Compliant Geo-distributed Query Processing

Kaustubh Beedkar
TU Berlin

Jorge-Arnulfo Quiané-Ruiz
TU Berlin, DFKI

Volker Markl
TU Berlin, DFKI

ABSTRACT

In this paper, we address the problem of compliant geo-distributed query processing. In particular, we focus on dataflow policies that impose restrictions on movement of data across geographical or institutional borders. Traditional ways to distributed query processing do not consider such restrictions and therefore in geo-distributed environments may lead to non-compliant query execution plans. For example, an execution plan for a query over data sources from Europe, North America, and Asia, which may otherwise be optimal, may not comply with dataflow policies as a result of shipping some restricted (intermediate) data. We pose this problem of compliance in the setting of geo-distributed query processing. We propose a compliance-based query optimizer that takes into account dataflow policies, which are declaratively specified using our policy expressions, to generate compliant geo-distributed execution plans. Our experimental study using a geo-distributed adaptation of the TPC-H benchmark data indicates that our optimization techniques are effective in generating efficient compliant plans and incur low overhead on top of traditional query optimizers.

ACM Reference Format:
Kaustubh Beedkar, Jorge-Arnulfo Quiané-Ruiz, and Volker Markl. 2021. Compliant Geo-distributed Query Processing. In *Proceedings of the 2021 International Conference on Management of Data (SIGMOD '21), June 18–27, 2021, Virtual Event, China.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/XXXXXX.XXXXXX

1 INTRODUCTION

Today's globalized world has widened the landscape of data analytics applications. Large organizations accommodate several databases and IT infrastructure at various sites[1]. As a result, organizations require to perform data analytics across different locations [27, 46, 58, 62]. Supporting geo-distributed data analytics (a.k.a. wide-area big data) in a unified manner is crucial for an organization's day-to-day operations, including back-office tasks and informing data-driven decisions.

In such geo-distributed settings, most works have mainly focused on expanding the scope of query processing frameworks to support sites at different physical locations [31, 32, 45, 56, 59]. Strategies typically involve distributing query operators (like join or aggregation) across sites based on several performance metrics, such as latency and bandwidth. For example, executing a two-way join query over data sources from Asia, Europe, and North America may be faster when performing the first join in Europe and the second one in Asia than the other way around.

Still, an important aspect of geo-distributed data analytics, which has not been considered yet, is cross-border dataflow restrictions [7, 11]. Dataflow restrictions arise from regulations or policies controlling the movement of data across borders. Such dataflow restrictions may pertain to personal data (e.g., personally identifiable information), business data (e.g., sales and financial), or data considered "important" to a country or organization. For example, processing data generated by autonomous cars in three different geographies, such as Europe, North America, and Asia may face different dataflow regulations: there may be legal requirements that only aggregated data may be shipped out of Europe, no data at all may be shipped out of Asia, and only a part of data may be shipped from North America to Europe. Thus, it is crucial to consider compliance with respect to dataflow constraints derived from these regulations during query processing. In particular, when generating and executing query execution plans, one must not violate any dataflow policy. We refer to this kind of distributed query processing as *compliant geo-distributed query processing*.

Although the concept of compliant query processing has been studied in the literature, it has been for notions different from the one we consider in this paper. Earlier work has focused on Hippocratic databases [2, 3], fine grained access control [8, 47, 61], and secure/privacy-preserving query processing [17, 36, 42, 57]. All these works are complimentary to our work: We investigate how to support *compliance with respect to data movement* during query processing, which to our knowledge, has not yet been studied.

Supporting compliant geo-distributed query processing entails two major research challenges. First, we need an easy (thus declarative) way to specify dataflow constraints. Doing so is not trivial because dataflow constraints may pertain to different types of data as well as its processing. For example, restrictions may apply to an entire dataset, parts of it, or even to information derived from it. Second, we have to find efficient ways to process queries in a manner that they are compliant with respect to dataflow constraints. In contrast to cost-based query optimization techniques, which focus solely on performance aspects, we need efficient ways to include compliance aspects in query optimization and processing.

In this paper, we present our initial efforts towards compliant geo-distributed query processing. We propose *policy expressions* as a simple way to specify dataflow constraints. Our policy expressions are SQL-like statements that concisely specify which data can legally be shipped to other sites in a declarative way. We also show how one can effectively and efficiently consider dataflow policies when generating query execution plans. Especially, we show how one can integrate a compliance-based query optimizer into the Volcano optimizer framework [25]. As a result, one can easily incorporate our solution into existing query processing frameworks.

[1] These sites can be at different organizations or geographical (national or international) locations. For clarity, we refer to both kinds of sites as geo-distributed sites.

SIGMOD '21, June 18–27, 2021, Virtual Event, China

ACM ISBN 978-1-4503-8343-1/21/06.
https://doi.org/10.1145/XXXXXX.XXXXXX

In summary, after further motivating the need for compliant query processing (Section 2), we present our major contributions:

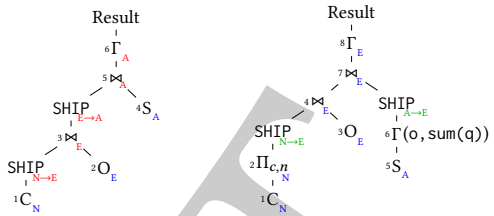
- We describe the foundations of compliant query processing: We formalize the concepts of cross-border dataflow policies and compliant query execution plans, which in turn allow us to formally define the problem of compliant query processing. (Section 3)
- We introduce policy expressions, and propose to use SQL-like statements, to specify which data can be legally shipped to other sites in a concise and declarative way. Using SQL-like statements offers the flexibility of "masking", i.e., making source data compliant for shipping to another site based on the dataflow constraints defined by regulations and policies. (Section 4)
- We propose a policy evaluator, which allows for easy integration of policy expressions into a compliance-based query optimizer. In particular, it determines to which cross-borders sites the output of operators can be shipped. (Section 5)
- We show how to use the Volcano optimizer generator to implement a compliance-based query optimizer. In particular, we enumerate the plan space by applying algebraic equivalence rules in a top-down fashion and filter compliant ones in a bottom-up fashion. To do so, we treat geo-locations as "interesting properties" associated with query plan operators and propose rules based on the structure of subplans to filter compliant plans. This allows us to determine whether or not a query can have a compliant execution plan w.r.t. the imposed restrictions. (Section 6)
- We experimentally evaluate the effectiveness and efficiency our compliance-based query optimizer using a geo-distributed setup based on the TPC-H schema. Overall, the results show that our optimization techniques are effective in generating compliant execution plans. They also show that our techniques incur a low overhead (in order of milliseconds) on top of traditional cost-based optimization approaches. (Section 7)

2 MOTIVATING EXAMPLE

Imagine a transnational company, CarCo, a manufacturer of cars headquartered in Europe. Assume that CarCo has several offices across Europe, a manufacturing unit with suppliers in Asia, and a subsidiary that overlooks sales operations in North America (for its American customers). Following the first quarter, the operations team in CarCo wants to analyze its financial data by integrating it with the sales data from North America as well as with the data from its suppliers in Asia. Note that this geo-distributed scenario is similar to the query processing pipelines reported by Microsoft [59], Facebook [54], Twitter [39], LinkedIn [4], and BigBench [23].

In this scenario, CarCo relies on a distributed DBMS (DDBMS) for its geo-distributed IT infrastructure. The DDBMS provides a query interface to CarCo for analyzing its geo-distributed data. Typically, the DDBMS transparently translates a user specified query into a query execution plan (QEP). To do so, the query optimizer of the DDBMS extends single site optimization across distributed databases. The optimizer considers communication costs between computing nodes (sites, for short) and introduces a global property that describes where the processing of each plan operator happens.

In the above geo-distributed environment, typical strategies to process a query [12, 45, 59] — that involves transferring intermediate results — may not comply with data movement regulations.



(a) Non-compliant (b) Compliant

Figure 1: Example execution plans for Q_{ex}.

European directives, for example, may regulate transferring only certain information fields (or combinations of information fields), such as non-personal information or information not relatable to a person. Likewise, regulations in Asia, too, may impose restrictions on data transfer. To illustrate, assume that CarCo's DDBMS has three databases D_N, D_E, and D_A located in North America (**N**), Europe (**E**), and Asia (**A**) respectively. D_N stores information about customers, D_E stores information about orders, and D_A stores the supply information. Consider the following schema

Customer	(**c**ustkey, **n**ame, **a**cctbal, **m**ktseg, **r**egion)	**N**
Orders	(**c**ustkey, **o**rdkey, **t**otprice)	**E**
Supply	(**o**rdkey, **q**uantity, **e**xtprice)	**A**,

and a query Q_{ex}, which is given as

```
SELECT C.name, SUM(O.totprice), SUM(S.quantity)
FROM Customer AS C, Orders AS O, Supply AS S
WHERE C.custkey=O.custkey AND O.ordkey=S.ordkey
GROUP BY C.name
```

Furthermore, based on recent studies on data movement regulations [1, 7, 11], consider dataflow policies $\mathscr{P}_N$, $\mathscr{P}_E$, and $\mathscr{P}_A$ that applies to data from North America, Europe, and Asia respectively:

$\mathscr{P}_N$ Customer data from North America can be shipped outside only after suppressing account balance information.

$\mathscr{P}_E$ Only aggregated Orders data from Europe can be shipped to Asia and an order's price cannot be shipped to North America.

$\mathscr{P}_A$ Only aggregated Supply data for orders' quantity and extended price from Asia can be shipped to Europe.

Figure 1 illustrates two QEPs for Q_{ex}. Here, the SHIP operator describes the point where intermediate results are communicated between two sites and Γ denotes the aggregation operator. For brevity, we suppress some attributes and use the first letter of tables', locations', and attributes' names. We will use the number beside each operator for referring to it in the text. Assume now the QEP in Figure 1(a) is more efficient than the QEP in Figure 1(b). In this case CarCo's DDBMS, which uses cost-based query optimization strategies, most likely will output the QEP in Figure 1(a). However, this plan is non-compliant: its SHIP operators violate dataflow policies $\mathscr{P}_N$ ($\text{SHIP}_{N\to E}$ ships Customer table without suppressing the account balance) and $\mathscr{P}_E$ ($\text{SHIP}_{E\to A}$ ships non-aggregated Order information to Asia). In contrast, the QEP in Figure1(b) is compliant: It performs both join operations in Europe and masks Customer and Supply data before shipping them to Europe; Masking via projection (operator 2) suppresses the account balance information of Customers and via aggregation suppresses the orders' quantity.

Our goal in this work is twofold. First, we enable users to specify dataflow policies in a simple but effective way. Second, we devise a query optimizer that (i) determines if a query is *legal*, i.e., it has at least one compliant QEP, with respect to dataflow policies, and (ii) translates a legal query into a compliant QEP.

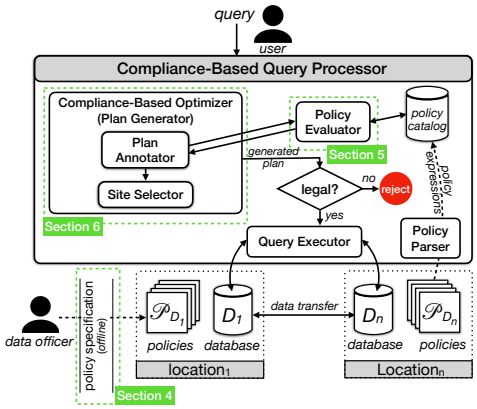


Figure 2: Overview of Compliant Query Processing.

3 COMPLIANT QUERY PROCESSING

Let us start by overviewing our query processing framework. We consider a distributed SQL database that is composed of geo-distributed databases. Each database D is tied to a location l. For simplicity, we assume that each location houses only one database. We use the notation D_l to denote database D at location l. We further consider that, in each location, D is stored using the relational model (i.e., tables, columns, and rows) and each node offers a gateway to access its database.[2] Without loss of generality, we assume that the geo-distributed schema is the union of all local schemas.

Figure 2 illustrates the overall architecture of our compliant query processing framework. A data officer at each location reflects her data movement policies using a declarative data specification. The system then stores these policies in a policy catalog for query optimization. Note that this policy specification process is an offline process. At querying time, the compliance-based query optimizer uses this policy catalog (via the policy evaluator), when enumerating plans, to validate if they are compliant with their input dataflow policies. The optimizer uses this validation mechanism to know when a final *query execution plan* (QEP) is violating an existing dataflow policy. If so, it rejects executing the query. Only when the resulting QEP is compliant, it proceeds with the query execution.

Our compliant query processing framework relies on two core parts: (i) the specification of dataflow policies associated with each data location, and (ii) the query optimization process under such policies. In the remainder of this section, we first formalize the concepts of dataflow policy and compliant query plan and then define the compliant query processing problem.

3.1 Dataflow Policies

A *cross-border dataflow policy* describes the restrictions on transfer of data across organizational and/or geographical borders [11]. In general, a dataflow policy specifies *which* information as well as *how* and *to where* this information can be legally transferred. In our framework, for each database D, we consider a dataflow policy $\mathscr{P}_D$, which is modeled as a set of tuples, $\mathscr{P}_D = \{ \langle \mathcal{D}, L_{\mathcal{D}} \rangle \}$, where $\mathcal{D}$ specifies the information in D that can be legally shipped to locations $L_{\mathcal{D}}$. This model allows us to specify different data transfer restrictions (on the same data) for different locations. For example, depending on a location, $\mathcal{D}$ can specify an entire table, some rows and columns, or some derived information (e.g., aggregates).

A crucial aspect in adhering to dataflow policies is to mask data in a way that renders it suitable to be transferred across borders. For this, we propose *policy expressions* (Section 4) as a simple way to express dataflow policies. The idea is to treat these policy expressions as first-class citizens during query optimization, to translate queries into compliant QEPs. To this end, we propose a policy evaluation algorithm (the policy evaluator in Figure 2; Section 5). Abstractly, the policy evaluation algorithm, denoted by $\mathcal{A}$, receives the schema of database D, a query Q, and a policy $\mathscr{P}_D$, and outputs a set $\mathcal{A}(Q, D, \mathscr{P}_D)$ of locations to which the result of $Q(D)$ can be legally shipped.[3] Recall the `Customer` database from Section 2. For $Q = \Pi_{c,n}(C)$, we have $\mathcal{A}(Q, D_N, \mathscr{P}_N) = \{ N, A, E \}$, and for $Q = \Pi_n(\sigma_{a=100}(C))$ we have $\mathcal{A}(Q, D_N, \mathscr{P}_N) = \{ N \}$.

3.2 Compliant Query Plan

A QEP is said to be a *compliant QEP* if it does not violate any of the dataflow policies, i.e., it does not transfer any intermediate data to a location that violates a policy. We further ensure that the resulting compliant QEP retains the query semantics, i.e., the output of the query should be the same if there were no dataflow policies. For example, Figure 1(b) illustrates a compliant QEP for our running query example Q_{ex}. Compared to the non-compliant QEP in Figure 1(a), the compliant plan implements: a projection operator, $\Pi_{c,n}$, suppressing the account balance information as required by policy $\mathscr{P}_N$; an aggregate operator, $\Gamma(o, sum(q))$, to adhere to policy $\mathscr{P}_A$, and; `SHIP` operators that do not violate any dataflow policy when transferring intermediate data.

We now formalize the notion of a compliant QEP. Without loss of generality, we denote a QEP by a directed graph $\mathcal{Q} = (\mathcal{O}, E)$, where $\mathcal{O}$ is the set of operator nodes and E denotes the set of edges as dataflow between operators. Moreover, let Q_o denote the (sub)query corresponding to a tree with operator o as its root node and let l_o describe the location where the processing of an operator o happens. Consider the QEP in Figure 1(b). For the (projection) operator o_2 we have $Q_{o_2} = \Pi_{c,n}(C)$ and $l_{o_2} = \text{N}$. For two operators $o, o' \in \mathcal{O}$, we denote by $o \rightarrow o'$, if the output of operator o is directly consumed by operator o'. We further denote by $\rightarrow^*$ the reflexive transitive closure of $\rightarrow$. Continuing our example, we have $o_1 \rightarrow o_2$ and $o_1 \rightarrow^* o_8$. For each operator $o \in \mathcal{O}$, let $\texttt{in}(o) = \{ o' \mid o' \rightarrow o \}$, $\texttt{ins}(o) = \{ o' \mid o' \rightarrow^* o \}$, and $\texttt{out}(o) = o' : o \rightarrow o'$. For instance in Figure 1(b), we have $\texttt{in}(o_4) = \{ o_2, o_3 \}$, $\texttt{ins}(o_4) = \{ o_1, o_2, o_3 \}$, and $\texttt{out}(o_4) = \{ o_7 \}$. Finally, for a *non-leaf* operator o denoted by

$$U_o = \{o' \mid o' \in \texttt{ins}(o) \text{ and } l_{o'} = l_{o''} \; \forall o'' \in \texttt{ins}(o') \cup \{ o' \} \text{ and}$$

$$l_{\texttt{out}(o')} \neq l_{o''} \; \forall o'' \in \texttt{ins}(\texttt{out}(o')) \cup \{ \texttt{out}(o') \}\}$$

the set of its descendant operators where $Q_{o'}$ is a subquery pertaining to a single database with all its operators processed at location $l_{o'}$. As example, in Figure 1(a), we have $U_{o_3} = \{ o_1, o_2 \}$ and in Figure 1(b), we have $U_{o_7} = \{ o_2, o_3, o_6 \}$.

For $\mathcal{Q} = (\mathcal{O}, E)$ to be compliant, we want that for all $o \in \mathcal{O}$, its execution at location l_o does not violate any dataflow poli Definition 1 formally captures this.

[2] Such an architecture is common in modern distributed data management systems, e.g., CockroachDB [10, 53] and Apache Drill [15].

[3] For notational simplicity, we also use the notation D to denote the schema in the context of algorithm $\mathcal{A}$.

Definition 1. *A query execution plan $\mathcal{Q} = (\mathcal{O}, E)$ is compliant if for all operators $o \in \mathcal{O}$ one of below conditions is satisfied:*

$$l_o = l \quad \texttt{in}(o) = \emptyset \text{ and } l \text{ is the table's source location} \tag{c1}$$

$$l_o \in \bigcap_{o' \in U_o} \mathcal{A}(Q_{o'}, D_{l_{o'}}, \mathscr{P}_{l_{o'}}) \tag{c2}$$

In the above definition, Condition c1 captures the fact that the execution of a leaf (tablescan) operator is always compliant at the table's location. Condition c2 ensures that any data that flows into a non-leaf operator o at location l_o must be compliant with respect to the dataflow policies associated with its data sources. For example, in QEP of Figure 1(b): leaf operators o_1, o_2, and o_3 satisfy Condition c1; o_2 satisfies Condition c2 as $N \in \mathcal{A}(C, D_N, \mathscr{P}_N) = \{N\}$; o_4 satisfies Condition c2 as $E \in \mathcal{A}(\Pi_{c,n}(C), D_N, \mathscr{P}_N) \cap \mathcal{A}(O, D_E, \mathscr{P}_E) = \{N, A, E\} \cap \{E\}$; and so on.

3.3 Compliant Query Processing Problem

Having defined a compliant QEP, we can formally define the compliant query processing problem that we focus on as follows:

Problem Statement. *Given a query Q over geo-distributed databases $D_1, D_2, \ldots, D_n$, where each database D is associated with a set of dataflow policies $\mathscr{P}_D$, find an optimal QEP $\mathcal{Q}$ that is compliant.*

Discussion. Our goal is to find the best possible execution plan with respect to a given cost model that is compliant with dataflow policies. In this paper, we consider traditional cost models (see Section 6) that determine total query execution cost. Our methods, however, are general in that they can also be adapted to other cost models (e.g., that determine query response time).

4 POLICY SPECIFICATION

We now discuss how a user (e.g., a data information officer) can specify dataflow policies in a convenient way. Recall that a dataflow policy is a set of tuples $\mathscr{P}_D = \{\langle \mathcal{D}, L_{\mathcal{D}} \rangle\}$, where $\mathcal{D}$ specifies the data in D that can be transferred to locations $L_{\mathcal{D}}$. We propose policy expressions as a simple and intuitive way to specify $\mathscr{P}_D$.

Scope. A crucial aspect in adhering to dataflow policies is to first transform the data via masking functions that renders it suitable to another location, i.e., suitable for a SHIP operator. In this paper, we confine to masking via relational operations (e.g., project, aggregate, or filter) that preserve the query semantics. For instance, a projection operator can mask certain columns by projecting them out before being consumed by a SHIP operator. We basically define two kinds of policy expressions: *simple* and *aggregate* expressions. Generally speaking, a simple expression is of the form of a Select-Project query that can specify restrctions pertaining to certain table, rows and/or columns. An aggregate expression is of the form of a Select-Project-GroupBy query and further allows specifyng restrictions pertaining to aggregated information. We detail these two of expressions in the following two subsections.

Disclosure Model. We focus on *which* and *where* data are allowed to be shipped. We propose attribute-based specification to express policies and follow a conservative approach. In other words, we assume that by default no attributes are allowed to be shipped anywhere unless specified otherwise by the policy. We note that in some cases negative instances, i.e., specifying what is not allowed, may be more convenient. This can be handled by an additional preprocessing step under a closed world assumption.

4.1 Simple Expressions

We first look at simple expressions that allow for shipping certain rows and columns of a table to another location without violating any dataflow policy. We define the syntax of a simple expression as a Select-Project (SP) query:

```
ship attribute list from table to location list
where condition list
```

This expression specifies cells, i.e., rows and columns, of a table to be shipped without affecting the query semantics.[4] The specified cells from the table in the **from** clause (i) belong to both columns in the **ship** clause and tuples that satisfy the predicates in the **where** clause, and (ii) can be shipped to locations in the **to** clause. Intuitively, if a subquery accesses only the specified cells, then its output can be shipped to locations specified in the expression.

Example 1. *Consider policy $\mathscr{P}_N$ from Section 2, which does not allow for shipping the account balance information of customers outside North America. Suppose the policy also allowed for shipping customer's mktsegment and region information to Europe for* CarCo*'s commercial customers. We can use the following policy expressions:*

```
ship custkey,name from Customer C to Asia, Europe

ship mktseg, region from Customer C to Europe
where mktseg='commercial'
```

The first expression specifies the cells corresponding to the custkey and name column of tuples in Customer table. The second expression specifies the cells corresponding to the mktseg and region columns of the tuples in the Customer table that satisfy the predicate $p \equiv mktseg = \text{`}commercial\text{'}$. Given these expressions, a query that accesses only the name column of the Customer table is legal with respect to Asia and Europe. Similarly, a query that additionally accesses the mktseg and region columns is only legal with respect to Europe if the specified cells satisfy predicate p. For example, the output of the query $\Pi_{c,n}\left(\sigma_{n \text{ LIKE } \text{`}A\%\text{'}}(C)\right)$ can be shipped to all locations, the output of $\Pi_{c,n,r}\left(\sigma_{n \text{ LIKE } \text{`}A\%\text{'}}(C)\right)$ cannot be shipped outside of North America, and output of $\Pi_{c,n,r}\left(\sigma_{n \text{ LIKE } \text{`}A\%\text{'} \wedge mktseg=\text{`}commercial\text{'}}(C)\right)$ must only be shipped to Europe.

4.2 Aggregate Expressions

Although simple expressions are sufficient to express a large variety of dataflow policies, there are policies that allow for shipping of aggregate information only. For these cases, we introduce aggregate expressions that allow us to specify aggregations over columns. Similar to simple expressions, aggregate expressions do not affect the query semantics. The syntax of an aggregate expression is similar to a Select-Project-GroupBy (SPG) query and is given as:

```
ship attribute list as aggregates aggregate types
from table to location list where condition list
group by attribute list
```

In the above syntax, the list of attributes in the **ship** clause specifies cells of columns that should be aggregated before being shipped to locations in the location list. The **as aggregate** clause specifies aggregation functions that should be used to aggregate

[4] For exposition, we restrict to expressions over a single table. This is not a limitation: one can specify a policy expression over more than one base table. In this case, the condition list in the **where** clause of the expression must contain the join predicate.

specified cells. As before, the specified cells must belong to columns in the attribute list for the tuples that satisfy the predicate in its **where** clause. Lastly, the **group by** clause specifies lists of grouping attributes for which the specified cells can be grouped by zero, one or more attributes from its attribute list.

Example 2. *Consider again the Customer table from Section 2 and assume that account balance information can be shipped only after aggregating. A possible expression is:*

```
ship acctbal as aggregates sum, avg from Customer C
to * group by mktseg, region
```

The above expression specifies how values of the acctbal column of the Customer table can be shipped outside. In particular, it specifies that (i) acctbal should be aggregated via the functions SUM or AVG and (ii) the cells of the acctbal column can be grouped by mktsegment and/or by nationkey. For example, output of the queries $\mathcal{G}_{sum(acctbal)}(C)$ *and* $_{region}\mathcal{G}_{avg(acctbal)}(C)$ *can be shipped to all locations, whereas of* $\mathcal{G}_{sum(acctbal)}\left(\sigma_{name='abc'}(C)\right)$ *and* $\Pi_{acctbal}(C)$ *cannot be shipped at all.*

5 POLICY EVALUATION

We now turn our attention to evaluating dataflow policies. In what follows, we assume that we are given: a query q; the schema of database D that contains tables referenced by q, and; the set of policy expressions $\mathscr{P}$ applicable to data in D. Given this input, the goal of policy evaluation algorithm $\mathcal{A}$ is to output the list $\mathcal{A}(q, D, \mathscr{P})$ of locations to which the query's output $q(D)$ can be shipped to.

Notations. Before delving into our policy evaluation algorithm, $\mathcal{A}$, we first introduce some necessary notations. Let A_q denote the set of attributes that appear in the output expressions of a query q, and P_q denote the query predicate. When q is an aggregate query, we further denote by G_q the set of attributes that appear in its `group by` clause and by f_a the aggregate function associated with attribute $a \in A_q \setminus G_q$.[5] Likewise for a policy expression e, we denote by A_e the set of its `ship` attributes, by L_e the set of its `to` locations, and by P_e its predicate. For aggregate expressions, we extend the notation and denote by G_e the set of its `group by` attributes and by F_e the set of its aggregate operations specified in `as aggregates` clause.

Evaluation Process. Algorithm 1 summarizes our policy evaluation algorithm. As an example, consider the policy expressions e_1–e_4 (left column of Table 1) that apply to a relation $T(A, B, C, D, E, F)$. At a high level, for a given query, the algorithm goes over all expressions and determines the set of legal locations in an attribute-wise fashion, i.e., from `ship` attributes of all the expressions that appear in the output expression of the query. In more detail, we associate, with each attribute a in the output expression of the query, a set L_a of locations to which it can be legally shipped (line 1; Algorithm 1). We then look for policy expressions that have `ship` attributes appearing in A_q (line 2). Next, we check if the rows specified by the query predicate are also specified by the predicate of each expression. That is, we check the logical implication $P_q \implies P_e$ (line 3). If the test passes, we then consider three cases:

(1) Selection Query & Simple Expression. In case of a selection query and a simple expression (lines 4–5), we simply associate

Algorithm 1 Policy evaluation algorithm.

Require: D, $\mathscr{P}$, q **Ensure:** set $\mathcal{A}(q, D, \mathscr{P})$ of legal locations

1: **for all** $a \in A_q$ **do** $L_a \leftarrow \emptyset$
2: **for all** Expression $e \in \mathscr{P}$ **such that** $A_q \cap A_e \neq \emptyset$ **do**
3: **if** $P_q \implies P_e$ **then**
4: **if** e is a simple expression **then**
5: **for all** $a \in A_q \cap A_e$ **do** $L_a \leftarrow L_a \cup L_e$
6: **else if** q is an aggregation query **then**
7: **if** $G_q \subseteq G_e$ **then** // *includes empty subset*
8: **for all** $a \in A_q \cap (A_e \cup G_e)$ **do**
9: **if** $a \in G_e$ **or** $(a \in A_e \wedge f_a \in F_e)$ **then**
10: $L_a \leftarrow L_a \cup L_e$
11: $\mathcal{A}(q, D, \mathscr{P}) \leftarrow \bigcap_{a \in A_q} L_a$

the locations in the expression's `to` clause with each of the `ship` attributes that appear in the query's output expression. For example, consider query q_1 from Table 1. Columns L_A, L_C, and L_D (below q_1) show locations associated with output attributes A, C, and D of q_1, resp. For example, after processing e_2, we have $L_A = \{ l_1, l_2, l_3, l_4 \}$.

(2) Aggregate Query & Simple Expression. In case of an aggregate query and a simple expression, we proceed as in the first case simply because the expression's `ship` attributes are "less" aggregated than that in the query. For example, consider the aggregate query q_2 shown on top of Table 1 (right). We associate locations l_2, and l_3 with attribute C after processing e_1.

(3) Aggregate Query & Aggregate Expression. In case of an aggregate query and an aggregate expression (lines 6–10), we first check whether the `group by` attributes in the query are a subset of the `group by` attributes of the expression.[6] This check is necessary because: (i) the grouping attributes are implicitly allowed to be shipped for the specified `ship` attributes, and (ii) the `ship` attributes can only be grouped by the grouping attributes. Then, for attributes in the expression's `group by` clause that also appear in the query's `group by` clause, we associate the locations in the expression's `to` clause. For `ship` attributes that appear in the query's output expression, we additionally check if they are aggregated via an operation that is allowed by the expression. If so, we assign expression's `to` locations to those attributes. For example, consider the aggregate query q_2 shown on top of Table 1 (right). After processing the expression e_4, we associate locations l_1 and l_2 with attributes C, F, and G: As an example, the last three columns in Table 1 show locations associated with output attributes of q_2.

Once we have processed all expressions, we check if the set L_a of associated locations for all output attributes $a \in A_q$ are non-empty. In such a case, we return their intersection as the set of locations that are legal w.r.t. q. For example, the output of query q_1 in Table 1 can be shipped to location l_3 and of query q_2 to locations l_1 and l_2.

Discussion. We note the completeness of our policy evaluator depends on the logical implication test (line 3 in Algorithm 1). In this work, we use a simple, yet effective technique similar to that described in [24]. This technique is sound but incomplete in some cases. For instance, the implication test fails for $P_q \equiv (A = 5 \wedge B = 3)$ and $P_e \equiv A + B = 8$. Still, the problem of testing logical implications is orthogonal to the problem we focus on in this paper.

[5] For notational convenience, we assume that an attribute is associated with only one aggregate function.

[6] This includes empty subset for cases when query has aggregation over entire column.

Expression e	$q_1 \equiv \Pi_{A,C,D}\,(\sigma_{B>15}(T))$			$q_2 \equiv \ {}_{C}\mathscr{G}_{\text{sum}(F*\,(1-G))}\,(T)$		
	L_A	L_C	L_D	L_C	L_F	L_G
$e_1 \equiv$ ship A, B, C from T to l_2, l_3	$\{ l_2, l_3 \}$	$\{ l_2, l_3 \}$	-	$\{ l_2, l_3 \}$	-	-
$e_2 \equiv$ ship A, B from T to l_1, l_2, l_3, l_4	$\{ l_1, l_2, l_3, l_4 \}$	$\{ l_2, l_3 \}$	-	-	-	-
$e_3 \equiv$ ship A, D from T to l_1 l_3 where B > 10	$\{ l_1, l_2, l_3, l_4 \}$	$\{ l_2, l_3 \}$	$\{ l_1, l_3 \}$	-	-	-
$e_4 \equiv$ ship F, G as aggregates sum, avg from T to l_1, l_2 group by E, C	$\{ l_1, l_2, l_3, l_4 \}$	$\{ l_2, l_3 \}$	$\{ l_1, l_3 \}$	$\{ l_1, l_2, l_3 \}$	$\{ l_1, l_2 \}$	$\{ l_1, l_2 \}$
$\mathcal{A}(q_i, D, \mathscr{P})$	$\{ l_3 \}$			$\{ l_1, l_2, l_3 \}$		

Table 1: Illustration of policy evaluation algorithm.

6 COMPLIANT QUERY OPTIMIZATION

We now describe our compliance-based optimizer for distributed query processing. We follow the two-phase optimization process, which is well studied in the distributed query optimization literature [9, 12, 21, 30, 35]. At a high level, in the first phase, the algorithm finds an optimal plan, in terms of join orders and join methods, using a cost model, but ignoring data shipping cost. This phase assumes that all tables are stored locally (as in centralized query optimization). In the second phase, the algorithm finalizes the previously found optimal plan by selecting sites (taking into account data transfer and scheduling costs) to execute the QEP.

Yet, to produce a compliant QEP, we have to adapt this two-phase optimization process. We essentially need to select sites for executing plan operators such that all conditions in Definition 1 are satisfied. However, this is far from being trivial. To see its non-triviality, recall the example from Section 2 and the two query plans in Figure 1. Suppose the first phase produces the QEP in Figure 1(a) (ignore the `SHIP` operators for now). In this case, it is impossible to decide an optimal sites selection such that dataflow policies are not violated. For example, in Figure 1(a), any site where operator o_3 is executed will violate either $\mathscr{P}_N$ or $\mathscr{P}_E$ or both. On the other hand, suppose now that the first phase produces a plan as shown in Figure 1(b) (ignoring the `SHIP` operators), we can indeed find a compliant QEP. The challenge resides in enabling the first phase to produce a plan such that in the second phase an optimal site selection always leads to the best possible[7] compliant plan.

Solution Overview. We use the Volcano optimizer generator [25] to tackle the above challenge (Section 6.1): We generate a top-down optimizer that, for a given logical expression and a physical property vector, produces the best possible QEP . Figure 3 illustrates how we use the Volcano optimizer generator to build such an optimizer, which we refer to as *plan annotator* (Section 6.2). Overall, the plan annotator receives a logical plan as input and outputs an annotated (physical) QEP. Our plan annotator uses the traditional cost model to determine query execution cost (assuming that all tables are stored locally) in which cost functions are based on input cardinalities.[8] This annotated QEP, along with join order and join methods, specifies a set of compliant sites (as annotations) for each operator in the plan. This annotation process represents the first phase of our two-phase optimization process. The second phase is composed of the *site selector* component (Section 6.3), which finalizes the location of each plan operator using dynamic programming, to produce a compliant QEP.

[7] In the search space of explored plans and with respect to the cost model.

[8] As we use the Volcano optimizer generator, one can use other cost-models. In this paper, we focus on how to adapt existing cost functions.

6.1 A Volcano-Based Optimizer

Let us first detail how we adapt the Volcano optimizer generator for geo-distributed settings. Figure 3 depicts this adaptation using beige and blue boxes. In particular, we perform the following adaptations: (i) We introduce new abstract logical properties for annotating operators (*properties* box); (ii) We adapt each physical operator's cost function, which "forces" the generated optimizer to chose plans with operators that are annotated (*cost functions* box), and; (iii) We introduce a set of rules that allow us to determine annotations in a way that Conditions c1 and c2 are met during plan enumeration (*annotation rules* box). Below, we explain each of these.

Properties. We introduce execution and shipping traits, which are associated with each plan operator. While an *execution trait* is a logical property that describes where an operator can be legally executed, a *shipping trait* describes where the output of an operator can be legally shipped. More formally, for an operator node n, we denote its execution trait with set $\mathcal{E}_n$ and its shipping trait with set $\mathcal{S}_n$. For example, Figure 4 shows a simplified search space of alternative plans for the example query Q_{ex}. Here, project operator (node 3) has an execution trait $\mathcal{E}_3 = \{ N \}$, which means it can be legally executed in North America. Likewise, it has shipping trait $\mathcal{S}_4 = \{ N, E \}$ (see node 4 on top of the projection), which means the output after the projection operator can legally be shipped to North America and Europe. Note that based on dataflow policies, operator nodes may have zero, one, or multiple execution and shipping traits.

Annotation Rules. They enable the optimizer to derive execution and shipping traits for plan operators during the optimization process. These rules work in a similar fashion as algebraic transformation rules. The optimizer via its rule engine matches an operator node with an annotation rule to derive its shipping and execution traits. We introduce four annotation rules:

AR 1: *A leaf operator node n (i.e., tablescan operator) is associated with an execution trait $\mathcal{E}_n = \{ l \}$, where l is the table's source location.* The rule is based on Condition c1 and the observation that a `tablescan` can always be legally executed where the table is.

AR 2: *An operator node n is associated with an execution trait $\mathcal{E}_n = \mathcal{E}_n \cup \left\{ l \mid l \in \cap_{n' \in \text{in}(n)} \mathcal{S}_{n'} \right\}$.* This rule enforces that an operator can be executed at a location l only if all of its inputs can be legally shipped to l. For example, joining data from EU and Asia in US can only be legal when both data can be legally shipped to US.

AR 3: *An operator node n is associated with a shipping trait $\mathcal{S}_n = \mathcal{S}_n \cup \mathcal{E}_n$.* This rule is based on the observation that an operator's output can always be legally shipped to the location where it can be legally executed. For example, the output of a projection in North America can be legally shipped to North America.

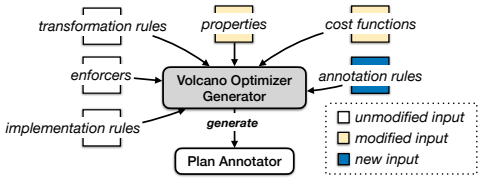


Figure 3: Building a compliance-based optimizer.

AR 4: *A operator node n is associated with a shipping trait* $\mathcal{S}_n = \mathcal{S}_n \cup \{l \mid l \in \mathcal{A}(Q_n, D, \mathscr{P}_D),\ Q_n$ is a local query over $D\}$. This rule derives shipping traits based on dataflow policies. It is crucial for satisfying Condition c2 for cross-database operations. In contrast to other rules, which are based on the expression tree's structure, this rule invokes the policy evaluation on query subexpressions pertaining to a single data source.

Compliance-Based Cost Function. Whenever a plan is explored using transformation rules or enforcers, the optimizer invokes an operator's cost function and uses its estimated cost for pruning the search space. We extend these cost functions to consider an operator's execution trait, to prevent the optimizer from discarding a (sub)plan in which operators are not yet annotated. In essence, the cost function evaluates an operator's cost to be infinite if $\mathcal{E}_n = \emptyset$.

6.2 Plan Annotator – Optimization Phase 1

As discussed earlier, we use the Volcano optimizer generator, with our adaptations discussed above, to generate the plan annotator. The latter receives as input a logical plan and a compliance-based optimization goal to output an annotated QEP.

Compliance-Based Optimization Goal. It is worth noting that the goal of the plan annotator is to produce an annotated plan. To do so, we set the optimization goal as a physical plan property vector that specifies the requirement of a non-empty shipping trait, along with desired physical properties (e.g., sort order). As we derive the execution and shipping traits in a bottom-up fashion (recall Rules 1–4), associating a non-empty shipping trait with the root operator is crucial: It allows the optimizer to consider a plan only when all of its operators are annotated.

Optimization Process. We now illustrate the optimization process of the plan annotator using our running example. Given a logical expression and the compliance-based optimization goal, the plan annotator starts with the initial expression tree. For example, Figure 4 shows the initial expression tree for query Q_{ex} using dotted lines. Starting with an initial expression tree, the plan annotator repeatedly applies rules via Volcano's rule engine to enumerate the search space of QEPs. The white boxes in Figure 3 illustrate these rules, which are (i) algebraic equivalence rules (e.g., filter pushdown and join commutativity), (ii) implementation rules for converting logical operator nodes to physical operator nodes (e.g., converting logical join to hash join), and (iii) enforcer rules that can "enforce" the input to have certain physical properties (e.g., sort order). Figure 4 also illustrates a simplified search space of QEPs for Q_{ex}:[9] we use numbers to refer to specific nodes in the text below.

Whenever a new subexpression is generated, the optimizer (via its rule engine) also applies the annotation rules for each new plan operator to derive its execution and shipping traits. For example, in Figure 4: Node 1 (tablescan operator; `Customer` table) has execution trait $\mathcal{E}_1 = \{N\}$ as the `Customer` table is located in North America (Rule 1) and has shipping trait $\mathcal{S}_2 = \{N\}$ (rule 3); Node 3 has execution trait $\mathcal{E}_3 = \{N\}$ (rule 2); Node 4 has shipping trait $\mathcal{S}_4 = \{N, E\}$ because for its relational expression $q = \Pi_{n,c}(C)$, we have $\mathcal{A}(q, D_N, \mathscr{P}_N) = \{E\}$ (rule 4, and as a result of rule 2). Node 5 has execution trait $\mathcal{E}_5 = \{E\}$ as both its inputs nodes 4 and 6 have N in their shipping trait (i.e., $\{N, E\} \cap \{E\}$; rule 2). Here, the projection operator is "masking" the account balance as desired by policy $\mathscr{P}_N$. In a similar fashion, the root node 7 has shipping trait $\mathcal{S}_7 = \{E\}$.

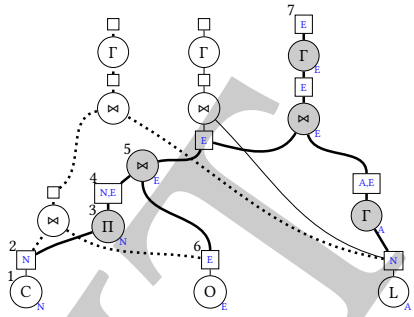


Figure 4: Simplified search space for Q_{ex}.

Overall, our plan annotator piggybacks on Volcano's search engine and cost-based pruning to determine an annotated plan with least cost. Our "adapted" cost functions ensure that the optimizer produces such a plan. It only outputs a plan when all its nodes are annotated; otherwise, it rejects the query. In our running example, Q_{ex} is legal with respect to dataflow policies and we can produce the results legally in Europe. We show such an optimal annotated plan using solid lines in Figure 4.

6.3 Site Selector – Optimization Phase 2

Given an annotated QEP, the site selector proceeds with the placement of operator nodes. In the worst case, this can lead to an exponential number of the possible compliant plans. For example, an annotated plan with m operator nodes each with $|\mathcal{E}| = k$ leads to m^k possibilities. Thus, instead of taking a simple greedy approach or an exhaustive enumeration, the site selector uses dynamic programming: It employs a memoized, recursive top-down approach.

Algorithm 2 shows this approach for site selection. The function CostOf(n, l) (lines 3–18) computes $C_{n,l}$ as the minimum cost of executing node n at location l. When n is a table scan operator, we have $C_{n,l} = 0$ for $l =$ table's location and otherwise ∞ (line 7). For all other operators it computes the cost by recursively computing the minimum cost of its inputs plus the data shipping cost[10] (lines 9–15). While computing $C_{n,l}$ for all nodes and their set of compliant locations, the algorithm keeps track of C^*_n as the best cost for node n and updates its optimal execution location l_n accordingly (line 17). In our running example, we show the best plan from step one using thick solid lines and gray nodes in Figure 4. It is easy to see that this leads to the plan shown in Figure 1(b) as a compliant QEP.

We note that our two-phase approach — like other variants [12, 35] — has less complexity but may lead to plans that have a higher communication cost. This is because the cost functions, in the first phase, ignore the location of data and hence join ordering might impact communication costs.

[9] For brevity, we do not show equivalence nodes in Figure 4.

[10] We use the message cost model [12] for computing the data shipping cost; see Sec. 7.

Algorithm 2 Site selection.

Require: Annotated plan **Ensure:** Compliant QEP

```
1: CostOf(⊤, l_⊤)                                   // ⊤ denotes a root node
2:
3: CostOf(n, l)
4: C_{n,l} ← LookUp(n, l)
5: if C_{n,l} not in lookup table then
6:    if in(n) = 0 then                              // Base case
7:       if l = location of basetable then C_{n,l} ← 0 else C_{n,l} ← ∞
8:    else                                           // recursively compute cost of inputs
9:       C_{n,l} ← 0
10:      for all n' ∈ in(n) do
11:         C'_l ← ∞                                // min cost of an input w.r.t. a location
12:         for all l' ∈ E_{n'} do
13:            C_l ← ShipCost(n', l', l) + CostOf(n', l')
14:            if C_l < C'_l then C'_l ← C_l
15:      C_{n,l} ← C_{n,l} + C'_l
16: Add C_{n,l} to lookup table                     // Update best cost C*_n for operator
17: if C_{n,l} < C*_n then C*_n ← C_{n,l};   l_n ← l          // n and select site l
18: return C_{n,l}
```

6.4 Correctness

We remark that our compliance-based query optimizer is *sound* in that it will never output a QEP that is not compliant. The following theorem captures its soundness.

Theorem 1. *The compliance-based optimizer never outputs a non-compliant query execution plan.*

Proof. We provide the proof by contradiction. Let us assume that the optimizer outputs a non-compliant plan $\mathcal{Q} = (\mathcal{O}, E)$. Then,

$$\exists o \in \mathcal{O} \text{ such that } l_o \notin \cap_{o' \in U_o} \mathcal{A}(Q_{o'}, D_{l_{o'}}, \mathscr{P}_{l_{o'}})$$

As sites are selected based on execution traits; Alg. 2, line 12

$$\implies \mathcal{E}_o \bigcap \cap_{o' \in U_o} \mathcal{A}(Q_{o'}, D_{l_{o'}}, \mathscr{P}_{l_{o'}}) = \emptyset$$

As traits are derived bottom-up; Rules 2 and 3

$$\implies \mathcal{E}_o \bigcap \mathcal{A}(Q_{o'}, D_{l_{o'}}, \mathscr{P}_{l_{o'}}) = \emptyset$$
$$\implies \mathcal{S}'_o \bigcap \mathcal{A}(Q_{o'}, D_{l_{o'}}, \mathscr{P}_{l_{o'}}) = \emptyset \text{ ,which contradicts Rule 4} \quad \square$$

Last, note that our approach may however be *incomplete*: in some cases the optimizer may fail to find a compliant QEP, i.e, it may safely but incorrectly reject a legal query. This is not inherent to our adaptations to the cost-based optimization, but it relies on transformation rules provided to the Volcano optimizer generator. For instance, consider the example query of Section 2 and its (simplified) search space of plans shown in Figure 4. Without an algebraic transformational rule that pushes an aggregation past a join, the plan annotator will not output an annotated plan because the compliance-based optimization goal will not be met. This is because the root (node 7; Figure 4) will have an empty shipping trait (similar to the other two plans shown in Figure 4), and thus the optimizer will reject the query.

In our approach, we rely on existing relational algebraic equivalence and query rewrite rules, which are sufficient for the dataflow policies that can be expressed by our policy expressions. Note that, completeness also relies on the completeness of the policy evaluation algorithm $\mathcal{A}$ (i.e., when logical implication test fails; recall Discussion in Section 5).

Location	DB	Tables
$L1$	db-1	Customer, Orders
$L2$	db-2	Supplier, Partsupp
$L3$	db-3	Part
$L4$	db-4	Lineitem
$L5$	db-5	Nation, Region

Table 2: TPC-H table distribution among five locations.

7 EXPERIMENTAL EVALUATION

We evaluated our optimization framework with the goal of investigating four main aspects: (i) its effectiveness in generating compliant execution plans for a large variety of queries and policy expressions, (ii) its overhead when compared to traditional cost-based optimization, (iii) quality of plans, and (iv) its scalability w.r.t. the number of policy expressions and locations. We found that:

- Our query optimizer was effective in generating compliant plans for all queries. In contrast to that, the cost-based optimization in 50–70% of the cases produced a non-compliant plan.
- The overhead (in terms of optimization time) of our compliant query optimizer was in the order of milliseconds when compared to the traditional cost-based approach.
- Our approach produced the same plans as the ones produced by the traditional cost-based approach whenever the later produced a compliant plan.
- For cases when the traditional cost-based approach produced a non-compliant plan, executing the compliant plan incurs an overhead that solely depends on the query and dataflow policies.
- The scalability of our optimizer is linearly proportional to the number of policy expressions that affect a query's search space.
- The optimization time increased roughly linearly w.r.t. number of locations.

7.1 Experimental Setup

Database. We consider a geo-distributed database consisting of TPC-H tables, which (following [12]) is distributed among five local databases as shown in Tab. 2. TPC-H data was generated using a scale factor of 10. Note that scale factor does not impact the query optimization. Furthermore, we consider the TPC-H schema as the global schema and used simple GAV mappings [13] to map tables in the global schema to tables in the local schema.[11]

Query Workload. As the overhead (i.e., the increase in the optimization time) incurred by our compliant query optimizer mainly depends on the number of joins in a given query, we consider six representative TPC-H queries in our experiments: Q_2, Q_3, Q_5, Q_8, Q_9, and Q_{10}. These queries have different complexities in terms of # joins (j) in their QEP. Q3 and Q10 have low complexity with $j = 2$ and $j = 3$, resp.; Q5 and Q9 have medium complexity with each having $j = 5$; and Q2 and Q8 have high complexity with $j = 13$ and $j = 7$, resp. We additionally consider 400 ad-hoc queries. We implemented a query generator to get these ad-hoc queries. Our query generator creates an ad-hoc query by randomly selecting a table and joining in additional tables using the PK-FK relationship. It chooses joining tables in a way that they span over two or more locations. It then randomly selects output columns and

[11] GAV mappings allows us to also specify tables that are distributed across locations; we consider such a setup in Sec. 7.5.

Policy Expression (e)
$e_1 \equiv$ `ship * from db-5.nation to *`
$e_2 \equiv$ `ship * from db-5.region to *`
$e_3 \equiv$ `ship partkey, suppkey, supplycost from db-2.partsupp to` L_3, L_4
$e_4 \equiv$ `ship partkey, mfgr, size, type, name from db-3.part to` L_4 `where size > 40 OR type LIKE '%COPPER%'`
$e_5 \equiv$ `ship extendedprice, discount as aggregates sum from db-4.lineitem to` L_1 `group by suppkey, orderkey`

Table 3: Snippet of expressions based on TPC-H data.

generates query predicates. For aggregation queries, it randomly chooses grouping as well as aggregation attributes. It does so with the help of a property file that is similar to the one used by our policy expression generator (see below). 55% of the queries references two tables, 35% referenced three, and 10% referenced four tables. About 30% of queries were aggregation queries. Each query on an average selected four columns as output columns and had 3–4 predicates (excluding join predicates) on average. We ran each of our considered queries seven times and report the average.

Policy Expressions. We implemented a policy expression generator, which takes as input a (local) database schema, a property file, an expression template (see below), and available geo-locations to consider. It then instantiates the template by querying the database. The property file specifies, among others, which attributes can be aggregated or/and serve as grouping key, which tables can be joined, and which range predicates can be imposed. In particular, we generate four sets of expressions using the following templates:

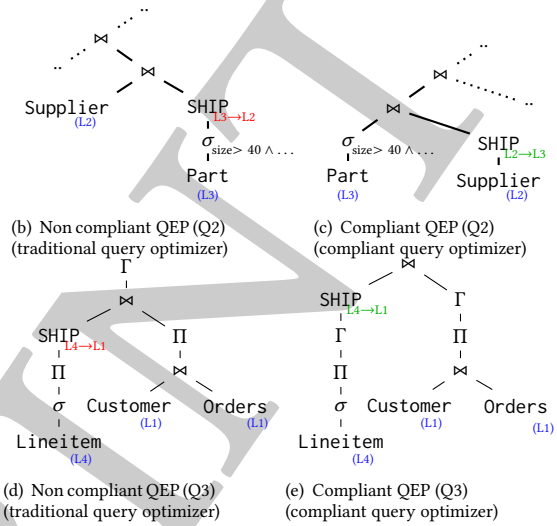
(1) **T** – These expressions specify restrictions on the entire table: `ship * from [table] to [locations]`.
(2) **C** – These expressions restrict the shipment of certain columns: `ship [attributes] from [table] to [locations]`.
(3) **CR** – This set of expressions is similar to set C but it additionally specify restrictions on certain rows: `ship [attributes] from [table] to [locations] where [condition]`.
(4) **CR+A** – Finally, the last set of expressions extends set CR by additionally specifying expressions that restrict shipping to only aggregates and include expressions: `ship [attributes] as aggregates [aggregate functions] from [table] to [locations] where [condition] group by [attributes]`.

Tab. 3 shows a snippet of expressions that we used in our experiments. It is worth noting that all policy expressions are of a form that there always exists at least one compliant QEP for each query.

Implementation. We implement our compliance-based query optimization framework in Java (JDK 1.8). For generating the plan annotator (Section 6.2), we use the Apache Calcite (v1.2.3) library [6]. Calcite provides an extensible query optimizer based on Volcano/-Cascades framework. Calcite allowed us to implement our shipping and execution traits, annotation rules, policy evaluation algorithm, cost model, and modifications to the optimization goal. We store the policy expressions in an in-memory data structure. We also compare our compliant query optimization with traditional cost-based query optimization. For the latter (i.e., the baseline), we used Calcite's cost-based optimizer as-is for the first phase in the two-phase optimization. For the second phase, we use our site selector algorithm by considering all locations to be legal. We ran all experiments on a machine equipped with two Intel i7-7560U CPUs and 16GB of main memory running Ubuntu 16.04.

Expr. set	#Expr.	Q2	Q3	Q5	Q8	Q9	Q10
T	8	NC	C	C	C	C	C
C	10	NC	C	C	C	C	C
CR	10	NC	NC	C	C	C	NC
CRA	10	NC	NC	C	C	C	NC

(a) QEPs produced by the traditional query optimizer



(b) Non compliant QEP (Q2) (traditional query optimizer)
(c) Compliant QEP (Q2) (compliant query optimizer)
(d) Non compliant QEP (Q3) (traditional query optimizer)
(e) Compliant QEP (Q3) (compliant query optimizer)

Figure 5: (a) Traditional query optimizer's failures: although it can yield to compliant (C) QEPs, it also produces non-compliant (NC) QEPs in several cases. (b)–(e) Excerpts of QEPs for TPC-H Q2 and Q3.

7.2 Effectiveness

We first study the effectiveness of our compliance-based optimization framework. Our main goal is to evaluate both (i) how effective our optimizer is for generating compliant QEPs, and (ii) how does our optimizer compare to traditional cost-based optimizers.

Results for TPC-H Queries. We report the results for all 6 TPC-H queries for each of our policy expressions set: T, C, CR, CR+A. This results into 24 variants, each comprising of a query and a set of policy expressions. Set T consists of eight policy expressions (cf. e_1; Tab. 3) and all other sets consists of ten policy expressions each.

Fig. 5 shows the results of these experiments. We observed that for all 24 variants, our approach was successful in producing a compliant QEP. This is not the case for the traditional cost-based optimization approach, which could not produced a compliant QEP for 8 queries as shown in Fig. 5(a). These results clearly show the effectiveness of our adaptations proposed in Section 6.1. We further illustrate this using excerpts of the QEPs for TPC-H queries Q2 and Q3 when considering the policy expressions in sets CR and CR+A, resp. In case of Q2 (see Fig. 5(b)), the traditional query optimizer produced a non-compliant QEP because the $\texttt{SHIP}_{L_3 \rightarrow L_2}$ operation contradicts expression e_4 (see Tab. 3). Our optimizer, in contrast, was able to yield a compliant QEP plan (Fig. 5(c)) by shipping the supplier table to location $L3$ instead. In case of Q3 (Fig. 5(d)), our optimizer enforces push down optimizations for aggregates (5(e)) whenever a policy expression specifies so (like e_5 as shown in Tab. 3) and it is possible without affecting the query semantics. This is in contrast to the traditional optimizer, which cannot enforce such push down optimizations (see 5(e)). Our approach can do

so as it takes execution traits into account when computing the cost of an operator. Recall that an operator with empty execution traits has infinite cost, which forces our optimizer to explore more alternatives. We also observed that our compliant query optimizer injects projections before a SHIP operator (as in example QEP of Fig. 1(b)) whenever attributes are restricted by a policy expression and are not needed by the query later.

Results for Ad-Hoc Queries. We also considered the 400 ad-hoc queries, which were equally divided among four groups. For each group we consider one of the four sets of policy expressions. Each set of policy expressions consists of 50 expressions, except for set T which consists of 8 expressions. Fig. 6(a) shows the fraction of ad-hoc queries for which a compliant QEP was obtained under different types of expressions. We observe that our approach successfully finds a compliant QEP for all 400 queries. In contrast, the cost-based approach produced a compliant QEP for ~50% of the the queries on average. For instance, it finds a compliant QEP for 42% of the queries when using the expressions in set T. This was more pronounced when using the expressions in set CR+A: it produced a compliant QEP only for 30% of the queries.

Based on all the results above, we conclude our proposed techniques (and adaptations) are effective in finding compliant QEPs.

7.3 Optimization Overhead

We proceed to evaluate the overhead of our compliant query optimizer when compared to the traditional cost-based query optimizer. For these experiments, we considered the six TPC-H queries for all four sets of policy expressions. Fig. 6 shows the optimization time in milliseconds and the standard error for these queries.

Minimal Overhead. Here, we are interested in assessing the overhead that our optimizer will always incur for any incoming query. For this we consider policies that impose no dataflow restrictions, i.e., we consider 8 expressions of the form `ship * from` t `to *`, one for each table t in the database (see Tab. 2).

Fig. 6(b) shows this minimal overhead incurred by our optimizer. We observe it is ~ 2× slower than the traditional query optimizer. This is because it additionally relies on annotation rules to derive shipping and execution traits for all operators. As a result, when it derives a trait for each operator, it creates a new equivalence operator node with shipping and execution traits. This leads to a 2× increase in the plan search space. This overhead is most pronounced for query $Q2$ because its QEP involves 13 joins, which is far more than the other queries we considered.

Policy Expressions Impact. We now evaluate the optimization time when considering different types of policy expressions (Figs. 6(c)–6(f)). Note that, for clarity, we report again the optimization times when using the traditional query optimizer. Let us start with the 8 policy expressions in set T (Fig. 6(c)). We observe that the optimization time increases 1.2× as deriving traits requires computing set intersections (recall annotation rules 1–3 from Section 6.1). Next, we analyze the optimization times under the policy expressions in sets C, R, and CR+A. Each set consists of 10 policy expressions. Figs. 6(d), 6(e), and 6(f) show these results, respectively. We observe that the increase in optimization time is more when considering policy expressions of type C. Note the scale of axis in Fig. 6(d). This is because when evaluating a policy expression of type C, the implication test always passes (recall line 3 in Algorithm 1). Policy evaluation is relatively cheaper for expression of type CR and CR+A, for which the implication test fails several times leading to rejecting a policy expression.

Overall, we consider that the overhead of our optimizer is acceptable: it is always less than few 100 milliseconds, except for query $Q2$, which is 900ms. Furthermore, we strongly believe that for most geo-distributed queries, the total query execution time will offset the increase in optimization time.

7.4 Optimization Quality

We now compare the quality of plans produced by our compliant query optimizer with those produced by the traditional query optimizer. We measure quality in terms of execution cost that arises from shipping intermediate query data between geo-distributed sites. To measure this cost, we simulate a network following [12] in which the cost of shipping N bytes from site i to site j takes $\alpha_{ij} + \beta_{ij} \times N$ time. Here α_{ij} is the start-up cost and β_{ij} is the cost per byte. We obtained the start-up cost by finding the time taken by a ping request from region i to j and the cost per byte by finding the time taken to transfer data from region i to j.[12]

The results are shown in Figs. 6(g) and 6(h) for the expressions set C and CR, respectively. Note that we show scaled execution cost w.r.t. traditional query optimization. Letters on top of each bar denote if the plan was compliant (C) or non-compliant (NC) (see also Fig. 5(a)). For queries Q3, Q5, Q8, Q9, and Q10 in Fig. 6(g), and for queries Q5, Q8, and Q9 in Fig. 6(h), we observe that the compliant plan has the same cost as that of the plan produced by traditional query optimizer. This is expected as our approach piggybacks on Volcano's search engine and cost-based pruning, and finds the same plan whenever the traditional optimizer outputs a compliant plan. We show this using the "=" sign. For all other cases, we observed that the cost of enforcing policies depends on the query and policies. For instance, for Q2 in Fig. 6(h), the compliant plan had an overhead of 18×. This stems from shipping the larger `Supplier` data (~8M tuples), which is compliant as opposed to shipping the smaller `Part` data (~8K tuples), which is non-compliant (see also Figs. 5(b) and 5(c)). Similarly, for Q3 and Q10, the compliant plan involved shipping data that is compliant but incurs a higher cost.

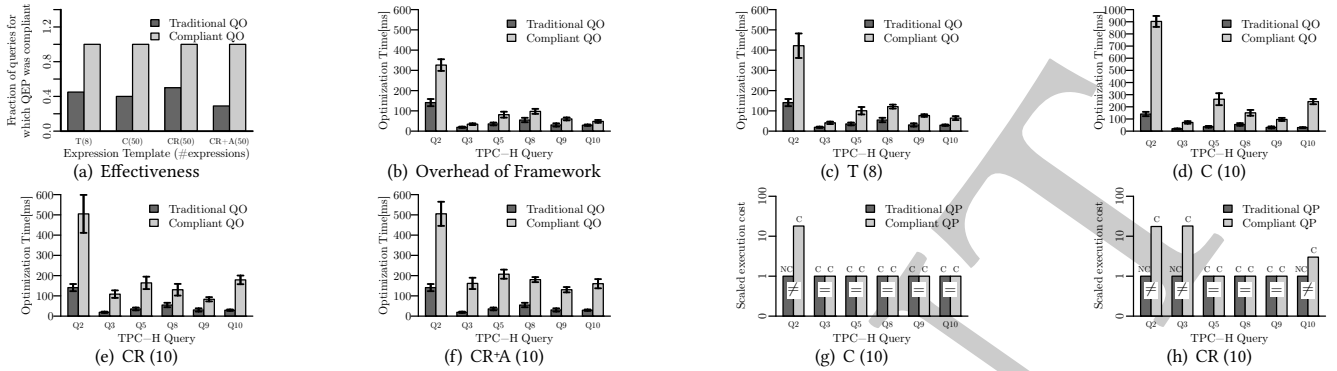
Overall, we found that out approach produced the same plans as the ones produced by the traditional cost-based approach whenever the later produced a compliant plan. For cases when the traditional cost-based approach produced a non-compliant plan, executing the compliant plan incurred an execution overhead that solely depends on the query and dataflow policies.

7.5 Scalability

We now evaluate the impact of the number of policy expressions and number of locations on optimization time.

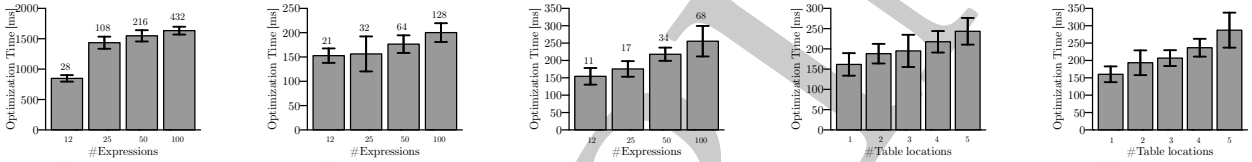
Impact of #policy expressions. We considered queries $Q2$, $Q3$, and $Q10$, and measured their optimization time in presence of 12, 25, 50, and 100 policy expressions of type CR+A. We considered expressions in CR+A as they include simple and aggregate expressions.

[12] Here we considered Europe, Africa, Asia, North America, and Middle East as locations L1–L5, resp. in Tab. 2.



(a) Effectiveness (b) Overhead of Framework (c) T (8) (d) C (10) (e) CR (10) (f) CR+A (10) (g) C (10) (h) CR (10)

Figure 6: Comparisons with traditional cost-based optimization. Effectiveness (a); Optimization overhead (b–f). (b) Minimum overhead (c–f) Optimization time for different types of policy expressions where the number in parenthesis denotes the number of expressions in the set; Optimization quality (g, h). Letters on top of bars denote if plan was compliant (C) or non-compliant (NC). The sign = (≠) denotes if the two plans were the same (not the same).



(a) TPC-H Q2 (b) TPC-H Q3 (c) TPC-H Q10 (d) TPC-H Q3 (e) TPC-H Q10

Figure 7: Scalability w.r.t.: (a – c) the # of expressions: the number on top of each bar denotes $\eta_{q,|E|}$; (d, e) the # of table locations.

Fig. 7 shows the results of these experiments. We first discuss the results for $Q2$ (Fig. 7(a)). We observe that the optimization time increased by ∼1.5× when the number of policy expressions increased from 12 to 25. Thereafter, upon doubling the number of policy expressions had little effect: the optimization time increased by up to ∼1.1× only. As we can see, this trend varies for the other queries. To better understand this trend, we additionally computed $\eta_{q,|E|}$ as the number of times a policy expression is considered by our optimizer for a given query q. Thus, $\eta_{q,|E|}$ denotes the number of times when one ore more `ship` attributes are present in a query's subexpression and the logical implication holds. In other words, it denotes when the policy evaluation algorithm (Algorithm 1) reaches line 4. The number on top of each bar shows the η's value for $Q2$, $Q3$, and $Q10$. We observe that the increase in optimization time upon doubling the number of expressions is proportional to the increase in the corresponding η values. For instance, doubling the number of policy expressions from 12 to 25 for $Q3$ only leads to a 1.5× increase in η. This explains why its optimization time increased compared to when the number of policy expressions doubled from 25 to 50. For $Q10$, the optimization time increases linearly as η increases too.

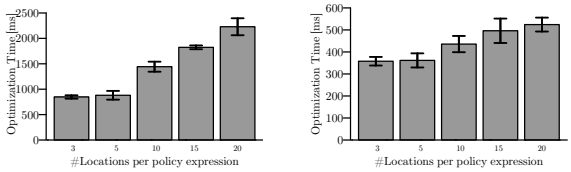
It is worth noting that it is highly unlikely that all policies will affect the search space of all queries. We, thus, conclude that our optimizer scales gracefully with number of policy expressions.

Impact of # table locations. We also extended our experimental setup in which TPC-H `Customer` and `Orders` tables are distributed among locations L1–L5. We consider TPC-H queries Q3 and Q10 and expressions of type CR+A, and measure their optimization time. The results are shown in Figures 7(d) and 7(e). We observed that the optimization time increased roughly linearly with respect to the number of locations, and this increase mainly stemmed from the plan annotator with site selector constituting a small fraction (up to 2ms). This is because a table t that is distributed among n locations is first rewritten as $t_1 \cup \cdots \cup t_n$, where t_i is a `tablescan` operator, which leads to an increase in the plan space of plan annotator, and consequently to an increase in total optimization time.

Impact of #locations in constraints. Lastly, we study how the number of '`to`' locations in policy expressions affect the optimization time. We consider eight policy expression of type `ship * from` t `to` $l_1, \ldots, l_n$, for each TPC-H table t, and vary the number of locations n from 3–20. Figure 8 shows the results for TPC-H Q2 and Q3. We chose these queries because they have the most and the least number of joins. We observed that for Q2, the optimization time increased by ∼1.6× as we increased n from 5 to 10, and it increased further by ∼1.7× for $n = 20$. This is because the number of locations in expressions do not impact the plan space, and the increase in optimization time mainly stems from the computing set operations while invoking annotation rules, which in turn depends on the number of locations in policy expressions (recall Section 6.1). For Q3, the increase in optimization time was ∼1.2× as we doubled the number of locations. Note that this increase in more pronounced for Q2 as its optimization involves computing 5× more set operations (when deriving traits) than Q3. We also observed that the increase in site selection time, was not linear w.r.t. the number of locations, however, it constituted only a small fraction (up to 1.5%) of total optimization time. For example, for Q2, it increased from 3ms (for $n = 3$) to 35ms (for $n = 20$).

8 RELATED WORK

Distributed Query Processing. This is a well studied problem in literature. A large body of work [12, 29, 30, 32–34, 41, 48, 55] has



(a) TPC-H Q2 (b) TPC-H Q3

Figure 8: Impact of #locations in policy expressions.

focused on many aspects of the problem, including system architecture, heterogeneity of data sources, query optimization, and query execution. We refer to [35] for a comprehensive overview on the topic. The problem has also received attention for geo-distributed environments considering different data processing platforms[14]. In this context, Pu et al. [45] address the problem of optimal (MapReduce) task placement among geo-distributed sites such that query response time is minimized. Vulimiri et al. [59, 60] proposed WANalytics and Geode, which seek to minimize data transfer costs. Similar in spirit, Viswanathan et al. [56] proposed the Clarinet system, which also considers WAN-aware optimization for achieving low query response times. All these works are complimentary to our work as they focus on various performance metrics: We primarily focus on compliance with respect to dataflow policies. To the best of our knowledge, none of the existing work considers constraints on data movement. The Geode system by Vulimiri et al. [60] addresses data movement constraints in a very limited way: their solution restricts replication of base data but allow for arbitrary queries (assumed to be legally vetted) on top of base data.

The closest framework to our work is PAQO [19, 20], which allows specifying location preferences for query plan operators using declarative preference clauses [18]. However, PAQO's constraint specification is unsuitable for the dataflow constraints that we consider in this paper. PAQO's constraint specification allows matching parts of the query to plan operators, which when used for dataflow constraints, requires the user to specify location preferences for each plan operator as a part of the user query. In contrast, we propose policy expressions to specify constraints on data and its movement, and automatically derive the (compliant) execution location of plan operators for a query during optimization.

Query Optimization. Among the aforementioned aspects of distributed query processing, our work is most related to query optimization in distributed databases. In particular, our approach to query optimization draws upon the two-step approach [29, 30, 35], which is a well known technique to reduce the overall complexity of optimization in distributed databases [12]. The key difference lies in our use of a customized Volcano optimizer (Sec. 6.1) to produce an annotated plan (Sec. 6.2), which is subsequently processed by the site selector (Sec. 6.3) to determine a compliant QEP.

Database Access Control. Access control mechanisms are an integral part of database systems. View-based methods use database views to enforce authorization to the database [26]. Agrawal et al. [3] as well as LeFevre et al. [40] introduced privacy policies at the data cell-level, but assuming that access control of each data tuple is determined by its owner. Fine-grained access control models include query rewriting techniques [28, 44], extensions to SQL language [8], and using authorization views [47]. We refer to [5] for an in-depth literature on access control. More recently, Shay et al. [52] advocated the use of *query control* as a complimentary method for database access control. In contrast to view-based access control, which focuses on filtering query answers to answer a query, query-based access control limits the query being asked. Our policy expressions (Sec. 3) fall into the category of query-based access control mechanism and can express a wide range of policies than those considered in [52]. Moreover, our expressions are tailored to express data movement between sites, which have not yet been considered in none of these works on access control.

Compliance in Databases. k-anonymization and differential privacy are well known mechanisms for privacy protection. McSherry [42] proposed the PINQ platform that provides differential privacy guarantees. Eltabakh et al. [16] proposed a k-anonymization operator that can be composed with other relational operators to enforce privacy in databases. More recently, compliance with respect to GDPR and other privacy regulations have also received much traction [22]. Shastri et al. [50, 51] and Mohan et al. [43] analyzed various aspects of GDPR compliance w.r.t. data management. Kraska et al. [37, 38] proposed an architectural vision for a database that allows for auditing, deletion, and user consent management. Schwarzkopf et al. [49] proposed abstractions to support privacy rights. In contrast to all these works, our notion of compliance is with respect to data movement incurred by queries. We consider incorporating notions of privacy as important future work.

9 CONCLUSION

More and more applications require executing analytics over data across multiple geo-distributed sites. Several systems have thus appeared to efficiently support geo-distributed query processing. However, they all lack support for compliance with respect to different data movement polices to which sites may be subjected to. In this paper, we have investigated how to make geo-distributed query processing compliant with dataflow policies. We have described the foundations and formalized the problem of compliant query processing. We then have introduced a set of techniques to solve the problem: We have proposed policy expressions as a simple way to express dataflow policies; We have devised an efficient policy evaluation mechanism to determine to which geo-distributed sites a query can ship data; We have showed how to use a Volcano optimizer generator to implement a compliance-based optimizer. Our results on a geo-distributed application of TPC-H data have indicated that our techniques are highly effective as well as efficient.

Our work leads to a number of other interesting follow-up research problems. We list here three of them that we consider important. First, in this paper, we have considered data movement policies that can be adhered to by masking data via relational operations: Supporting data masking via differential privacy or k-anonymization is an important future work. Second, having a hybrid access control mechanism comprising both query- and view-based control is another interesting future work. Third, expanding the scope of queries to beyond relational algebra, i.e., general dataflow programs that have arbitrary user-defined functions and iterations is also an important future work.

Acknowledgments. This work was funded by the German Ministry for Education & Research as BIFOLD – "Berlin Institute for the Foundations of Learning and Data" (01IS18025A and 01IS18037A).

REFERENCES

[1] Regional Privacy Frameworks and Cross-Border Data Flows. https://www.gsma.com/publicpolicy/resources/regional-privacy-frameworks-and-cross-border-data-flows.

[2] Rakesh Agrawal, Roberto Bayardo, Christos Faloutsos, Jerry Kiernan, Ralf Rantzau, and Ramakrishnan Srikant. 2004. Auditing Compliance with a Hippocratic Database. In *VLDB*. 516–527.

[3] Rakesh Agrawal, Jerry Kiernan, Ramakrishnan Srikant, and Yirong Xu. 2002. Hippocratic Databases. In *VLDB*. 143–154.

[4] Aditya Auradkar et al. 2012. Data Infrastructure at LinkedIn. In *ICDE*. 1370–1381.

[5] E. Bertino, G. Ghinita, and A. Kamra. 2011. *Access Control for Databases: Concepts and Systems*.

[6] Apache Calcite. https://calcite.apache.org/.

[7] Francesca Casalini and Javier López González. 2019. Trade and Cross-Border Data Flows. 220 (2019). https://doi.org/https://doi.org/10.1787/b2023a47-en

[8] Surajit Chaudhuri, Tanmoy Dutta, and S Sudarshan. 2007. Fine Grained Authorization Through Predicated Grants. In *ICDE*.

[9] Chandra Chekuri, Waqar Hasan, and Rajeev Motwani. 1995. Scheduling Problems in Parallel Query Optimization. In *PODS*. 255–265.

[10] CockroachDB. https://www.cockroachlabs.com/.

[11] Nigel Cory. 2017. *Cross-Border Data Flows: Where Are the Barriers, and What Do They Cost?* http://www2.itif.org/2017-cross-border-data-flows.pdf

[12] A. Deshpande and J. M. Hellerstein. 2002. Decoupled query optimization for federated database systems. In *ICDE*. 716–727.

[13] AnHai Doan, Alon Halevy, and Zachary Ives. 2012. *Principles of Data Integration* (1st ed.). Morgan Kaufmann Publishers Inc.

[14] S. Dolev, P. Florissi, E. Gudes, S. Sharma, and I. Singer. 2019. A Survey on Geographically Distributed Big-Data Processing Using MapReduce. *IEEE Transactions on Big Data* 5, 1 (2019), 60–80.

[15] Apache Drill. http://drill.apache.org/.

[16] Mohamed Y. Eltabakh, Jalaja Padma, Yasin N. Silva, Pei He, Walid G. Aref, and Elisa Bertino. 2012. Query Processing with K-Anonymity. In *IJDE*, Vol. 3. Issue 2.

[17] F. Emekci, D. Agrawal, A. E. Abbadi, and A. Gulbeden. 2006. Privacy Preserving Query Processing Using Third Parties. In *ICDE*. 27–27.

[18] Nicholas L. Farnan, Adam J. Lee, Panos K. Chrysanthis, and Ting Yu. 2011. Don't Reveal My Intension: Protecting User Privacy Using Declarative Preferences during Distributed Query Processing. In *European Symposium on Research in Computer Security*, Vol. 6879. 628–647.

[19] Nicholas L. Farnan, Adam J. Lee, Panos K. Chrysanthis, and Ting Yu. 2013. PAQO: A Preference-Aware Query Optimizer for PostgreSQL. *Proc. VLDB Endow.* 6, 12 (2013), 1334–1337.

[20] Nicholas L. Farnan, Adam J. Lee, Panos K. Chrysanthis, and Ting Yu. 2014. PAQO: Preference-aware query optimization for decentralized database systems. (2014), 424–435.

[21] Minos N. Garofalakis and Yannis E. Ioannidis. 1997. Parallel Query Scheduling and Optimization with Time- and Space-Shared Resources. In *VLDB*. 296–305.

[22] GDPRbench. https://www.gdprbench.org/gdpr.

[23] Ahmad Ghazal et al. 2013. BigBench: Towards an Industry Standard Benchmark for Big Data Analytics. In *SIGMOD*. 1197–1208.

[24] Jonathan Goldstein and Per-Å ke Larson. 2001. Optimizing Queries Using Materialized Views: A Practical, Scalable Solution. In *SIGMOD*. 331–342.

[25] Goetz Graefe and William J. McKenna. 1993. The Volcano Optimizer Generator: Extensibility and Efficient Search. In *ICDE*. 209–218.

[26] Patricia P. Griffiths and Bradford W. Wade. 1976. An Authorization Mechanism for a Relational Database System. *ACM Trans. Database Syst.* 1, 3 (1976), 242–255.

[27] Ashish Gupta et al. 2014. Mesa: Geo-Replicated, Near Real-Time, Scalable Data Warehousing. *PVLDB* 7, 12 (2014), 1259–1270.

[28] Alon Y. Halevy. 2001. Answering Queries Using Views: A Survey. *The VLDB Journal* 10, 4 (2001), 270–294.

[29] Waqar Hasan and Rajeev Motwani. 1995. Coloring Away Communication in Parallel Query Optimization. In *VLDB*. 239–250.

[30] W. Hong and M. Stonebraker. 1991. Optimization of parallel query execution plans in XPRS. In *ICPDS*. 218–225.

[31] Chien-Chun Hung, Leana Golubchik, and Minlan Yu. 2015. Scheduling Jobs across Geo-Distributed Datacenters. In *SoCC*. 111–124.

[32] Vanja Josifovski, Peter Schwarz, Laura Haas, and Eileen Lin. 2002. Garlic: A New Flavor of Federated Query Processing for DB2. In *SIGMOD*. 524–532.

[33] Holger Kache, Wook-Shin Han, Volker Markl, Vijayshankar Raman, and Stephan Ewen. 2006. POP/FED: Progressive Query Optimization for Federated Queries in DB2. In *VLDB*. 1175–1178.

[34] Anastasios Kementsietsidis, Frank Neven, Dieter Van de Craen, and Stijn Vansummeren. 2008. Scalable Multi-Query Optimization for Exploratory Queries over Federated Scientific Databases. *PVLDB* 1, 1 (2008), 16–27.

[35] Donald Kossmann. 2000. The State of the Art in Distributed Query Processing. *ACM Comput. Surv.* 32, 4 (2000), 422–469.

[36] Ios Kotsogiannis, Yuchao Tao, Xi He, Maryam Fanaeepour, Ashwin Machanavajjhala, Michael Hay, and Gerome Miklau. 2019. PrivateSQL: A Differentially Private SQL Query Engine. *PVLDB* 12, 11 (2019), 1371–1384.

[37] Tim Kraska, Michael Stonebraker, Michael Brodie, Sacha Servan-Schreiber, and Daniel Weitzner. 2019. *SchengenDB: A Data Protection Database Proposal*. 24–38.

[38] Tim Kraska, Michael Stonebraker, Michael Brodie, Sacha Servan-Schreiber, and Daniel J. Weitzner. DATUMDB: A Data Protection Database Proposal.

[39] George Lee, Jimmy J. Lin, Chuang Liu, Andrew Lorek, and Dmitriy V. Ryaboy. 2012. The Unified Logging Infrastructure for Data Analytics at Twitter. *PVLDB* 5, 12 (2012), 1771–1780.

[40] Kristen LeFevre, Rakesh Agrawal, Vuk Ercegovac, Raghu Ramakrishnan, Yirong Xu, and David DeWitt. 2004. Limiting Disclosure in Hippocratic Databases. In *VLDB*. 108–119.

[41] Alon Y. Levy, Anand Rajaraman, and Joann J. Ordille. 1996. Querying Heterogeneous Information Sources Using Source Descriptions. In *VLDB*. 251–262.

[42] Frank D. McSherry. 2009. Privacy Integrated Queries: An Extensible Platform for Privacy-Preserving Data Analysis. In *SIGMOD*. 19–30.

[43] Jayashree Mohan, Melissa Wasserman, and Vijay Chidambaram. 2019. Analyzing GDPR Compliance Through the Lens of Privacy Policy. In *Poly'19 co-located at VLDB 2019*, Vijay Gadepally, Timothy Mattson, Michael Stonebraker, Fusheng Wang, Gang Luo, Yanhui Laing, and Alevtina Dubovitskaya (Eds.).

[44] Amihai Motro. 1989. An Access Authorization Model for Relational Databases Based on Algebraic Manipulation of View Definitions. In *ICDE*. 339–347.

[45] Qifan Pu, Ganesh Ananthanarayanan, Peter Bodik, Srikanth Kandula, Aditya Akella, Paramvir Bahl, and Ion Stoica. 2015. Low Latency Geo-distributed Data Analytics. In *SIGCOMM*. 421–434.

[46] Ariel Rabkin et al. 2014. Aggregation and Degradation in JetStream: Streaming Analytics in the Wide Area. In *NSDI*. 275–288.

[47] Shariq Rizvi, Alberto Mendelzon, S. Sudarshan, and Prasan Roy. 2004. Extending Query Rewriting Techniques for Fine-grained Access Control. In *SIGMOD*. 551–562.

[48] Silvio Salza, Giovanni Barone, and Tadeusz Morzy. 1995. Distributed query optimization in loosely coupled multidatabase systems. In *ICDT*, Georg Gottlob and Moshe Y. Vardi (Eds.). 40–53.

[49] Malte Schwarzkopf, Eddie Kohler, M. Frans Kaashoek, and Robert Morris. Position: GDPR Compliance by Construction.

[50] Supreeth Shastri, Vinay Banakar, Melissa Wasserman, Arun Kumar, and Vijay Chidambaram. 2020. Understanding and Benchmarking the Impact of GDPR on Database Systems. *Proc. VLDB Endow.* 13, 7 (2020), 1064–1077.

[51] Supreeth Shastri, Melissa Wasserman, and Vijay Chidambaram. GDPR Anti-Patterns: How Design and Operation of Modern Cloud-scale Systems Conflict with GDPR. arXiv:1911.00498 [cs.CY]

[52] Richard Shay, Uri Blumenthal, Vijay Gadepally, Ariel Hamlin, John Darby Mitchell, and Robert K. Cunningham. 2019. Don't Even Ask: Database Access Control through Query Control. *SIGMOD Rec.* 47, 3 (2019), 17–22.

[53] Rebecca Taft, Irfan Sharif, Andrei Matei, Nathan VanBenschoten, Jordan Lewis, Tobias Grieger, Kai Niemi, Andy Woods, Anne Birzin, Raphael Poss, Paul Bardea, Amruta Ranade, Ben Darnell, Bram Gruneir, Justin Jaffray, Lucy Zhang, and Peter Mattis. 2020. CockroachDB: The Resilient Geo-Distributed SQL Database. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*. 1493–1509.

[54] Ashish Thusoo et al. 2010. Data Warehousing and Analytics Infrastructure at Facebook. In *SIGMOD*. 1013–1020.

[55] Vasilis Vassalos and Yannis Papakonstantinou. 1997. Describing and Using Query Capabilities of Heterogeneous Sources. In *VLDB*. 256–265.

[56] Raajay Viswanathan, Ganesh Ananthanarayanan, and Aditya Akella. 2016. CLARINET: WAN-Aware Optimization for Analytics Queries. In *OSDI*. 435–450.

[57] Nikolaj Volgushev, Malte Schwarzkopf, Ben Getchell, Mayank Varia, Andrei Lapets, and Azer Bestavros. 2019. Conclave: Secure Multi-Party Computation on Big Data. In *EuroSys*. Article 3, 18 pages.

[58] Ashish Vulimiri et al. 2015. Global Analytics in the Face of Bandwidth and Regulatory Constraints. In *NSDI*. 323–336.

[59] Ashish Vulimiri, Carlo Curino, Brighten Godfrey, Konstantinos Karanasos, and George Varghese. 2015. WANalytics: Analytics for a Geo-Distributed Data-Intensive World. In *CIDR*.

[60] Ashish Vulimiri, Carlo Curino, P. Brighten Godfrey, Thomas Jungblut, Jitu Padhye, and George Varghese. 2015. Global Analytics in the Face of Bandwidth and Regulatory Constraints. In *NSDI*. 323–336.

[61] Qihua Wang, Ting Yu, Ninghui Li, Jorge Lobo, Elisa Bertino, Keith Irwin, and Ji-Won Byun. 2007. On the Correctness Criteria of Fine-Grained Access Control in Relational Databases. In *VLDB*. 555–566.

[62] Zhe Wu et al. 2013. SPANStore: Cost-Effective Geo-Replicated Storage Spanning Multiple Cloud Services. In *SOSP*. 292–308.