's query to XDB, the XDB Client receives an XDB query in return. The XDB Client, then executes (transparently from the user) the XDB query on the DBMS prescribed by the root task in $\mathcal{G}$. As mentioned above, the XDB query is of the form `select * from <table>`, where the table is defined as a view over local and/or foreign tables. Evaluating such views triggers a recursive evaluation of remote views (defined as foreign tables) via the SQL/MED interface.

We illustrate the execution through our running example in Figure 8. First, the XDB Client executes the XDB query `SELECT * FROM CVVNM` on HDB. To evaluate CVVNM, HDB fetches CVVN from CDB. As this particular task dependency is explicit, HDB will in this case materialize CVVN from CDB. However, as CVVN is defined on CDB as a foreign table (cf. Figure 7 DDL 2-2) over C and VVN from VDB, CDB will first fetch VVN from VDB. For that, CDB sends `SELECT * FROM CVVN` to VDB, resulting in VDB starting to produce tuples, which are consumed by CDB to produce tuples that are finally consumed by HDB. HDB then processes the remaining operations and returns the results to the client.

As depicted in Figure 8, evaluating the XDB query has a trickle-down effect leading to a cascaded execution. The execution of our defined views enables pipeline-parallel execution of a cross-database query over multiple DBMSes. More specifically, as soon as the first DBMS (i.e., the one executing query fragment reference base tables) starts producing tuples (intermediate data), the dependent DBMS starts consuming tuples (via its foreign wrapper), and so on. Note that DBMS-to-

DBMS communication happens only through simple `SELECT * FROM <table>` queries. We achieve this simple communication pattern because of our previously described delegation phase that removes the complexity of sending complex inter-DBMS queries to fetch intermediate data. Overall, our delegation approach enables seamless communication between DBMSes and allows executing decentralized inter-DBMS pipelines without altering system components of underlying DBMSes.

VI. Evaluation

We now present our experimental study using data based on the TPC-H benchmark in the context of cross-database query processing. In particular, we compared XDB's performance with respect to overall runtime performance, data transfer cost, and data scalability by comparing it to (i) state-of-the-art federated query processing system Presto [26]; (ii) our implementation of the well-known Garlic approach [31]; and (iii) ScleraDB, which supports cross-database querying. We also studied the performance breakdown of XDB's overall runtime with respect to different query processing steps —i.e., logical optimization, plan annotation, plan-finalization and delegation, and finally execution. Overall, we found that:

- In terms of query execution time, XDB outperformed Presto by up to $6\times$, Garlic by up to $4\times$, and ScleraDB by up to $30\times$.
- XDB leads to less data transfer cost by up to 3 orders of magnitude when compared to baselines.
- XDB's execution time scales well for different data sizes.
- XDB's optimization and delegation phases lead to negligible (up to 10s) overhead compared to the execution time.

A. Experimental Setup

Cross-database Environment. We consider a distributed testbed with seven DBMSes. Our testbed consists of a multinode setup, where each physical node hosts one DBMS. We utilize PostgreSQL v12, Hive v3.1.2 (with Hadoop v3.1.4), and MariaDB v10.5, which we launch in containers (using Docker v20.10.8) on different physical nodes. Each node in our cluster is equipped with 2x Intel Xeon Silver 2.1GHz CPUs, 512GB main memory and a 4TB SSD running Ubuntu 20.04.2 LTS (kernel 5.4.0-26-generic x86_64). Nodes are interconnected through 1Gbit network interfaces.

Data & Cross-database Queries. We use data based on the TPC-H benchmark data [51], which we distribute among the seven DBMSes considering different table distribution (TD) as shown in Table III. For different experiments, we considered TPC-H data with scale factors (sf) 1, 10, 50, and 100. For cross-database queries, we consider the TPC-H queries Q3 (3 joins), Q5 (6 joins), Q7 (5 joins), Q8 (8 joins), Q9 (6 joins), and Q10 (4 joins). Our choice of queries is based number evaluating performance with respect to the number of joins (ranging from three to eight) in the cross-database queries.

Implementation & Baselines. We implemented XDB's core components in Java (JDK 8). We implement our DBMS connectors (DCs) for interacting with systems through DBMS-specific DDL statements using the latest JDBC drivers. We compare XDB with Trino v0.354 (a fork of PrestoSQL) [26], which is

TABLE III: Table Distributions with table abbrv., e.g., part:p.

	db1	db2	db3	db4	db5	db6	db7
TD1	l	c, o	s,n,r	p,ps	-	-	-
TD2	l,s	o,n,r	c	p,ps	-	-	-
TD3	l	o	s	ps	c	p	n,r

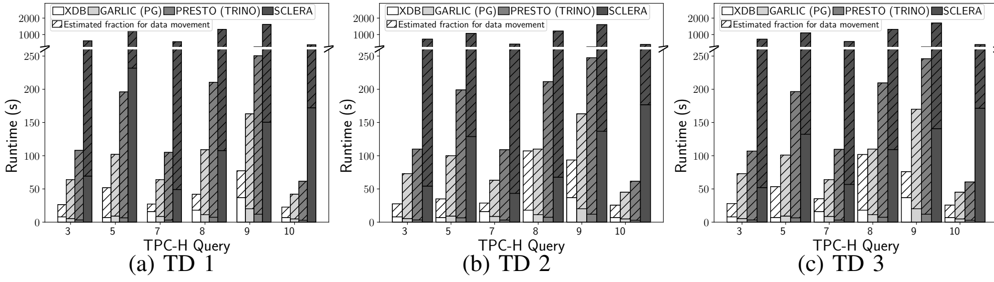
the state-of-the-art system for distributed query processing. We considered Presto with 2, 4, and 10 worker nodes. As Trino is a full-fledged distributed query engine and hence may impose performance overheads, we also consider a simple Garlic-like approach [15], where we used a PostgreSQL instance as a mediator, which connects to underlying systems using its SQL/MED capabilities. Additionally we considered ScleraDB [52] v4.0.

Methodology. We report the performance measure as total time elapsed between submitting a query and receiving the final result. For XDB, we break down this time into time taken by the optimization and delegation phases. We also report the transfer size as the total data transferred between systems during query execution as obtained from Docker's network statistics. For Garlic and Presto, we also report the estimated fraction of total time for fetching intermediate data to the mediator's execution engine. To do so, we measure the total time for "localized" tables, i.e. we preload intermediate results of individual subqueries into local tables and report the estimate as the difference in total time when considering remote and local tables. For all experiments, we report the average of 5 independent runs with exclusive access to the machines.

B. Overall Performance

In our first set of experiments, we compare XDB's overall performance with Presto (4 nodes) and Garlic. We considered TPC-H data (sf 10) based on all three table distributions (cf. Table III), and show the results results in Figures 9a–9c.

In-situ vs Mediator-based execution. We observe that cross-database query processing with XDB results in an improved query execution time (up to $4\times$) compared to Garlic, and (up to $6\times$) to Presto, (up to $30\times$) to ScleraDB. This performance improvement stems from our in-situ approach, which delegates entire query execution to underlying DBMSes. In Figures 9a–9c, the shaded region shows the (estimated) fraction of time (μ) that systems spend for data transfer. To derive this estimate for MW systems, we materialize intermediate results within the mediator, measure the "local" query execution time, and substract it from the total cross-database query execution time. For XDB we enforce its derived plan on a single DBMS and substract its runtime from the total time. For example, in these experiments, we observed that on average μ_{Garlic} was 80s and μ_{Presto} was 150s. Presto's overhead is more than Garlic's because Presto uses JDBC-connectors while our Garlic implementation leverages PostgreSQL's native transfer protocols. XDB also employs the native transfer protocols but does not suffer from this overhead as it decentralizes query execution by pipelinening operations across multiple DBMSes (recall Figure 8). ScleraDB also uses an "in-situ" cross-database querying approach, however, it pays a large performance penalty (up to $30\times$) as it moves all intermediate tables explicitly through its mediator (recall naive execution in Section V) and employs heuristics



(a) TD 1 (b) TD 2 (c) TD 3

Fig. 9: Overall runtime performance for different table distributions.

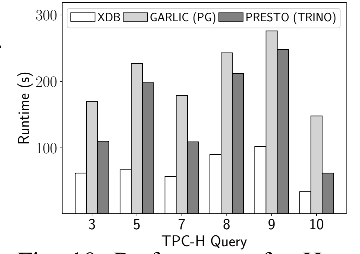


Fig. 10: Performance for Heterogeneous DBMSes (TD 1)

TABLE IV: Analysis of delegation plans for Queries Q3, Q5, and Q8 for TD 1 and TD 2 (with rounded row numbers).

TD	Q3		Q5		Q8	
	$t_i \xrightarrow{x} t_j$	#rows	$t_i \xrightarrow{x} t_j$	#rows	$t_i \xrightarrow{x} t_j$	#rows
TD1	db2:$\bowtie (c,o) \xrightarrow{i}$db1:$\bowtie (?,l)$	1,5M	db3:$\bowtie (\bowtie (n,r), s) \xrightarrow{i}$db1:$\bowtie (?, \bowtie (?,l))$	20K	db3: $\bowtie (n,r) \xrightarrow{i}$ db2: $\bowtie (\bowtie (?,c), o)$	5
			db2:$o \xrightarrow{e}$db1:$\bowtie (?, \bowtie (?,l))$	2,3M	db2: $\bowtie (\bowtie (?,c), o) \xrightarrow{i}$ db1: $\bowtie (\bowtie (l,?), ?)$	900K
			db1:$\bowtie (?, \bowtie (?,l)) \xrightarrow{e}$db2:$\bowtie (?,c)$	1,8M	db4:$p \xrightarrow{e}$ db1: $\bowtie (\bowtie (l,?), ?)$	15K
					db1: $\bowtie (\bowtie (l,?), ?) \xrightarrow{i}$ db3: $\bowtie (\bowtie (?,s), ?)$	24K
					db3:$n2 \xrightarrow{i}$ db3: $\bowtie (\bowtie (?,s), ?)$	25K
Σ		1,5M		4M		960K
TD2	db3:$c \xrightarrow{e}$db2:$\bowtie (?,o)$	300K	db2: $\bowtie (n,r) \xrightarrow{i}$ db1: $\bowtie (\bowtie (\bowtie (?,s), l), ?)$	5	db2: $\bowtie (n,r) \xrightarrow{i}$ db3: $\bowtie (c,?)$	5
	db2$\bowtie (?,o) \xrightarrow{i}$db1:$\bowtie (?,l)$	1,5M	db2:$o \xrightarrow{i}$ db1: $\bowtie (\bowtie (\bowtie (?,s), l), ?)$	2,3M	db3: $\bowtie (c,?) \xrightarrow{i}$ db2: $\bowtie (o,?)$	300K
			db1: $\bowtie (\bowtie (\bowtie (?,s), l), ?) \xrightarrow{e}$ db3: $\bowtie (?,c)$	1,8M	db2: $\bowtie (o,?) \xrightarrow{i}$ db1: $\bowtie (\bowtie (\bowtie (\bowtie (?,l), ?), s), ?)$	900K
					db4:$p \xrightarrow{e}$ db1: $\bowtie (\bowtie (\bowtie (\bowtie (?,l), ?), s), ?)$	15K
					db2:$n2 \xrightarrow{i}$ db1: $\bowtie (\bowtie (\bowtie (\bowtie (?,l), ?), s), ?)$	25
Σ		1,8M		4,1M		1,2M

to define the join operator placement. As ScleraDB performs almost an order of magnitude worse than other approaches, we do not consider it further in our performance evaluation.

We also evaluate XDB's performance by changing the underlying DBMSes in our table distribution (Table III) as follows: MariaDB for db2, Hive for db3, and PostgreSQL for all other dbs. We employed the available ODBC/JDBC SQL/MED implementations for the communication between the DBMSes. We compared XDB to a 4-worker-node Presto using sf 10 data. Our results in Figure 10 show a similar performance trend to the previous experiments (Figure 9a–9c). In this case, XDB outperforms Presto by a factor $\sim 2x$ on average. This is because XDB's performance also depends on the performance of the underlying systems with respect to cross-database joins. For example, MariaDB is not designed to be a high-performance OLAP DBMS while Hive is designed to handle data on a distributed file system but in this case, it operates on one node only. Yet, this experiment shows that our cross-DBMS approach, employed with out-of-the-box RDBMSes, outperforms a specialized distributed MW-based system.

Effect of table distribution. In these experiments, we also study how different table distributions (TD) affect cross-database query processing. As expected for Garlic and Presto, the execution plan remains unaffected. This is because query processing (after projection and selection pushdowns) is centralized within the mediator. For XDB, however, execution time is affected by different table distributions (recall from Section IV-B that while logical optimization is TD agnostic, plan annotation and finalization are TD dependent). For example, here Q3 took 21s for TD 1 and ~28s for TDs 2 and 3, Q5 took ~50s for TDs 1 and 3 and 35s for TD2, and Q8 took $40s$ for TD 1 whereas it took ~105s for TDs 2 and 3. To gain further insights into these results, we analyzed the delegation plan for each query (and TD) with respect to the number of tasks and number and type of inter-DBMS data movements. In particular, we computed for each $t_i \xrightarrow{x} t_j$ $x \in \{i, e\}$ the cardinality of relation $t_i.r$. Table IV shows these statistics for queries Q3, Q5, and Q8 for TD 1 and TD 2 (we omit other queries and TD 3 due to space constraints).

We observed that the change in execution time depends on the number, type, and amount of data movement. For instance, the delegation plan for Q3 involved 2 tasks and moving $\sim$1.4M rows via implicit movement for TD 1 (cf. Table III; table c and o are colocated). Whereas, it involved 3 tasks for TD 2 and TD 3 (not shown here) as a result of tables c, o, and l being on different DBMSes, which lead to additionally moving ~300K rows via explicit movement (highlighted under Q3 column in Table IV). This explains the slight increase in runtime for Q3 in TDs 2 and 3 compared to TD 1. In the case of Q5, although the delegation plans slightly differ in terms of number of tasks and type of movements, the number of intermediate rows moved remained the same for TD 1, and 3, while in TD 2 these are ~ 20K less. These differences are also reflected in Q5's execution times. Lastly, for Q8 the increase in the number of tasks and intermediate data had a more pronounced effect on its runtime for TD 1 in comparison to TD 2 and 3 (cf. column Q8; Table IV).

Scaling out the mediator. Furthermore, we also evaluate how XDB's decentralized execution (with inter-DBMS pipeline parallelism) compares to Presto's scale out capabilities. We used TD 1 and compared XDB's execution time to Presto (w/ 2, 4, and 10 worker nodes). The results are shown in Figure 11. We observed that scaling the number of workers did not improve the runtime performance of Presto. While the "actual" processing time of Presto improves by adding workers, its centralized (mediator-based) execution offsets its scale-out capabilities for cross-database query processing.

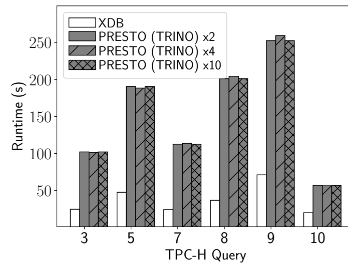


Fig. 11: Scaling Presto (TD 1)

Overall, we conclude that the in-situ query execution approach is more amenable for cross-database query processing.

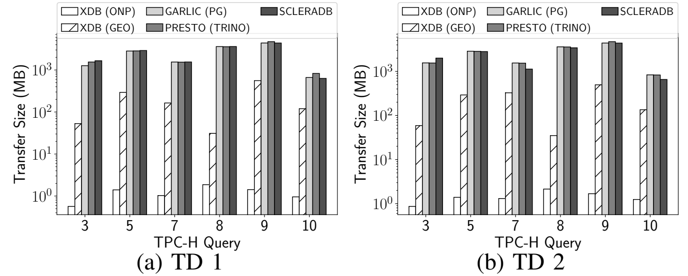
C. Data Transfer Cost

The amount of data transferred during query execution is an important factor with regard to the overall query performance and monetary cost, as current cloud vendors charge by the amount of incoming data for managed querying services (e.g., AWS Athena [35]). In these experiments, we evaluate XDB in terms of overall data transfer. We simulate two real-world scenarios where (1) DBMSes of an organization are located on-premise and (2) DBMSes are geo-distributed (e.g., in different data centers). We assume in both scenarios that XDB, Presto, and Garlic are located in a managed cloud environment. We considered TD 1 and TD 2 with sf 10, and all test queries. The results are shown in Figure 12. Here, XDB (ONP) and XDB (GEO) denote the on-premise and geo-distributed scenarios, respectively. Note that in both scenarios Presto and Garlic transfer the same amount of data over the network as data only moves from DBMSes to the mediator.

On-premise DBMSes. We observe that XDB (ONP) transfers only a minimal amount (up to ∼2MB) of data to the cloud compared to Presto and Garlic. This is because XDB enables underlying DBMSes to communicate directly, and hence avoids sending intermediate data to the cloud. The only data that is sent to the cloud is the final query result and lightweight control messages during query optimization and delegation. In contrast, Presto and Garlic fetch all intermediate data to the cloud to compute the final result (cf. Figures 4a and 4b). While Presto and Garlic push down selections, projections, and in some cases joins, to reduce the data transfer, we observe that a substantial amount of data is transferred to the cloud (up to ∼4.5GB for Q9 over all TDs).

Geo-distributed DBMSes. In the geo-distributed scenario, XDB (GEO) transferred less data than Presto and Garlic (up to 115× for Q8 and TD 1). We note that table distribution affects the amount of data transfer in XDB (GEO) as different TDs lead to different delegation plans. Yet, for our test queries, this amount was always less for the in-situ approach.

In sum, XDB transfers less data than MW approaches.



(a) TD 1
(b) TD 2

Fig. 12: Data transferred during query execution.

D. Data Scalability

In our next set of experiments, we evaluate XDB's scalability with respect to different data sizes. We evaluate the overall runtime performance considering TD 1 and data corresponding to TPC-H scale factors 1, 10, 50 and 100.

Individual queries. We consider three queries with different numbers of tables, namely Q3 (3 tables), Q9 (6 tables), and Q8 (8 tables) for which results are shown in Figures 13a–13c. For all scale factors, XDB outperformed Presto and Garlic (by up to 5× for Q8 sf 10). Furthermore, we observe that XDB's runtime increased for Q3 from 10s to 26s when scaling data from sf 1 to sf 10, and further increased to 179s for sf 50 and 374s for sf 100. This is expected as increasing the scale factor proportionally increases the intermediate data for a query. For example, for Q3 intermediate data increased to 53MB for sf 10 and to further 548MB for sf 100.

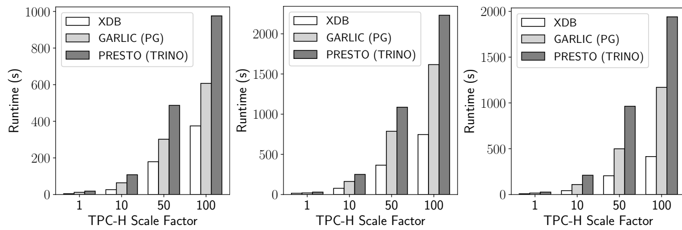
All queries. Additionally, we measured the average increase in runtime for all queries (see Figure 14). In comparison to Presto and Garlic, XDB led to average speed-ups of 4× and 3×, respectively for all scale factors. Also, the increase in runtime for XDB was proportional to the increase in the intermediate data transferred during execution, e.g. intermediate data increased from 120MB (sf 1) to ∼1.2GB (sf 10), and to ∼13GB (sf 100).

Overall, increases in runtime with respect to increase in data size is linearly proportional to intermediate query data.

E. Performance Breakdown

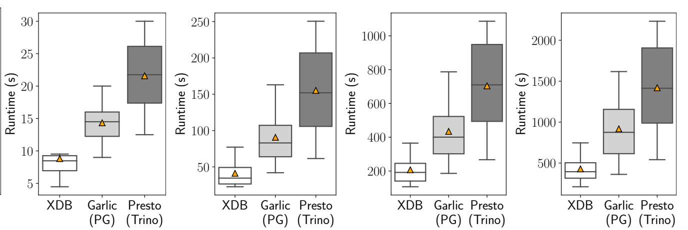
Next, we evaluate the performance breakdown for XDB's execution time. We breakdown XDB's query execution into four states: Preparation (`prep`) includes parsing and analyzing the query, and gathering metadata from underlying DBMSes; Logical optimization (`lopt`) derives an optimized logical plan; Plan annotation and finalization (`ann`) where XDB communicates with underlying DBMSes to derive the delegation plan; and finally Delegation and Execution (`exec`), where XDB delegates the plan and DBMSes execute the query. We show the performance breakdown in Figure 15 (note the log-scale y-axis).

We observe that the time spent in different phases depends on the query and the scale factor. For example, for Q3; TD 1 (Figure 15a), XDB spent 50% of the total execution time on preparing, annotating, and optimizing the query for sf 1, 12% for sf 10, and 2% for sf 100. This more or less generalizes to other queries as well. Overall, for all queries and scale factors (Figures 15a and 15b), the total time of the `prep`, `lopt`, and `ann` phases was always less than 10s. Moreover, for a large scale factor, the `exec` time offsets these phases. In general,



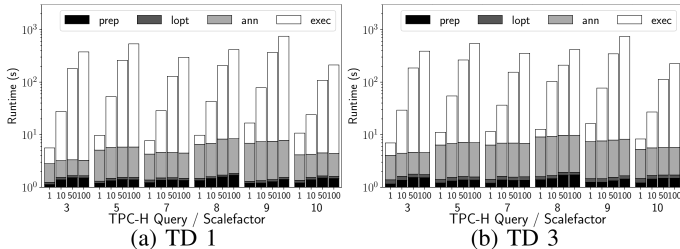
(a) Q3 (b) Q9 (c) Q8

Fig. 13: Runtime performance when scaling the data.



(a) Sf 1 (b) Sf 10 (c) Sf 50 (d) Sf 100

Fig. 14: Average runtime performance for data scaling.



(a) TD 1 (b) TD 3

Fig. 15: XDB query processing phase breakdown.

the increase in `prep` depends on the metadata operations of the underlying DBMSes, as query parsing and analysis remain the same for different sizes. The `lopt` time does not increase with the scale factor. This is because logical optimization only depends on the query and is independent of the scale factor. The `ann` phase is also independent of the scale factor but depends on the delegation plan. Furthermore, we observe that the `ann` time stays constant across queries and scale factors. This is because during optimization XDB's optimizer prunes the search space by only considering four options for a given cross-database operation (recall Section IV).

Based on our experiments, we conclude that our proposed techniques for in-situ cross-database processing lead to acceptable ($\leq 10s$*) overhead w.r.t. the total query execution time.*

VII. Additional Related Work

We now discuss related work on distributed query processing in addition to that discussed in Section II-B.

In-situ Query Processing. The term *in-situ* query processing has been mentioned in literature and often refers to processing of data without changing its format or location. QUIS [32] is a MW-based system for heterogeneous data processing. NoDB [33] proposes to process raw data in-situ, i.e., without first loading them into a DBMS. DaskDB [34] supports SQL together with data analytics operations without loading them into a DBMS and proposes a common runtime that handles both SQL and Python UDFs in-situ on raw files, similar to [53]. While the latter works refer to the term in-situ for processing raw files, XDB refers to the term for processing cross-db queries without requiring a mediating execution engine.

Polystores and Cross-Platform Systems. While XDB embraces DBMS heterogeineity in the realm of relational systems, Polystores [17]–[20] were proposed to tackle data model heterogeneity. However, they rely on rule-based methods for query optimization and employ the centralized MW paradigm to execute queries over disparate tables. Other approaches abstract data storage [54], [55] to provide a unified data model, but they still rely on existing systems for query processing. Approaches such as Rheem [21], [22] and Musketeer [56] propose to combine multiple platforms for a single workload. They, thus, employ optimization techniques for operator placement across different systems, but they do not focus on optimizations achieved through query rewrites.

Geo-Distributed Analytics. Beedkar et. al [57]–[59] consider a similar execution environment, but focus on integrating compliance aspects in query optimization. Wan-aware systems [60]–[63] propose approaches that aim at minimizing query latency and data transfer cost for MapReduce-like queries in cross-data center environments. In contrast, XDB's optimization goal and execution environment is different: it considers finding an optimal operator ordering, alongside an optimal movement operation and operator placement, and decentralized heterogeneous and autonomous DBMSes.

VIII. Conclusion

We proposed XDB, an efficient middleware system that uses a novel in-situ execution model for supporting cross-database query processing over existing DBMSes. XDB offloads all cross-database operations to DBMSes themselves, which allows to completely remove any central entity in query execution, thereby reducing data movements. In particular, we have proposed delegation plans – an abstraction that allows XDB to assign individual execution of tasks to the underlying DBMSes. For this, XDB employs a three-step optimization process to generate optimal delegation plans, which contain information about operator placement and data movement strategies. XDB is non-intrusive, as it rewrites delegation plans to a series of DDL operations leveraging the SQL/MED standard and views. Our evaluation showed that XDB is of up to $6\times$ faster than baseline mediator-wrapper systems and reduces the data movement across the network by up to 3 orders of magnitude. In future work, we plan to integrate XDB as Agora's [64] decentralized execution layer, and to investigate techniques that allow XDB to be part of composable DBMSes [65].

Acknowledgements

This work was funded by the German Ministry for Education & Research as BIFOLD - Berlin Institute for the Foundations of Learning & Data (01IS18025A; 01IS18037A; BIFOLD22B), and supported by Software Campus as PolyDB (01IS17052).

REFERENCES

[1] E. Commission, C. Directorate-General for Communications Networks, Technology, E. Scaria, A. Berghmans, M. Pont, C. Arnaut, and S. Leconte, *Study on data sharing between companies in Europe : final report*. Publications Office, 2018.

[2] https://www.mckinsey.de/publikationen/data-sharing-in-industrial-ecosystems, 2022.

[3] A. Simitsis, P. Vassiliadis, and T. Sellis, "Optimizing etl processes in data warehouses," in *21st International Conference on Data Engineering (ICDE'05)*. Ieee, 2005.

[4] D. Calvanese, G. De Giacomo, M. Lenzerini, D. Nardi, and R. Rosati, "Data integration in data warehousing," *International Journal of Cooperative Information Systems*, vol. 10, no. 03, 2001.

[5] M. Karpathiotakis, A. Floratou, F. Özcan, and A. Ailamaki, "No data left behind: real-time insights from a complex data ecosystem," in *Proceedings of the 2017 Symposium on Cloud Computing*, 2017.

[6] B. G. Lindsay, "A retrospective of r*: a distributed database management system," *Proceedings of the IEEE*, vol. 75, no. 5, 1987.

[7] P. A. Bernstein, N. Goodman, E. Wong, C. L. Reeve, and J. B. Rothnie Jr, "Query processing in a system for distributed databases (sdd-1)," *ACM Transactions on Database Systems (TODS)*, vol. 6, no. 4, 1981.

[8] J. C. Corbett, J. Dean, M. Epstein, A. Fikes, C. Frost, J. J. Furman, S. Ghemawat, A. Gubarev, C. Heiser, P. Hochschild *et al.*, "Spanner: Google's globally distributed database," *ACM Transactions on Computer Systems (TOCS)*, vol. 31, no. 3, 2013.

[9] R. Taft, I. Sharif, A. Matei, N. VanBenschoten, J. Lewis, T. Grieger, K. Niemi, A. Woods, A. Birzin, R. Poss *et al.*, "Cockroachdb: The resilient geo-distributed sql database," in *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*, 2020.

[10] U. Cubukcu, O. Erdogan, S. Pathak, S. Sannakkayala, and M. Slot, "Citus: Distributed postgresql for data-intensive applications," in *Proceedings of the 2021 International Conference on Management of Data*, 2021.

[11] A. Y. Halevy, Z. G. Ives, J. Madhavan, P. Mork, D. Suciu, and I. Tatarinov, "The piazza peer data management system," *IEEE Transactions on Knowledge and Data Engineering*, vol. 16, no. 7, 2004.

[12] R. Huebsch, J. M. Hellerstein, N. Lanham, B. T. Loo, S. Shenker, and I. Stoica, "Querying the internet with pier," in *Proceedings 2003 VLDB Conference*. Elsevier, 2003.

[13] P. Boncz and C. Treijtel, "Ambientdb: relational query processing in a p2p network," in *International Workshop on Databases, Information Systems, and Peer-to-Peer Computing*. Springer, 2003.

[14] S. Chawathe and et al., "The tsimmis project: Integration of heterogenous information sources," 1994.

[15] W. F. Cody, L. M. Haas, W. Niblack, M. Arya, M. J. Carey, R. Fagin, M. Flickner, D. Lee, D. Petkovic, P. M. Schwarz *et al.*, "Querying multimedia data from multiple repositories by content: the garlic project," in *Working Conference on Visual Database Systems*. Springer, 1995.

[16] J. Shute, R. Vingralek, B. Samwel, B. Handy, C. Whipkey, E. Rollins, M. Oancea, K. Littlefield, D. Menestrina, S. Ellner *et al.*, "F1: A distributed sql database that scales," 2013.

[17] J. Duggan and et al., "The bigdawg polystore system," *ACM Sigmod Record*, vol. 44, no. 2, 2015.

[18] J. Wang and et al., "The myria big data management and analytics system and cloud services." in *CIDR*, 2017.

[19] R. Alotaibi and et al., "Towards scalable hybrid stores: Constraint-based rewriting to the rescue," in *Proceedings of the 2019 International Conference on Management of Data*, 2019.

[20] B. Kolev and et al., "The cloudmdsql multistore system," in *Proceedings of the 2016 International Conference on Management of Data*, 2016.

[21] D. Agrawal, S. Chawla, B. Contreras-Rojas, A. Elmagarmid, Y. Idris, Z. Kaoudi, S. Kruse, J. Lucas, E. Mansour, M. Ouzzani *et al.*, "Rheem: enabling cross-platform data processing: may the big data be with you!" *Proceedings of the VLDB Endowment*, vol. 11, no. 11, 2018.

[22] S. Kruse, Z. Kaoudi, B. Contreras-Rojas, S. Chawla, F. Naumann, and J. Quiané-Ruiz, "RHEEMix in the data jungle: a cost-based optimizer for cross-platform systems," *VLDB J.*, vol. 29, no. 6, pp. 1287–1310, 2020.

[23] D. Kossmann, "The state of the art in distributed query processing," *ACM Computing Surveys (CSUR)*, vol. 32, no. 4, 2000.

[24] G. Wiederhold, "Intelligent integration of information," in *Proceedings of the 1993 ACM SIGMOD international conference on Management of data*, 1993.

[25] M. T. Özsu and P. Valduriez, *Principles of distributed database systems*. Springer, 1999, vol. 2.

[26] R. Sethi, M. Traverso, D. Sundstrom, D. Phillips, W. Xie, Y. Sun, N. Yegitbasi, H. Jin, E. Hwang, N. Shingte *et al.*, "Presto: Sql on everything," in *2019 IEEE 35th International Conference on Data Engineering (ICDE)*. IEEE, 2019.

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

[28] S. Melnik, A. Gubarev, J. J. Long, G. Romer, S. Shivakumar, M. Tolton, and T. Vassilakis, "Dremel: interactive analysis of web-scale datasets," *Proceedings of the VLDB Endowment*, vol. 3, no. 1-2, 2010.

[29] G. M. Essertel, R. Y. Tahboub, J. M. Decker, K. J. Brown, K. Olukotun, and T. Rompf, "Flare: Optimizing apache spark with native compilation for scale-up architectures and medium-size data." in *OSDI*, 2018, pp. 799–815.

[30] H. Gavriilidis, "Computation offloading in jvm-based dataflow engines," *BTW 2019–Workshopband*, 2019.

[31] M. J. Carey and et al., "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*. IEEE, 1995.

[32] J. Chamanara, B. König-Ries, and H. Jagadish, "Quis: in-situ heterogeneous data source querying," *Proceedings of the VLDB Endowment*, vol. 10, no. 12, pp. 1877–1880, 2017.

[33] I. Alagiannis, R. Borovica, M. Branco, S. Idreos, and A. Ailamaki, "Nodb: efficient query execution on raw data files," in *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data*, 2012, pp. 241–252.

[34] A. Watson, S. K. Das, and S. Ray, "Daskdb: Scalable data science with unified data analytics and in situ query processing," in *2021 IEEE 8th International Conference on Data Science and Advanced Analytics (DSAA)*. IEEE, 2021, pp. 1–10.

[35] https://aws.amazon.com/athena/pricing, 2022.

[36] M. Lenzerini, "Data integration: A theoretical perspective," in *Proceedings of the twenty-first ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems*, 2002.

[37] J. Melton, J. E. Michels, V. Josifovski, K. Kulkarni, and P. Schwarz, "Sql/med: a status report," *ACM SIGMOD Record*, vol. 31, no. 3, 2002.

[38] https://www.postgresql.org/docs/current/ddl-foreign-data.html, 2022.

[39] https://dev.mysql.com/doc/refman/5.6/en/federated-storage-engine.html, 2022.

[40] J. Camacho-Rodríguez, A. Chauhan, A. Gates, E. Koifman, O. O'Malley, V. Garg, Z. Haindrich, S. Shelukhin, P. Jayachandran, S. Seth *et al.*, "Apache hive: From mapreduce to enterprise-grade big data warehousing," in *Proceedings of the 2019 International Conference on Management of Data*, 2019.

[41] https://github.com/exasol/virtual-schemas, 2022.

[42] https://clickhouse.com/docs/en/engines/table-engines/special/external-data/, 2022.

[43] https://www.ibm.com/docs/en/db2/11.5?topic=federation, 2022.

[44] P. G. Selinger, M. M. Astrahan, D. D. Chamberlin, R. A. Lorie, and T. G. Price, "Access path selection in a relational database management system," in *Readings in Artificial Intelligence and Databases*. Elsevier, 1989.

[45] L. M. Haas, J. C. Freytag, G. M. Lohman, and H. Pirahesh, "Extensible query processing in starburst," in *Proceedings of the 1989 ACM SIGMOD international conference on Management of data*, 1989.

[46] M. T. Roth, F. Ozcan, and L. M. Haas, "Cost models DO matter: Providing cost information for diverse data sources in a federated system," in *VLDB*. Morgan Kaufmann, 1999.

[47] Q. Zhu and P.-A. Larson, "A query sampling method for estimating local cost parameters in a multidatabase system," in *Proceedings of 1994 IEEE 10th International Conference on Data Engineering*. IEEE, 1994.

[48] Q. Zhu and P.-å. Larson, "Solving local cost estimation problem for global query optimization in multidatabase systems," *Distributed and parallel databases*, vol. 6, no. 4, 1998.

[49] A. Rahal, Q. Zhu, and P.-Å. Larson, "Evolutionary techniques for updating query cost models in a dynamic multidatabase environment," *The VLDB journal*, vol. 13, no. 2, 2004.

[50] V. Giannakouris, "Building learned federated query optimizers," in *Proceedings of the VLDB 2022 PhD Workshop co-located with the 48th International Conference on Very Large Databases (VLDB 2022), Sydney, Australia, September 5, 2022*, ser. CEUR Workshop Proceedings, Z. Bao and T. K. Sellis, Eds., vol. 3186. CEUR-WS.org, 2022. [Online]. Available: http://ceur-ws.org/Vol-3186/paper_5.pdf

[51] http://www.tpc.org/tpch/, 2022.
[52] https://github.com/scleradb/sclera, 2022.
[53] P. M. Grulich, S. Zeuch, and V. Markl, “Babelfish: Efficient execution of polyglot queries,” *Proceedings of the VLDB Endowment*, vol. 15, no. 2, pp. 196–210, 2021.
[54] A. Jindal, J. Quiané-Ruiz, and J. Dittrich, “WWHow! Freeing Data Storage from Cages,” in *Conference on Innovative Data Systems Research, CIDR*, 2013.
[55] P. Cudré-Mauroux, E. Wu, and S. Madden, “The Case for RodentStore: An Adaptive, Declarative Storage System,” in *Conference on Innovative Data Systems Research, CIDR*. www.cidrdb.org, 2009. [Online]. Available: http://www-db.cs.wisc.edu/cidr/cidr2009/Paper_97.pdf
[56] I. Gog and et al., “Musketeer: all for one, one for all in data processing systems,” in *Proceedings of the Tenth European Conference on Computer Systems*, 2015.
[57] K. Beedkar, J. Quiané-Ruiz, and V. Markl, “Compliant geo-distributed query processing,” in *SIGMOD '21: International Conference on Management of Data, Virtual Event, China, June 20-25, 2021*, G. Li, Z. Li, S. Idreos, and D. Srivastava, Eds. ACM, 2021, pp. 181–193. [Online]. Available: https://doi.org/10.1145/3448016.3453687
[58] K. Beedkar, D. Brekardin, J. Quiané-Ruiz, and V. Markl, “Compliant geo-distributed data processing in action,” *Proc. VLDB Endow.*, vol. 14, no. 12, pp. 2843–2846, 2021. [Online]. Available: http://www.vldb.org/pvldb/vol14/p2843-beedkar.pdf
[59] K. Beedkar, J. Quiané-Ruiz, and V. Markl, “Navigating compliance with data transfers in federated data processing,” *IEEE Data Eng. Bull.*, vol. 45, no. 1, pp. 50–61, 2022. [Online]. Available: http://sites.computer.org/debull/A22mar/p50.pdf
[60] A. Vulimiri, C. Curino, P. B. Godfrey, T. Jungblut, J. Padhye, and G. Varghese, “Global analytics in the face of bandwidth and regulatory constraints,” in *12th USENIX Symposium on Networked Systems Design and Implementation (NSDI 15)*, 2015.
[61] Q. Pu, G. Ananthanarayanan, P. Bodik, S. Kandula, A. Akella, P. Bahl, and I. Stoica, “Low latency geo-distributed data analytics,” *ACM SIGCOMM Computer Communication Review*, vol. 45, no. 4, pp. 421–434, 2015.
[62] K. Kloudas, M. Mamede, N. Preguiça, and R. Rodrigues, “Pixida: optimizing data parallel jobs in wide-area data analytics,” *Proceedings of the VLDB Endowment*, vol. 9, no. 2, pp. 72–83, 2015.
[63] R. Viswanathan, G. Ananthanarayanan, and A. Akella, “Clarinet: Wan-aware optimization for analytics queries,” in *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*, 2016.
[64] J. Traub, Z. Kaoudi, J.-A. Quiané-Ruiz, and V. Markl, “Agora: Bringing together datasets, algorithms, models and more in a unified ecosystem [vision],” *ACM SIGMOD Record*, vol. 49, no. 4, 2021.
[65] H. Gavriilidis, L. Behme, S. Papadopoulos, S. Bortoli, J. Quiané-Ruiz, and V. Markl, “Towards a modular data management system framework,” in *1st International Workshop on Composable Data Management Systems @ VLDB*, S. R. Valluri and M. Zaït, Eds., Sydney, Australia, 2022.