RESEARCH-ARTICLE

Language-integrated privacy-aware distributed queries

GUIDO SALVANESCHI, Technical University of Darmstadt, Darmstadt, Hessen, Germany

MIRKO KÖHLER, Technical University of Darmstadt, Darmstadt, Hessen, Germany

DANIEL SOKOLOWSKI, Technical University of Darmstadt, Darmstadt, Hessen, Germany

PHILIPP HALLER, KTH Royal Institute of Technology, Stockholm, Stockholms, Sweden

SEBASTIAN ERDWEG, Johannes Gutenberg University Mainz, Mainz, Rheinland-Pfalz, Germany

M. MEZINI, Technical University of Darmstadt, Darmstadt, Hessen, Germany

Proceedings of the ACM on Programming Languages, Volume 3, Issue OOPSLA (October 2019)
https://doi.org/10.1145/3360593
EISSN: 2475-1421

Language-Integrated Privacy-Aware Distributed Queries

GUIDO SALVANESCHI, Technische Universität Darmstadt, Germany
MIRKO KÖHLER, Technische Universität Darmstadt, Germany
DANIEL SOKOLOWSKI, Technische Universität Darmstadt, Germany
PHILIPP HALLER, KTH Royal Institute of Technology, Sweden
SEBASTIAN ERDWEG, Johannes Gutenberg-Universität Mainz, Germany
MIRA MEZINI, Technische Universität Darmstadt, Germany

Distributed query processing is an effective means for processing large amounts of data. To abstract from the technicalities of distributed systems, algorithms for *operator placement* automatically distribute sequential data queries over the available processing units. However, current algorithms for operator placement focus on performance and ignore privacy concerns that arise when handling sensitive data.

We present a new methodology for privacy-aware operator placement that both prevents leakage of sensitive information *and* improves performance. Crucially, our approach is based on an information-flow type system for data queries to reason about the sensitivity of query subcomputations. Our solution unfolds in two phases. First, placement space reduction generates deployment candidates based on privacy constraints using a syntax-directed transformation driven by the information-flow type system. Second, constraint solving selects the best placement among the candidates based on a cost model that maximizes performance. We verify that our algorithm preserves the sequential behavior of queries and prevents leakage of sensitive data. We implemented the type system and placement algorithm for a new query language SecQL and demonstrate significant performance improvements in benchmarks.

CCS Concepts: • **Security and privacy** → **Domain-specific security and privacy architectures**; • **Computing methodologies** → *Distributed programming languages.*

Additional Key Words and Phrases: Data Privacy, SQL, Information-Flow Type System, Operator Placement, Scala

ACM Reference Format:
Guido Salvaneschi, Mirko Köhler, Daniel Sokolowski, Philipp Haller, Sebastian Erdweg, and Mira Mezini. 2019. Language-Integrated Privacy-Aware Distributed Queries. *Proc. ACM Program. Lang.* 3, OOPSLA, Article 167 (October 2019), 30 pages. https://doi.org/10.1145/3360593

Authors' addresses: Guido Salvaneschi, Technische Universität Darmstadt, Germany, salvaneschi@cs.tu-darmstadt.de; Mirko Köhler, Technische Universität Darmstadt, Germany, koehler@cs.tu-darmstadt.de; Daniel Sokolowski, Technische Universität Darmstadt, Germany, sokolowski@cs.tu-darmstadt.de; Philipp Haller, KTH Royal Institute of Technology, Sweden, phaller@kth.se; Sebastian Erdweg, Johannes Gutenberg-Universität Mainz, Germany, erdweg@uni-mainz.de; Mira Mezini, Technische Universität Darmstadt, Germany, mezini@cs.tu-darmstadt.de.

2475-1421/2019/10-ART167
https://doi.org/10.1145/3360593

1 INTRODUCTION

Language-integrated queries address the impedance mismatch between the database query languages and conventional programming languages (Copeland and Maier 1984). Prior research on language-integrated queries has focused on syntactic and translational aspects, ensuring that application developers can author queries in the language and the compiler translates those language-integrated queries into proper database queries (Cheney et al. 2013). In this paper, we argue and demonstrate that language-integrated queries provide advantages beyond syntactic integration. Specifically, we show how language-integrated queries enable query-aware static analysis within and across queries, and we show how these analysis results can be used during context-aware query compilation. This way, our language-integrated queries enable *privacy-aware distributed operator placement*.

A major challenge to achieve high performance in a distributed system is utilizing all available processing units to their full extent, taking load balancing, latency, and bandwidth into account. In particular, systems that solve the *operator placement problem* (Cugola and Margara 2013; Lakshmanan et al. 2008) translate high-level descriptions of sequential computations into efficient distributed computations providing a high-level language-based interface to distributed systems. In this paper, we consider operator placement from the perspective of *data privacy*.

While it is relatively easy to secure communication channels in a distributed system, it is virtually impossible to protect sensitive data unless one controls the machine and can prevent a concurrent process, the OS, or the hardware from leaking information. However, requiring full control over the execution environment for ensuring privacy contradicts the trend towards cloud-based computing services where third parties provide processing units on demand. Data privacy issues also occur due to legal circumstances that require data to remain within a geographical region or legal body.

The main contribution of this paper is a two-phase algorithm for automated privacy-aware operator placement of language-integrated queries over relational data. The first phase, *placement space reduction*, generates deployment candidates that do not violate privacy constraints. The second phase finds the best placement based on a cost model. Placement space reduction depends on three pieces of information: (1) on the sensitivity of the source relations, where we allow each column to declare a different sensitivity, for example, to differentiate between a person's name, home address, and bank account; (2) on the privilege of the involved processing units, which determines what data may be forwarded where; and (3) on the information flow of the query, because privacy concerns only arise where sensitive data can flow to a non-privileged processing unit.

The second phase, *cost model placement*, is based on metrics from the system (we consider the case of link bandwidth and CPU load, but the approach can easily be extended to other metrics such as latency). The selectivity of operators indicates the ratio between the amount of input and output data for each operator and is obtained via monitoring. The cost model placement receives a set of candidate deployments from placement space reduction and a system specification including the selectivity of operators to find the optimal placement using constraint solving.

Our approach derives an operator placement that is provably correct: it preserves the sequential behavior and never leaks sensitive data. To reason about the flow of information through language-integrated queries, we have developed a query-aware static taint analysis in form of an information-flow type system, building on previous work on information flow (e.g., by Myers (1999); Volpano et al. (1996)). Our type system tracks the taint of each column individually and is parametric in the language used for defining projections and selections. An important challenge our type system solves is correctly tracking taints when a projection aggregates data from different columns and when a selection correlates data from different columns. We use the type system to derive

the information flow of the input query accurately and to reason about the correctness of the placement algorithm. Our approach has a deeper integration than existing LINQs. Specifically, the information-flow type system is language-integrated, calling upon an information-flow type system of the host language. For example, in SecQL, it is essential that we obtain taint information about projection and selection functions, which are written in Scala.

The fundamental contribution of this work is that, for the first time, we tackle the operator placement problem using language-based techniques. Crucially, our approach allows us to reason about placement with the same formalism that specifies the information flow and formally derive correctness properties. Yet, thanks to the two-step approach, we still retain the advantages of placement based on metrics, which come into play in the second phase.

We have implemented our type system and privacy-aware operator placement algorithm for a language-integrated query language called SecQL[1] in Scala. Based on the result of the operator placement, we distribute input queries using Akka actors (Akka 2019) with asynchronous message passing. We have evaluated SecQL on a benchmark suite of business-oriented ad-hoc queries inspired by TPC-H (TPC 2019), as well as on a case study for a hospital information system which we introduce in the following Section. The contributions of this paper are as follows:

- We identify and discuss interactions between data privacy and operator placement for distributed queries (Section 2).
- We develop a query-aware static taint analysis to reason about the flow of sensitive information (Section 3).
- We design a two-step mechanism for privacy-aware operator placement amenable to formal proof – that it preserves the original query behavior and does not leak sensitive data. The first step (Section 4), formalized as a code transformation, receives a query's information flow as input and outputs a set of constraints for the second step (Section 5), based on a cost model to account for performance.
- We demonstrate the applicability of our approach with performance benchmarks and a case study (Section 6 and Section 7). Privacy-aware operator placement (1) decreases query run time by up to 97% in comparison to a privacy preserving deployment, which does not use our deployment logic in a suite of business oriented ad-hoc queries, and (2) the performance is comparable to a non-private, optimal placement demonstrating that our approach can provide privacy almost for free in a number of realistic use cases.

The rest of the paper is organized as follows. Section 2 motivates key issues of privacy-aware operator placement and provides an informal overview of SecQL, including its operator placement algorithm. Section 3 formalizes SecQL's taint analysis using a type system. In Section 4, we formalize SecQL's privacy-aware operator placement algorithm and prove key theorems. In Section 5, we describe constraint resolution with a cost model. Section 6 provides an overview of SecQL's implementation. In Section 7, we present an experimental evaluation of SecQL. Section 8 reviews related work, and Section 9 concludes.

2 SECQL AND PRIVACY-AWARE PLACEMENT

We use a running example of a hospital information system to motivate and introduce our approach. In the hospital, the clinical database *PatientDB* stores information about the current patients in the hospital. The *KnowledgeDB* database contains data of case reports (symptoms and corresponding diagnoses, etc.). In addition, the *PersonDB* database provides general information about citizens. We assume that all databases are running on different hosts. Their information can be combined to

[1]The implementation with the case studies and the benchmarks: https://github.com/stg-tud/SecQL/tree/OOPSLA19.

```
1 val result = join((person, patient, knowledge) =>
2   patient.symps == knowledge.symps,
3   join((person,patient) => person.id == patient.id,
4     personDB,
5     patientDB
6   ),
7   selection(knowledge =>
8     knowledge.diagnosis == "Allergy",
9     knowledgeDB
10  ))
```

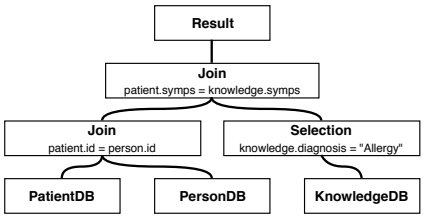


Fig. 1. Local query.

```
1 val result = join((person, patient, knowledge) =>
2   patient.symps == knowledge.symps,
3   join((person,patient) => person.id == patient.id,
4     remote(client, personDB),
5     remote(client, patientDB)
6   ),
7   selection(knowledge =>
8     knowledge.diagnosis == "Allergy",
9     remote(client, knowledgeDB)
10  ))
```

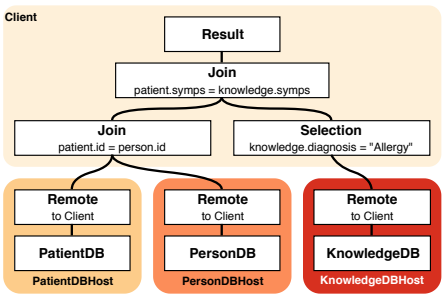


Fig. 2. Distributed query.

automatically find diagnoses and suggestions for the current patients, e.g., a doctor can request the name of all patients that have symptoms that could point to allergy.

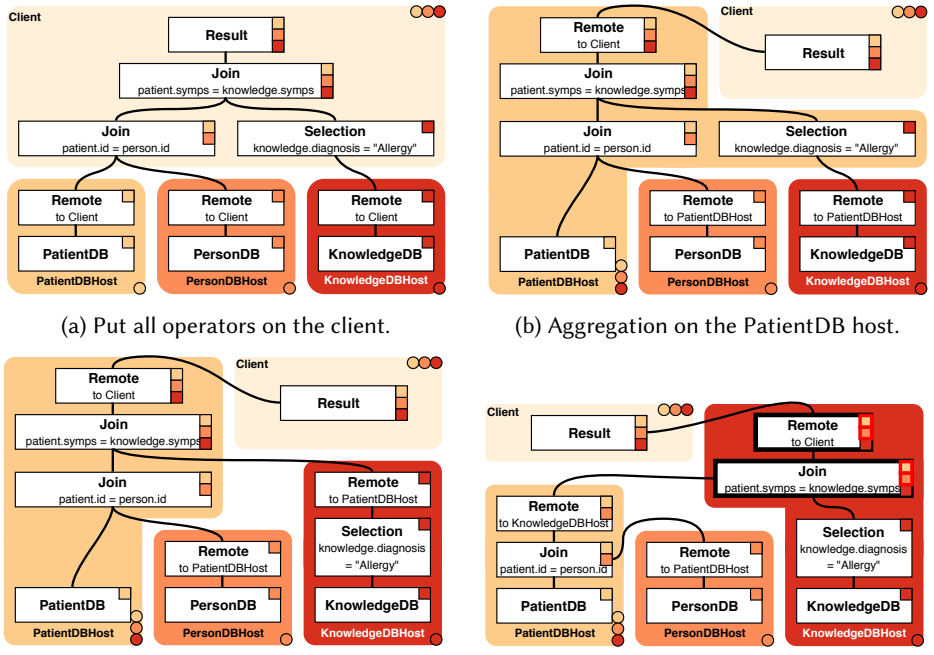
```
1 val result = SELECT (*)
2   FROM (personDB, patientDB, knowledgeDB)
3   WHERE ((person, patient, knowledge) =>
4     person.id == patient.id AND
5     patient.symps == knowledge.symps AND
6     knowledge.diagnosis == "Allergy")
```

For simplicity, in the running example we assume that each database only consists of a single table (e.g., the *PersonDB* contains a *PersonDB* table) and use the terms database and table as equivalent in the following.

2.1 Local Queries and Operator Placement

The query presented in the previous Section desugars to the expression in Figure 1 (left). Such an expression corresponds to the relational algebra operator tree as shown in Figure 1 (right). In the tree representation, the condition from the WHERE-clause has been split into three operators: the selection `knowledge.diagnosis == "Allergy"` (close to the `KnowledgeDB` source), the join `person.id == patient.id`, and the join `patient.symps == knowledge.symps`.

We now proceed with distributing the operators in the query above across multiple machines (e.g., in a private cloud). To this end, we express distribution in the language. Please note that this will be eventually performed automatically by our system and the user only has to specify the location of the data sources and sink explicitly. We introduce the `remote` operator, which is placed together with a query in a subexpression on a different machine and creates a remote link to its parent operator. For example, `remote(toHost, q)` places the query `q` on a remote host such that the host `toHost` receives the result of executing `q`. The location of the remote operator itself can be



(a) Put all operators on the client.

(b) Aggregation on the PatientDB host.

(c) Selection on KnowledgeDB host.

(d) Privacy violation.

Fig. 3. Query tree distribution examples.

evaluated bottom-up, because the locations of the query tree leaves (the tables) are known. For instance, the query shown before can be transformed adding the `remote` markers as in Figure 2 (left). Here, we assume that every database is deployed on a different host and that all operators as well as the result relation are placed on the client. For each table, there is a remote link to the client. This corresponds to the distribution in Figure 2 (right), where hosts are illustrated by different background boxes.

2.2 The Effect of Privacy on Operator Placement

The sensitivity of the information in the databases has an effect on the operator placement. For example, if the hosts of *PersonDB*, *PatientDB*, and *KnowledgeDB* do not have access rights for each other's data, the only place where data can be aggregated is the client, as shown in Figure 3a. This approach requires to (1) perform computation on the client and (2) transfer all data to the client. The consequence is higher load and higher bandwidth consumption. For the other placement examples in Figure 3, we assume the following privacy constraints for the tables. The information in *PatientDB* is private, which means that only the client and the host of *PatientDB* should be able to access the data. In contrast, *KnowledgeDB* and *PersonDB* contain more public data, which may not only be shared with the client, but also with the host of *PatientDB*. In Figure 3 the taint of an operator is indicated by colored squares and the permissions of a host by colored circles.

Less restrictive privacy constraints enable more placement choices. As shown in Figure 3b, the data in *PersonDB* and *KnowledgeDB* can be transferred to the host of *PatientDB*, aggregated there, and then sent to the client. This reduces the amount of computation that must be performed on the client and the amount of data transferred over the network.

Another alternative is to move the selection of the diagnosis (`knowledge.diagnosis == "Allergy"`) to the host of *KnowledgeDB*, as shown in Figure 3c. This approach further reduces network consumption, because only the *KnowledgeDB* records on allergies are transferred to the host of *PatientDB*.

Some placements may not satisfy a given privacy configuration. For example, moving the join of the field `symps` to the host of *KnowledgeDB* introduces a privacy violation, because that host would then access data from *PatientDB*, which may not leak to the *KnowledgeDB* host (Figure 3d).

2.3 Privacy in SecQL

In SecQL, security classes (i.e. sensitivity) of data are specified by labels. Labels are assigned column-wise. Each host can only read data with security classes when it has the corresponding permission. SecQL extends relational algebra operators with new syntax for modeling aspects that concern distribution and security. First, it is possible to specify the host on which a table lives. In the query context, we specify which hosts exist via a map. The following code snippet demonstrates how to associate labels to hosts:

```
implicit val queryContext = QueryEnvironment.create(Map(
    host1 -> Set("green", "red")))
```

The implicit above assigns the `green` label and the `red` label to the host named `host1`. Developers can also declassify a query to lower the security class of data. The following code snippet removes the `red` label and the `blue` label from the security class of `table1`:

```
DECLASS (SELECT (*) FROM table1, "red", "blue")
```

The declassification can be used to lower the security class after an aggregation, in case the programmer knows that no sensitive information is disclosed:

```
DECLASS (SELECT SUM (*) FROM t, "red")
```

The responsibility of declassification is on the developer, because it is generally an unsafe operation. In the context of the hospital IT system, declassification can be used, for example, to efficiently perform queries on anonymized data. The information-flow type system may fail to recognize the anonymization step: then, it is necessary to manually declassify the data so that it can be processed with fewer permissions. In general, declassification is a widely acknowledged important functionality for this class of systems with a number of use cases (Myers and Liskov 1997; Zdancewic 2013). In SecQL, we assume that queries are written by a trusted administrator, and thus the usage of declassification is always trusted. A discussion of related work, orthogonal to ours, aiming at relaxing such assumption is in Section 8.

In our **threat model**, we assume that the client can access all data necessary to compute a query result. Also, the assumption in SecQL is that a trusted administrator writes the SecQL program, which is partitioned and distributed among the hosts in the system by our (trusted) distribution engine. Hosts can be malicious and can try to access data resources that they are not allowed to access. SecQL ensures that hosts cannot access data resources violating the specified access control policies. As usual, we assume that entities in our system are labeled with sets of security classes $A ::= \{\overline{a}\}$ that form a lattice. In this work, we consider a select-only attacker who cannot run insert/update queries. Inserting data in the source databases requires authentication and is not interesting for the operator placement problem, which is the focus of this work. For the same reason, triggers are not part of the system model.

2.4 Placement

SecQL automatically finds optimal deployments for queries in a distributed system in a way that satisfies privacy requirements. An overview of SecQL's approach is shown in Figure 4.

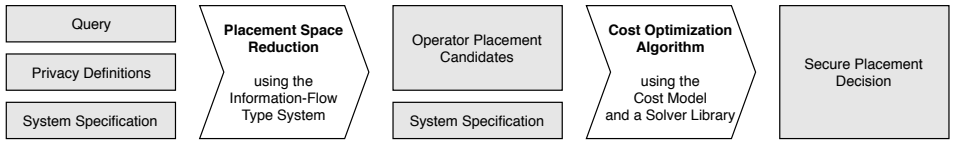


Fig. 4. Overview of SecQL.

SecQL receives as input the user query, the privacy definitions and a system specification. The query is optimized with standard techniques (e.g., pushing down selections) beforehand, which is not further discussed in this work. The privacy definitions include the security classes, i.e. labels, for all data sources and the host permissions. The system specification contains information about the nodes' resources.

The query is processed in two phases. First, *placement space reduction* generates all possible deployments that do not violate the privacy constraints. The constraints are derived from the system specification and the privacy definitions through the *information-flow type system*, which evaluates the permissions that are needed to access the data processed by each operator. SecQL ensures that all operators are deployed on hosts with sufficient permissions.

Second, *cost model optimization* considers the performance estimation for each deployment to find the optimal deployment among all candidates. The cost of a deployment is defined by a combination of bandwidth and machine resources usage (Section 5). The output is a single deployment that is both optimal and secure.

3 SECQL INFORMATION FLOW

We formalize SecQL as an SQL-like core calculus with a denotational semantics. To track the flow of source data through SecQL queries, we define a static taint analysis for our calculus in the form of an information-flow type system.

3.1 SecQL Core Calculus

$$
\begin{array}{lcll}
x & ::= & \text{<name>} & \text{Table identifier} \\
l & ::= & \text{<name>} & \text{Column identifier} \\
bv & ::= & \text{<number>} \mid \text{<string>} \mid \ldots & \text{Base values} \\
v & ::= & bv \mid \{\overline{l = v}\} & \text{Values} \\
tv & = & 2^{v} & \text{Table values}
\end{array}
$$

$$
\begin{array}{lcll}
q & ::= & \text{tableref}(x) \mid \text{select}(q, e) & \text{Queries} \\
 & & \mid \text{project}(q, e) \mid \text{crossProduct}(q, q) & \\
 & & \mid \text{union}(q, q) \mid \text{intersect}(q, q) & \\
 & & \mid \text{difference}(q, q) & \\
e & ::= & \text{<exp>} & \text{Expressions} \\
B & ::= & \text{Number} \mid \text{String} \mid \ldots & \text{Base types} \\
R & ::= & B \mid \{\overline{l : R}\} & \text{Raw untainted types}
\end{array}
$$

Fig. 5. Syntax of SecQL's core calculus.

Figure 5 shows the syntax of the SecQL core calculus. We assume globally scoped identifiers x to identify tables, such as *PersonDB* and *PatientDB* in the previous Section. Column identifiers l distinguish individual columns in those tables. Values v are either base values bv such as numbers, or record values that assign base values or nested record values to columns. A table value consists of a set of records, each representing a row in the table. The query language q is standard except that we abstract over the expression language e used for programming projections and selections. A notable omission of our core calculus is joins, which can be modeled using selections and cross products.

As a reference, we present the denotational semantics of our calculus in Figure 6. The semantics $[\![\cdot]\!]$ of queries q is standard for relational algebra (Dyreson and Snodgrass 1998; Vassiliou 1979) except that we abstract over the semantics $[\![\cdot]\!]^{exp}$ of the expression language e. In the semantics

$$[\![\cdot]\!]^{exp}_{\tau} : e \to [\![\tau]\!]$$
$$[\![\cdot]\!]^{exp}_{\tau} = \ldots \quad \text{(left abstract)}$$
$$[\![\cdot]\!]_R : q \to (x \mapsto tv) \to 2^R$$
$$[\![\text{tableref}(x)]\!]_R(\rho) = \rho(x)$$
$$[\![\text{select}(q, e)]\!]_R(\rho) = \{v \mid v \in [\![q]\!]_R(\rho), f = [\![e]\!]^{exp}_{R \to \text{Bool}}, f(v)\}$$
$$[\![\text{project}(q, e)]\!]_{R_2}(\rho) = \{f(v) \mid v \in [\![q]\!]_{R_1}(\rho), f = [\![e]\!]^{exp}_{R_1 \to R_2}\}$$
$$[\![\text{crossProduct}(q_1, q_2)]\!]_{\{l_1:R_1, l_2:R_2\}}(\rho) = \{\{l_1 = v_1, l_2 = v_2\} \mid v_1 \in [\![q_1]\!]_{R_1}(\rho), v_2 \in [\![q_2]\!]_{R_2}(\rho)\}$$
$$[\![\text{union}(q_1, q_2)]\!]_R(\rho) = [\![q_1]\!]_R(\rho) \cup [\![q_2]\!]_R(\rho)$$
$$[\![\text{intersect}(q_1, q_2)]\!]_R(\rho) = [\![q_1]\!]_R(\rho) \cap [\![q_2]\!]_R(\rho)$$
$$[\![\text{difference}(q_1, q_2)]\!]_R(\rho) = [\![q_1]\!]_R(\rho) \setminus [\![q_2]\!]_R(\rho)$$

Fig. 6. Denotational semantics of SecQL's core calculus.

a ::= <name>	Taint identifier	T ::= $B{\sim}A \mid \{\overline{l : T}\}$	Tainted types
A ::= $\{\overline{a}\}$	Taint set	q ::= $\ldots \mid \text{declass}(q, A)$	Declassification query

Fig. 7. Additional syntax for tainted types in SecQL.

$$\text{taint} : T \to A$$
$$\text{taint}(B{\sim}A) = A$$
$$\text{taint}(\{\overline{l : T}\}) = \bigcup \overline{\text{taint}(T)}$$

$$\text{raw} : T \to R$$
$$\text{raw}(B{\sim}A) = B$$
$$\text{raw}(\{\overline{l : T}\}) = \{\overline{l : \text{raw}(T)}\}$$

$$\text{lift} : R \to T$$
$$\text{lift}(B) = B{\sim}\emptyset$$
$$\text{lift}(\{\overline{l : R}\}) = \{\overline{l : \text{lift}(R)}\}$$

$$\cdot + \cdot : T \to A \to T$$
$$(B{\sim}A_0) + A = B{\sim}(A_0 \cup A)$$
$$\{\overline{l : T}\} + A = \{\overline{l : (T + A)}\}$$

$$\cdot - \cdot : T \to A \to T$$
$$(B{\sim}A_0) - A = B{\sim}(A_0 \setminus A)$$
$$\{\overline{l : T}\} - A = \{\overline{l : (T - A)}\}$$

$$\cdot \uplus \cdot : T \times T \to T$$
$$B{\sim}A_1 \uplus B{\sim}A_2 = B{\sim}(A_1 \cup A_2)$$
$$\{\overline{l : T_1}\} \uplus \{\overline{l : T_2}\} = \{\overline{l : T_1 \uplus T_2}\}$$
$$\_ \uplus \_ = \textit{undefined}$$

Fig. 8. Auxiliary functions over tainted types.

of the selection operator, we use the semantics of the expression language to obtain a filtering function $f : R \to \text{Bool}$, and in the semantics of the projection operator, we use it to obtain a mapping function $f : R_1 \to R_2$.

3.2 Tainted Types

We define an information-flow type system for the SecQL core calculus to track the flow of sensitive information. To this end, we introduce taint identifiers a that mark which privilege is required for accessing the annotated information. Sets of taint identifiers A can occur as annotations on base types B and our type system tracks taints through queries. Figure 7 introduces the relevant additional syntax. As the Figure shows, we also introduce syntax to allow a query to declassify sensitive information by removing taints A. While the type system takes declassification into account, the dynamic semantics of SecQL simply ignore declassification:

$$[\![\text{declass}(q, A)]\!]_R(\rho) = [\![q]\!]_R(\rho)$$

As usual in information-flow systems, the usage of declassification is controlled externally.

Before presenting the information-flow type system for SecQL, we first introduce a number of auxiliary functions in Figure 8. Function raw and taint extract the raw type R and taint set A of a tainted type T, respectively. Function lift maps a raw type R to a tainted type T with empty taint sets throughout. Functions + and − take a tainted type T and add/remove a taint set A from all taint annotations within T. Finally, function $\uplus$ combines the taints of two tainted types with the same raw type or is undefined otherwise.

For selection and projection, the type system depends on type checking and taint analysis for the expression language e. To this end, we assume the existence of a function taintOfBase with the following signature:

$$\text{taintOfBase} : e \longrightarrow T \longrightarrow A$$

Given expression e and tainted type T, taintOfBase computes the taint set corresponding to the output values of e when fed a value of type T. That is, taintOfBase(e, T) statically approximates the taints required to compute $[\![e]\!]_R^{exp}(v)$ for all $v \in [\![T]\!]$. Note that taintOfBase does not introduce other taints than already defined in T, because the output values of e rely only on the input value with type T. Thus, taintOfBase$(e, T) \subseteq \text{taint}(T)$. Moreover, we assume the existence of a typing judgment for expressions e:

$$\Vdash e : R_1 \longrightarrow R_2$$

In practice, the typing judgment and taint analysis for expressions may be defined simultaneously in a single function.

3.3 SecQL **Information-Flow Type System**

$$\frac{C(x) = T}{C \vdash \text{tableref}(x) : T} \quad \text{(T-Table)}$$

$$\frac{C \vdash q : T \qquad \Vdash e : \text{raw}(T) \longrightarrow \text{Bool} \qquad A = \text{taintOfBase}(e, T)}{C \vdash \text{select}(q, e) : T + A} \quad \text{(T-Select)}$$

$$\frac{C \vdash q : T \qquad \Vdash e : \text{raw}(T) \longrightarrow R \qquad \forall\, l \in \text{fields}(R).\ A_l = \text{taintOfBase}(\lambda v.\ e(v).l, T) \qquad \forall\, l \in \text{fields}(R).\ S_l = \text{lift}(R.l) + A_l}{C \vdash \text{project}(q, e) : \{\overline{l : S_l}\}_{l \in \text{fields}(R)}} \quad \text{(T-Proj)}$$

$$\frac{C \vdash q_1 : T_1 \qquad C \vdash q_2 : T_2}{C \vdash \text{crossProduct}(q_1, q_2) : \{l_1 : T_1, l_2 : T_2\}} \quad \text{(T-CProd)}$$

$$\frac{C \vdash q_1 : T_1 \qquad C \vdash q_2 : T_2 \qquad \text{raw}(T_1) = \text{raw}(T_2)}{C \vdash \text{union}(q_1, q_2) : T_1 \uplus T_2} \quad \text{(T-Union)}$$

$$\frac{C \vdash q_1 : T_1 \qquad C \vdash q_2 : T_2 \qquad \text{raw}(T_1) = \text{raw}(T_2)}{C \vdash \text{intersect}(q_1, q_2) : T_1 \uplus T_2} \quad \text{(T-Intersect)}$$

$$\frac{C \vdash q_1 : T_1 \qquad C \vdash q_2 : T_2 \qquad \text{raw}(T_1) = \text{raw}(T_2)}{C \vdash \text{difference}(q_1, q_2) : T_1 \uplus T_2} \quad \text{(T-Diff)}$$

$$\frac{C \vdash q : T}{C \vdash \text{declass}(q, A) : T - A} \quad \text{(T-Declass)}$$

Fig. 9. Information-flow type system for SecQL.

We are now ready to discuss the information-flow type system for SecQL, which we show in Figure 9. The typing relation is expressed by the typing judgment $C \vdash q : T$. The environment $C : x \mapsto T$ maps each source table to a (possibly) tainted type of table elements.

T-Table: The type of a table is provided by the environment C.

T-Select: The rule extends the taint of every column in type T with the taint set A – the taints of all T-fields that are used during the evaluation of e. This models the fact that information about these fields can be gained by looking at any field of T after the selection operation. Therefore, every field of T is now tainted by the fields that have been used in e.

T-Proj: We type check the subquery q and the well-typed expression e. To obtain tainting that covers all information flows precisely, we call taintOfBase separately for each column value in the result type R of e. To this end, we construct the auxiliary program $(\lambda v.\ e(v).l)$, which represents that slice of e that solely computes the value for column l, and we obtain the corresponding taint set A_l for that column. From $R.l$ and the taint set A_l, we construct the tainted type S_l and finally the return type of the projection operator.

T-CProd: The cross product creates a nested record that contains two columns l_1 and l_2, which store values yielded by the left and right subquery, respectively. The taints of values within the nested records do not change.

T-Union, T-Intersect and T-Diff: The based operators can only be safely used when the raw types of the subquery coincide. In these cases $\uplus$ is well-defined and combines the taints of T_1 and T_2 as to conservatively approximate the actual taint.

T-Declass: Declassification recursively removes all taints in A from the type T of the subquery.

Discussion. Our type system provides a specification for the information flow of SecQL queries. The core language of our SecQL implementation is agnostic w.r.t. the base language. For T-Select and T-Proj, we adopt a conservative approach in the implementation that considers the worst case: To account for all potential correlations, the taint of all input fields is assigned to each field in the output. A more accurate static analysis of e in select(q, e) and project(q, e) can determine whether the output only depends on a subset of the fields in q and hence only the taint of those fields should be propagated. We implemented the type system according to the rules above, plus simple extensions to account for operators not included in the core calculus such as joins (Section 6). In the next Section, we use the type system to inform privacy-aware operator placement and to reason about the correctness of our placement algorithm.

4 PLACEMENT SPACE REDUCTION

We present the first step for privacy-aware operator placement. Placement space reduction selects candidate placements based on the operator's taint and based on the privilege of available machines. We model the generation of candidate placements as a query transformation that adds placement markers to the query. These markers signify machine boundaries in the deployed query. To reason about deployed queries and the privilege of the available machines, we first define a few extensions to SecQL. We then present the placements generation algorithm and verify its correctness.

4.1 Distributed SecQL

We make the following changes to the syntax of SecQL:

$$\begin{array}{lll} h & ::= & \texttt{<name>} & \text{Host identifier} \\ q & ::= & \ldots \mid \mathsf{tableref}(x, h) \mid \mathsf{remote}(h, q) & \text{Queries} \end{array}$$

We use h to identify different host machines that are available for executing query operators. We also extend the syntax of SecQL queries. First, we change table references to identify the host that the table is initially deployed on. Second, we introduce placement markers $\mathsf{remote}(h, q)$ that signify that the resulting data of q is passed from the host it is evaluated on to h.

To account for the new syntax, we extend the denotational semantics. However, since the denotational semantics models the sequential behavior and is agnostic to distribution, the extension is straightforward:

$$[\![\mathsf{tableref}(x, h)]\!]_R(\rho) = \rho(x)$$
$$[\![\mathsf{remote}(h, q)]\!]_R(\rho) = [\![q]\!]_R(\rho)$$

Next we extend the information-flow type system from above for distributed SecQL. This extension is a little more involved because we need to propagate information about the permissions of available machines. We capture this information in a privilege context $P : h \mapsto A$ that assigns each available host a set of taint identifiers. Intuitively, if $P(h) = A$, then h may receive and process any data whose taint is in A. Using the privilege context, we define a new typing judgment $P, C \vdash q : T$ featuring the following two rules.

$$\frac{C(x) = T \qquad \mathsf{taint}(T) \subseteq P(h)}{P, C \vdash \mathsf{tableref}(x, h) : T} \quad \text{(T-Table)} \qquad \frac{P, C \vdash q : T \qquad \mathsf{taint}(T) \subseteq P(h)}{P, C \vdash \mathsf{remote}(h, q) : T} \quad \text{(T-Remote)}$$

Rule T-Table ensures through its second premise that the host h, on which the table is deployed on, has sufficient privilege. Since host h owns the table, we typically expect P to be selected such

$$\frac{}{P, C \vdash \mathsf{tableref}(x, h) \rightsquigarrow \mathsf{tableref}(x, h) \mid h} \quad \text{(Tr-Table)}$$

$$\frac{P, C \vdash q \rightsquigarrow q' \mid h}{P, C \vdash \mathsf{unOp}(q, e) \rightsquigarrow \mathsf{unOp}(q', e) \mid h} \quad \text{(Tr-UnOp-1)}$$

$$\frac{P, C \vdash q \rightsquigarrow q' \mid h \qquad C \vdash q : T \qquad h' \neq h \ \wedge \ T \subseteq P(h')}{P, C \vdash \mathsf{unOp}(q, e) \rightsquigarrow \mathsf{unOp}(\mathsf{remote}(h', q'), e) \mid h'} \quad \text{(Tr-UnOp-2)}$$

$$\frac{P, C \vdash q_1 \rightsquigarrow q_1' \mid h_1 \qquad P, C \vdash q_2 \rightsquigarrow q_2' \mid h_2 \qquad h_1 = h_2}{P, C \vdash \mathsf{binOp}(q_1, q_2) \rightsquigarrow \mathsf{binOp}(q_1', q_2') \mid h_1} \quad \text{(Tr-BinOp-1)}$$

$$\frac{P, C \vdash q_1 \rightsquigarrow q_1' \mid h_1 \qquad P, C \vdash q_2 \rightsquigarrow q_2' \mid h_2 \qquad h_1 \neq h_2 \qquad P, C \vdash q_2 : T_2 \qquad \mathsf{taint}(T_2) \subseteq P(h_1)}{P, C \vdash \mathsf{binOp}(q_1, q_2) \rightsquigarrow \mathsf{binOp}(q_1', \mathsf{remote}(h_1, q_2')) \mid h_1} \quad \text{(Tr-BinOp-2)}$$

$$\frac{P, C \vdash q_1 \rightsquigarrow q_1' \mid h_1 \qquad P, C \vdash q_2 \rightsquigarrow q_2' \mid h_2 \qquad h_1 \neq h_2 \qquad P, C \vdash q_1 : T_1 \qquad \mathsf{taint}(T_1) \subseteq P(h_2)}{P, C \vdash \mathsf{binOp}(q_1, q_2) \rightsquigarrow \mathsf{binOp}(\mathsf{remote}(h_2, q_1'), q_2') \mid h_2} \quad \text{(Tr-BinOp-3)}$$

$$\frac{P, C \vdash q_1 \rightsquigarrow q_1' \mid h_1 \qquad P, C \vdash q_2 \rightsquigarrow q_2' \mid h_2 \qquad P, C \vdash q_1 : T_1 \qquad P, C \vdash q_2 : T_2 \qquad h' \neq h_1 \wedge h' \neq h_2 \wedge \mathsf{taint}(T_1) \cup \mathsf{taint}(T_2) \subseteq P(h')}{P, C \vdash \mathsf{binOp}(q_1, q_2) \rightsquigarrow \mathsf{binOp}(\mathsf{remote}(h', q_1'), \mathsf{remote}(h', q_2')) \mid h'} \quad \text{(Tr-BinOp-4)}$$

Fig. 10. Algorithm: Select all secure deployments.

that this check succeeds. Rule T-Remote ensures through its second premise that the receiving host h has sufficient privilege to accomodate the data produced by q. Finally, except for T-Table, we adopt all type rules from Section 3.3 to the new judgment $P, C \vdash q : T$ by simply passing down the privilege context P alongside C.

4.2 Placement Space Reduction Algorithm

We formalize the placement space reduction as query transformation that introduces placement markers. The input is an operator tree. The query transformation algorithm outputs a set of possible placements that satisfy the taint constraints. These candidate placements are later used by an optimization phase to find the optimal placement.

We write $P, C \vdash q \rightsquigarrow q' \mid h$ to indicate that q is transformed to q' given contexts P and C as above. The result q' contains a placement marker remote whenever data crosses from one host to another, and q' itself is deployed on host h. We require contexts P and C during placement in order to call upon the typing judgment for queries when joining streams.

Algorithm. Figure 10 shows the algorithm. The rule Tr-Table retains a table on the host h, on which it is deployed. Next, we have rules for each unary operator unOp in SecQL, i.e., unOp $\in$ {select, project, declass}. The rule Tr-UnOp-1 only applies the placement rule recursively. This means that the placement of the unary operator is the same as its child query. It is safe to deploy the operator on the host h of the subquery q, because selection, projection, and declassification do not introduce additional taints. Rule Tr-UnOp-2 states that we can also change the host if there is at least one host h' that has sufficient privilege. Finally, we have four rules for each binary operator binOp $\in$ {crossProduct, union, intersect, difference}. The rule Tr-BinOp-1 states that if both children q_1 and q_2 are on the same host, the binary operator can also be placed on this host. The rule Tr-BinOp-2 checks the case that the children are on different hosts, but the host of q_1 has the privilege to access the data of q_2. In this case, we can add one remote link from the host of q_2 to the host of q_1. The rule Tr-BinOp-3 is analogous to Tr-BinOp-2, but covers the case where the

host of q_2 can read the data of q_1. The last rule Tr-BinOp-4 can be applied when we can find a host h' that has privilege to access q_1 and q_2 but hosts neither of those queries. In this case, we can place the binary operator on the host h' and add remote links for both children. Operator placement fails in case no host can be found that has sufficient privilege. Note that, in the algorithm, the client is treated just like another host.

4.3 Properties

We first introduce some preliminary definitions concerning subqueries and their deployment.

Definition 4.1. A query q_1 is a (direct) *subquery* of a query q_2 iff
$q_2 = \text{select}(q_1, e)$, $\text{project}(q_1, e)$, ..., $\text{declass}(q_1, A)$. $Sub(q)$ indicates the direct subqueries of q. $Sub(q)^*$ is the set of direct and indirect subqueries of q, i.e., the reflexive transitive closure of the Sub relation.

A deployment d defines a placement host for each subquery of a query q. The symbol $\bot$ indicates an undefined placement.

Definition 4.2. A *deployment* $d : q \rightarrow h$ of a query q is a mapping $[q_i \mapsto h]$ such that $\forall q_i \in Sub(q)^*, d(q_i) \neq \bot$.

In a *valid* deployment, the initial placement of source tables is respected in the placement of the rest of the query and the deployment of (sub)queries is consistent with the hosts that are syntactically specified in the query via $\text{remote}(h, q_i)$ terms.

Definition 4.3. A deployment $d : [q_i \mapsto h]$ *is valid for a query* q iff
if $q = \text{tableref}(x, h)$, then $d(q) = h$ (i)
if $q = \text{remote}(h, q_1)$, then $d(q) = h$, $d(q_1) = h_1$, and d is valid for q_1 (ii)
otherwise, $\forall q' \in Sub(q).\ d(q') = d(q)$ and d is valid for q' (iii)

The security property ensures that a (sub)query is never deployed on a server that does not have sufficient rights to access the data.

Definition 4.4. A deployment d of a query q is *secure according to a privilege context* P iff $\forall q_i : T \in Sub(q)^*.\ \text{taint}(T) \subseteq P(d(q_i))$

We can now define a sound deployment. Soundness ensures that a deployment is both consistent with the structure of the query and secure.

Definition 4.5. A deployment d is *sound for a query* q *according to a policy* P iff it is valid and secure.

It makes sense to distinguish cases for which a sound deployment can be found:

Definition 4.6. Policy P is *feasible for a query* q *in* C if
$\forall q' \in Sub(q)^* \quad P, C \vdash q' : T \quad \exists h.\text{taint}(T) \subseteq P(h)$.

We first show that the transformation in Figure 10 preserves the type of the queries.

Theorem 4.7. *Let q, q' be queries such that $P, C \vdash q : T$ and $P, C \vdash q \rightsquigarrow q' \mid h$. Then $P, C \vdash q' : T$*

We state the theorem ensuring that the transformation in Figure 10 produces a sound deployment.

Theorem 4.8. *If for a query q, $P, C \vdash q : T$, a deployment produced by the transformation in Figure 10 is sound.*

Next, we look at a completeness property of the deployment algorithm:

Theorem 4.9. *If P is feasible for a query q, $P, C \vdash q : T$, then the algorithm in Figure 10 produces a deployment d.*

Another requirement for the placement algorithm is that the transformed query evaluates to the same result as the original query:

Theorem 4.10. *Let q, q' be queries such that $P, C \vdash q : T$ and $P, C \vdash q \rightsquigarrow q' \mid h$. Then it holds $[\![q]\!]_{raw(T)}(\rho) = [\![q']\!]_{raw(T)}(\rho)$*

5 COST MODEL OPERATOR PLACEMENT

In this Section, we describe the second phase of our operator placement mechanism. We use the result of the placement space reduction (Section 4) to narrow down the set of candidate placements to a single one, based on metrics optimization. To find the optimal solution for the placement problem, we formulate placement as a constraint satisfaction problem (CSP) (Section 5.1) augmented with a cost function (Section 5.2).

CSPs are modeled by constraint networks. These are 3-tuples (V, D, C), where V is a sequence of decision variables, D is a sequence of domains for the respective variables, and C is a set of constraints. A solution of a CSP assigns values from the domains D to their respective variables V such that all constraints in C are satisfied. A constraint optimization problem (COP) augments the constraint network by a cost function that needs to be minimized while still satisfying the corresponding CSP.

5.1 Placement Constraints

Let q be a SecQL query with privileges P and environment C. We start by adding placement constraints derived from the placement space reduction to the CSP. First, we identify each host $h \in H$ where H is the set of available hosts for q. Let O be the set of all operators that are present in q. We introduce a decision variable $op\text{-}var_o$ for each operator $o \in O$. The value of $op\text{-}var_o \in H$ is the host h on which the operator o is placed. From the placement space reduction phase, we obtain a set of possible placements for q: For each operator $o \in O$, the placement space reduction phase provides a set of candidate hosts H_o on which o can be placed:

$$H_o = \{h \in H \mid P, C \vdash o \rightsquigarrow o' \mid h\}$$

For each operator o, we add a constraint to enforce that it can only be placed on host h that is in the corresponding set of candidate hosts H_o.

$$\bigvee_{h \in H_o} op\text{-}var_o = h$$

As placement space reduction complies with the security constraints, the optimal placement is secure as well.

5.2 Cost Model

Bandwidth. First, we consider minimizing the total bandwidth used by the query. For that, we define the *selectivity* of an operator. The selectivity is the data that it produces relative to the data it consumes. For example, a selection operator with selectivity of 0.5 outputs half as much data as it consumes. Selectivity has been used for placement algorithms, e.g., by Zhou et al. (2006). We statically approximate the selectivity sel_o based on the assumption that the groups of input data for an operator are equally sized, e.g., $sel_o = 0.5$ for selection operators. We use the selectivity to define the data that is sent over an outgoing link of an operator o relative to other operators:

$$\mathsf{data}(o) = \begin{cases} 1.0 & \text{if } o = \mathsf{tableref}(x, h) \text{ for some } x, h \\ sel_o \cdot \mathsf{data}(o') & \text{if } o = \mathsf{unOp}(o') \text{ or } o = \mathsf{remote}(h, o') \text{ for some } h \\ sel_o \cdot (\mathsf{data}(o'_1) + \mathsf{data}(o'_2)) & \text{if } o = \mathsf{binOp}(o'_1, o'_2) \end{cases}$$

For simplicity, we assume that the bandwidth of an outgoing link of a table is 1.0. For unary operators, the bandwidth is the input bandwidth multiplied by the selectivity of the unary operator – binary operators are defined similarly.

We refer to the directed edges in the operator graph as *links*. Network bandwidth is consumed by links between operators, which are placed on different nodes. Let $(o, o') \in L$ be a link from operator o to operator o', and L be the set of all links. Network bandwidth is only used when the two operators o and o' of a link (o, o') are placed on different hosts, i.e., $op\text{-}var_o \neq op\text{-}var_{o'}$. We define the set B of all links in query q that are between two different hosts.

$$B = \{(o, o') \mid op\text{-}var_o \neq op\text{-}var_{o'}\}$$

The bandwidth cost $cost_{bw}$ for a query q is defined as the sum of the data sent over all links that connect two different hosts for a deployment of q.

$$cost_{bw} = \sum_{(o,o') \in B} \text{data}(o)$$

Prioritized Placement. The next approach optimizes performance by placing operators on the available host with the *best processing capabilities* adopting the model from Cardellini et al. (2017). The *processing capability* $\text{cap}(h)$ of a host h is a synthetic measure of its overall processing possibility and it is, in the rest of the paper, assumed as part of the system specifications or obtained experimentally, e.g., by monitoring. The function $\text{cap} : H \to \mathbb{R}_{\geq 1}$ maps hosts to their capabilities; higher values correspond to higher capabilities, e.g., if $\text{cap}(h_1) = 1.0$ and $\text{cap}(h_2) = 2.0$, h_2 is twice as fast as h_1. When only considering processing capabilities, then the optimal host for an operator o is the host that maximizes $\text{cap}(op\text{-}var_o)$. We combine multiple operators by summing them up and use the inverse to transform the model into a minimization objective:

$$cost_{prio} = \sum_{o \in O} \frac{1}{\text{cap}(op\text{-}var_o)}$$

CPU Load Distribution. Another cost function aims to equally distribute the CPU load over the available hosts. We define the cost of executing an operator o as $load_o \in \mathbb{R}_{\geq 0}$: the bigger $load_o$, the more expensive it is to execute operator o. As the load of an operator is proportional to the amount of data it processes, we define $load_o = c_t \cdot \text{data}(o)$ with a coefficient c_t for the operator type t and the estimate for the amount of processed data. We define the load on a host h as:

$$load_h = \sum_{o \in O \mid op\text{-}var_o = h} load_o$$

Fair load distribution is achieved when the assigned load $load_h$ is proportional to each host processing capabilities $\text{cap}(h)$. Overloaded hosts shall be penalized with a penalty that must be higher than the benefit achieved for host under-loading. We achieve this result by squaring on the ratio of $load_h$ and $\text{cap}(h)$. Thus, the total CPU load cost is:

$$cost_{load} = \sum_{h \in H} \left(\frac{load_h}{\text{cap}(h)} \right)^2$$

Intuitively, each summand is a measure of the deviation from the optimal load placed on a host. The sum for all operators in a query is a measure of the unfairness of CPU utilization.

The formulas above exhibit different properties concerning the dependency between different operator placements. $cost_{bw}$ models dependency between distinct operator placements, because the value $\text{data}(o)$ depends on the placement of all operators that generate traffic on the link $o, o' \in O$. In $cost_{load}$, operator placement decisions depend on each other as well, because processing power is treated as a resource that is consumed by the execution of operators. So, placing an operator on a node, lowers the probability that another operator is placed on the same node. Dependent

placement decisions are more accurate, but the explanation of a placement decision requires one to consider the whole placement plan. In contrast, independent placement decisions can be explained to users more easily. $cost_{prio}$ models this objective independently by assuming that a placement candidate h with the best capabilities $\text{cap}(h)$ executes an operator the fastest.

5.3 Total Cost

As total cost, we propose 4 combinations of the costs above, combining $cost_{bw}$ with $cost_{load}$ or $cost_{prio}$ once as a product and once as weighted sum. Any of these functions is used as the cost function in the COP and thus gets minimized by the constraint solver. These cost functions combine the objectives to (1) reduce network load in the case of $cost_{bw}$, and (2a) distribute CPU load in the case of $cost_{load}$, or (2b) place by priority in the case of $cost_{prio}$. Together they open up a two dimensional comparison of placement decision, which relates to different application scenarios:

$$cost_{load\Sigma} = \alpha \cdot cost_{bw} + \frac{1}{\left(\sum_{o \in O} load_o\right)^2} \cdot cost_{load} \quad (\Sigma\text{BL}) \qquad cost_{load\Pi} = cost_{bw} \cdot cost_{load} \quad (\Pi\text{BL})$$

$$cost_{prio\Sigma} = \alpha \cdot cost_{bw} + \frac{1}{|O|} \cdot cost_{prio} \quad (\Sigma\text{BP}) \qquad cost_{prio\Pi} = cost_{bw} \cdot cost_{prio} \quad (\Pi\text{BP})$$

Dependency among placement decisions. First, we consider functions ΣBL and ΠBL, which use $cost_{load}$ to capture the dependency among operator placement decisions. $cost_{load}$ models processing power as resource consumed by operators and therefore offers a model, which reduces local CPU bottlenecks. In contrast, functions ΣBP and ΠBP, which use $cost_{prio}$, model operator placement decisions to be independent from each other by omitting the reduction of processing capabilities by operator execution. In turn, they are easier to apply in scenarios, where other objectives than CPU load and network bandwidth are also relevant, e.g., if memory resources of nodes are very heterogeneous.

Priority of objectives. In functions ΣBL and ΣBP, we combine the two considered objective combinations with first priority on the bandwidth optimization. Therefore, the resulting placement has always minimal bandwidth consumption according to $cost_{bw}$. The second priority objective – either $cost_{load}$ or $cost_{prio}$– has only impact on deciding which of the placements with minimal network bandwidth is chosen. This is achieved by normalizing the second priority objective to $[0, 1]$ and defining a coefficient $\alpha \gg 1$ for the bandwidth objective. In settings with fast network and low or very heterogeneous CPU resources, e.g., in computing clusters or in mixed IoT and cloud scenarios, placement based on the bandwidth objective might cause deployments with CPU bottlenecks. ΠBL and ΠBP address this issue by assigning equal priority between the two optimization objectives. The multiplication enables the combination of objectives with different ranges, and was successfully used for placement before, e.g., by Pietzuch et al. (2006).

6 IMPLEMENTATION

We implemented SecQL as a domain-specific language embedded in Scala. SecQL is built on top of i3QL (Mitschke et al. 2014), which provides syntax for SQL-like queries, local incremental data processing and relational algebra optimizations. The syntactical correctness of queries is enforced by the Scala type system. This applies also to expressions for selections and projections, which are Scala functions. i3QL uses lightweight modular staging (LMS) (Rompf and Odersky 2012) in order to inspect and edit functions. LMS also provides further optimizations, such as common subexpression elimination. In SecQL, we use the relational algebra trees generated by i3QL and distribute the operators, thus gaining benefits from all optimizations.

Privacy and Distribution. Compared to i3QL, SecQL introduces permission labels and privacy taint identifiers, which are represented by Scala objects. These objects are treated as types in the SecQL query compilation step – which happens at run time because of staging. The implementation of the information-flow type system tracks the propagation of taint information in the query. Eventually, host permissions and operator taint information are compared to check that there are sufficient permissions to execute an operator on a host. Moreover, we added optimizations specific to the privacy preserving placement to i3QL. For example, we reorder joins and cross products so that operators with the same labels are grouped together. This optimization allows us to reduce the number of remote links, because operators with the same label can all stay on the same server. For this optimization, we use the property that joins, cross products, unions and intersections are commutative and associative when the order of the elements in the resulting tuple is corrected after a reordering.

We extend i3QL to compile to Akka Actors and Akka Streams (Akka 2019). All groups of one or more connected operators in the relational algebra tree extracted from a query, which are placed on the same node, are wrapped by an Akka Streams operator and deployed inside an actor. The communication among actors is based on Akka Streams, which provide asynchronous messages over TCP/IP and back-pressure control. As there are no run-time privacy checks, labels do not have a run-time representation. However, we extended i3QL to support the additional syntax for privacy introduced in Section 2.

Constraint-Based Optimization. We implemented the constraint-based optimization using JaCoP (Kuchcinski and Szymanek 2017), a constraint solver library for Java which has already been used for operator placement (Thoma et al. 2014). JaCoP supports a variety of constraints and solving algorithms, enabling us to model constraints both in the domain of finite integers and floating point numbers. The CSP solver works on a simplified query graph, where nodes model operators and weighted edges model the links between operators. Operators are only defined by their selectivity, a load property and node placement constraints.

7 EVALUATION

The evaluation answers the following research questions: (1) What is the performance of the privacy-aware operator placements generated by SecQL? (2) How does the performance regarding run time and system resource usage change when using different cost functions? (3) What is the impact of an accurate system specification on the performance?

All tests are performed in the cloud with a Docker container for each node plus a container for a controller to manage the test execution and to collect measurement data. We use Amazon Web Services (AWS) Fargate (AWS 2019). The containers in the case study in Section 7.1 and in the homogeneous setup S_h of the benchmark from Section 7.2 are executed on Intel Xeon E5-2670 v2 CPUs at 2.5 GHz (AWS EC2 M4). In the variable benchmark setup S_v, the containers are hosted on Intel Xeon E5-2686 V4 CPUs at 2.3 GHz machines (AWS EC2 M3). We use different machines for the setups due to Amazon Fargate automatically selecting instance types.

To measure the run time of a query, we take the wall-clock execution time to compute all results of the query. We also record CPU time and memory consumption as the difference of the respective measurements before and after query execution. We warmup the JVM processes by executing the query once (i.e., for 1M result tuples) and resetting the system state before the measurement.

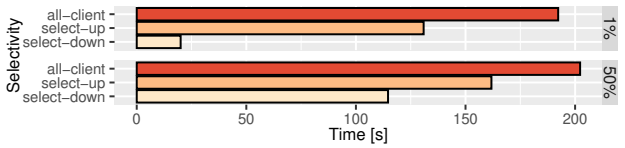


Fig. 11. Query run times for the hospital case study.

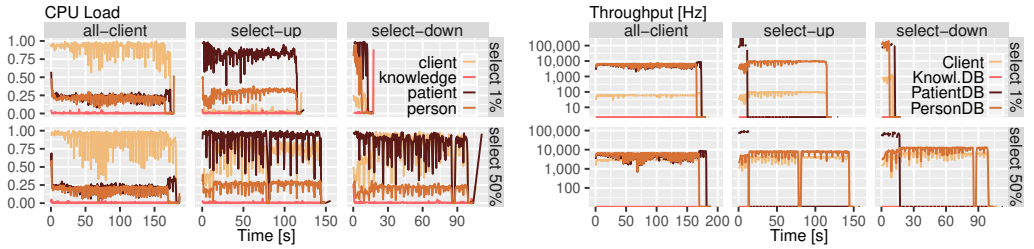


Fig. 12. CPU load and throughput for the hospital case study.

Fig. 13. Memory consumption for the hospital case study.

7.1 Hospital IT Case Study

In this Section, we study how placement decisions of SecQL affect performance in a realistic case study. We implemented a simple hospital IT system (the query in Section 2 is from this case study) based on the well-known HELP hospital information system (Gardner et al. 1999), originally developed at the Department of Medical Informatics, University of Utah, and in routine use at the Hospitals of Intermountain Health Care (IHC) in Utah, USA (OpenClinical 2004). To demonstrate the effect of pushing selections to table nodes, the query is extended with a selection to filter tuples in PersonDB. We evaluate the cases where the filter passes through 1% and 50% of tuples, which we call 1%-selectivity and 50%-selectivity, respectively. We also evaluate three types of operator placements:

select-down pushes selections down to the table hosts.
select-up pushes selections to the hosts that operate on the selection results.
all-client places the entire query on the client (only tables remain on their original hosts).

Each node is deployed into a dedicated container with 2 vCPUs and 4 GB memory on the AWS EC2 M4 machines described above. Figure 11 shows the query run time for each placement and filter selectivity (shorter is better). As expected, run time is shortest for select-down. In the case of 1%-selectivity, select-down reduces run time by 89.58% compared to all-client. In the case of 50%-selectivity, select-down reduces run time by 43.32%.

Figure 12 (left) shows the CPU load of the nodes. Both select-down and select-up significantly reduce CPU load on the client (lower average load and shorter run time). In contrast, the load on the patient node increases because of increased serialization. Yet, data locality and local selection before join show improvements in select-up compared to all-client. Similarly, the required CPU time

for 1%-selectivity is much lower than for 50%-selectivity with the select-down placement, because fewer tuples are processed, transmitted and serialized. Figure 12 (right) shows the throughput at the input tables and result relation. The high performance of local selection in case of select-down is confirmed by a comparatively high throughput at PatientDB and PersonDB inputs. Overall the throughput graphs show roughly constant and time invariant processing speed of SecQL.

Figure 13 displays memory consumption. The measurements of settled memory consumption before and after the execution memory consumption at the client is lower for select-up and select-down by factors 2.3 and 2.4 for 50%-selectivity and factors 75.9 and 92.1 for 1%-selectivity. The overall memory growth increased by less than 10% for 50%-selectivity, which is caused by additional tuple copies on other nodes compared to the all-client placement. However, for 1%-selectivity, the impact of these additional tuple copies is far less, so that an overall reduction of 31.5% and 33.3% in memory growth was measured. We also display minimum and maximum memory consumption (right) – the actual consumption varies in the range between these lines because of periodic garbage collection. The graph visualizes the decreased memory consumption on the client node for select-up compared to all-client, which is even lower for select-down. Overall, memory consumption grows linearly.

Discussion. The case study analyses the impact of operator placement decisions and operator selectivity. The results show that concerning system throughput, CPU load and memory consumption, good operator placement decisions improve the performance of a query in SecQL significantly. Early reduction of the amount of data leads to better performance. Finally, SecQL resource consumption exhibits a linear performance behavior.

7.2 Performance Benchmark

To evaluate the performance of SecQL in detail, we implement a number of queries, which we describe in the following and refer to as the *company benchmark*. All queries are based on the data model of a hypothetical manufacturing company. We use similar source data as in the industry-standard TPC-H database benchmark (TPC 2019), such as parts, line items, and suppliers. Data is organized into 9 tables with 4 different security domains; each query accesses at least 2 different security domains. We do not use the original TPC-H benchmark, because it does not have security data classification nor is it suitable for distribution and event processing. The data schema exhibits a high degree of dependency among relations, making it hard to represent the data as independent events occurring at various places in the distributed system, while ensuring referential transparency. Therefore, we changed the schema by removing some of the dependencies. The benchmark queries cover all SecQL operators, and test the following *extreme cases* (Flyvbjerg 2006): (i) combining many source relations; (ii) combining aggregation, declassification, and nested queries; (iii) queries with a large number of selections; (iv) queries yielding a large number of result tuples. In addition to these cases, our benchmark suite includes typical business-oriented ad-hoc queries (TPC 2019). Specifically, query Q1 selects the components of a product via a join of 3 tables; only two updates are propagated to the result. Q2 contains multiple joins distributed across multiple hosts. Q3 combines 7 sources and requires declassification. Q4 contains multiple selections. Q5 is a nested query: the nested query performs an aggregation followed by a declassification. Q6 computes a cross product with many entries. Q7 evaluates the `EXISTS` operator, which is compiled to a join. Q8 uses the `UNION` operator. Q9 uses the `INTERSECT` operator. Q10 stores the result on the client.

Placement and Setup. For each query, Table 1 shows the number of all possible placements without considering privacy, and the number of placement candidates after deployment space reduction (i.e., all possible privacy-aware placements) generated by our system. These placement candidates define the search space of the CSP applied in the second phase of the placement algorithm.

Table 1. Placement candidates of the company benchmark.

Query	Q1	Q2	Q3	Q4	Q5	Q6	Q7	Q8	Q9	Q10
All Possible Placements	625	3,125	1,220,703,125	15,625	3,125	625	15,625	3,125	390,625	625
Privacy Preserving	40	4	8,192	160	80	625	1,000	32	256	16

For each query, we evaluate three different operator placements:

$\mathbf{P_{client}}$ *Client-only placement.* All operators are on the client. Only the tables are on a remote host.
$\mathbf{P_{optimal}}$ *Placement without privacy.* Operators are placed according to a cost model from Section 5.
$\mathbf{P_{private}}$ *Privacy preserving placement.* Same as $P_{optimal}$, but with privacy constraints from Section 4.

P_{client} is the comparison baseline, because we are not aware of any more performant system that is close enough to SecQL to allow a sensible comparison. Related work either considers batch processing rather than event processing or does not consider privacy constraints. Zeng et al. (2015) chose this baseline for the same reason. To indicate the performance impact of our privacy enforcement, we compare also with $P_{optimal}$.

We consider two different setups (Table 2). In the *homogeneous setup* S_h, all nodes have the same processing power. In the *variable setup* S_v, the nodes have different numbers of vCPUs and sufficient memory while the client has the least resources. To evaluate the impact of our placement algorithm and privacy constraints relative to the naive solution with all operators on the client, we normalize measurements (run time, memory consumption, total CPU time) for $P_{optimal}$ and $P_{private}$ such that the measurement for P_{client} equals 1.0 in each case. Run time in Figure 14 measures the wall-clock execution time to compute all results of a query, total CPU time in Figure 15 (in vCPU usage) and memory consumption in Figure 16 are sums over all nodes. Each graph shows a cost function.

Table 2. Performance benchmark setup.

	Homog. Setup (S_h)		Variable Setup (S_v)	
Host	vCPUs	memory	vCPUs	memory
public	2	4 GB	1	4 GB
production	2	4 GB	4	8 GB
purchasing	2	4 GB	2	4 GB
employees	2	4 GB	1	4 GB
client	2	4 GB	0.5	4 GB

Results. In most cases, all cost models achieve placements that reduce the run time and total CPU load. Memory usage increases moderately compared to P_{client}. The average results differ for the two setups: in S_h run time reduces by 43.6% (SD 31.7%), CPU time by 40.0% (SD 33.8%) and memory growth by 0.5% (SD 8.7%), while in S_v run time reduces by 52.0% (SD 34.3%), CPU time by 7.2% (SD 58.1%) and memory growth by 0.5% (SD 8.9%). These results show that SecQL provides on average an improvement over P_{client}, independent of the cost model. The performance of P_{client} in S_v has been worse than in S_h, because the client node only has a fourth of the processing capability. This explains the higher performance gain for S_v which is achieved by placing operators on other nodes. To inspect the behavior of each query – beyond aggregate results – we consider the two dimensions of combined objectives (bandwidth and load vs. bandwidth and priority) and objective priority (Σ against Π):

For queries (Q1, Q2, Q4, Q7 and Q8) decisions based on ΣBL or ΠBL perform better than ΣBP or ΠBP, because the CPU load is distributed more uniformly. This finding is especially relevant for S_h. The opposite holds only for Q3, where the distribution of load leads to more data transfer over network, which results in worse performance. Yet, here also $P_{private}$ performs better than $P_{optimal}$, because of the imprecision of selectivity values. We expect that, with more precise values, ΣBL or ΠBL would perform better. Nonetheless, this case exemplifies furthermore that placement of ΣBP or ΠBP tolerates significant inaccuracies between real and estimated operator selectivity.

ΠBL and ΠBP placements, where objectives are equally prioritized, often lead to better results (Q2, Q4, Q7, Q10) than ΣBL and ΣBP, which prioritize network bandwidth reduction. Q9 is an exception

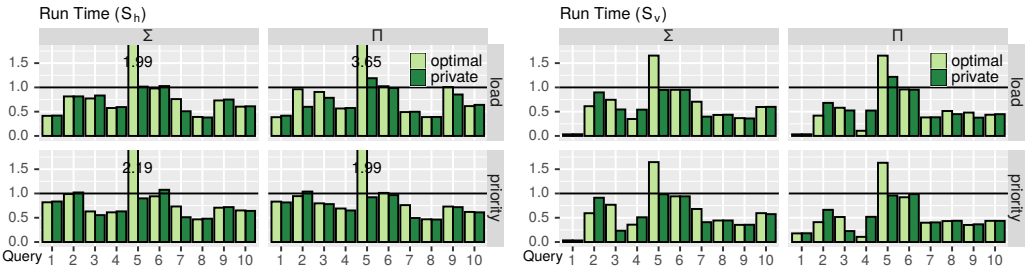


Fig. 14. Query run time of all nodes relative to client-only placement for company benchmark.

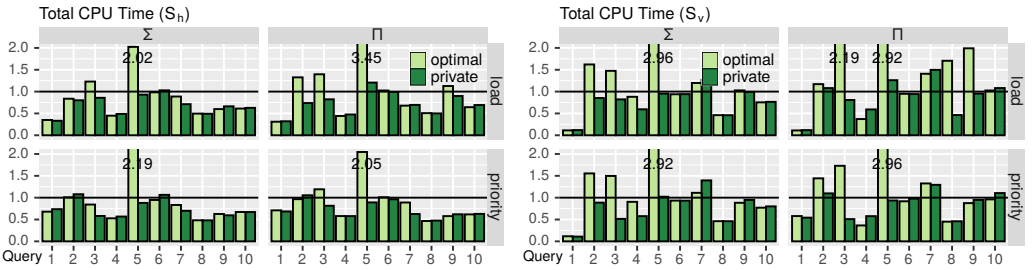


Fig. 15. Total CPU time across all nodes relative to client-only placement for company benchmark.

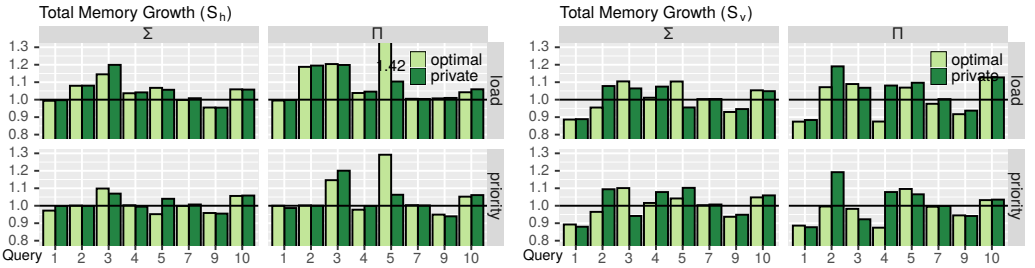


Fig. 16. Total memory consumption for all nodes relative to client-only placement (company benchmark). [2]

in S_h, because load distribution leads to higher network consumption. Also, ΠBL achieves less optimal results for Q2 and Q8, because the load distribution leads to the placement of operators on nodes which are neither source nor sink of the query. While this decision reduces CPU bottlenecks, in these cases, the serialization and transmission overhead has eventually negative impact on the overall performance. Also, ΠBP leads to worse results for Q1 and Q4, as the load distribution improves the performance less than additional network communication reduces it.

Discussion. The measurements show that SecQL's placement can reduce the run time by up to 97% and reduce CPU time by up to 89% compared to P_{client}. The memory consumption of the entire system during execution stays in a similar range compared to the one for client-only placement. As expected, the evaluation shows that our performance model is sometimes not perfectly accurate. This is the case especially in Q2, Q3, Q5 and Q7, which show better performance for $P_{private}$ than for the non-constrained counterpart $P_{optimal}$. These cases exhibit the largest difference between estimated and real operator selectivity, impacting the accuracy of bandwidth for all cost functions and for the operator load model for ΠBL and ΣBL. This relation is therefore examined next.

[2] Q6 and Q8 are excluded, because memory consumption does not grow significantly for their execution.

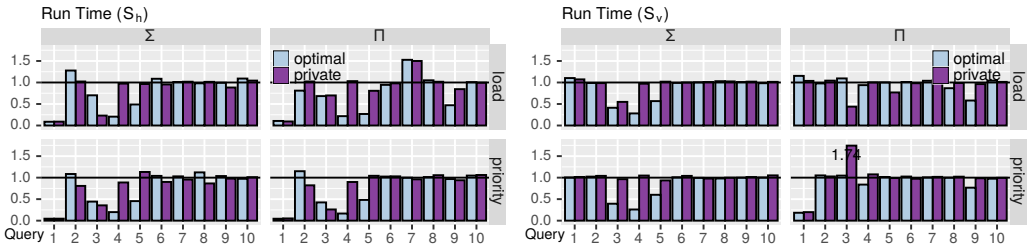


Fig. 17. Run time: precise selectivity values relative to estimated selectivity values (company benchmark).

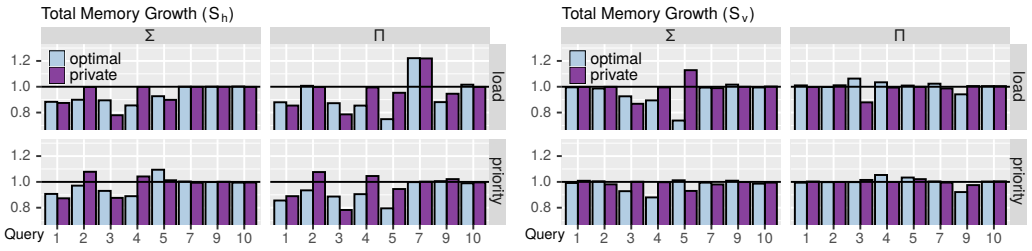


Fig. 18. Memory consumption: precise selectivity relative to estimated selectivity (company benchmark). [2]

7.3 Operator Selectivity

SecQL relies on a system specification, which contains information about the available resources of a node (e.g. memory or CPU) and operator selectivity, for making optimal placement decisions. In many application scenarios, the resources of nodes are known, but precise operator selectivity values have to be estimated *a priori*. As shown in the previous Sections, a simple estimation suffices to generate an efficient deployment. In this Section, we evaluate the effect of precise selectivity obtained via monitoring. We assess the impact of operator selectivity values for the company benchmark comparing the results of the estimate and the results with the monitored values.

Figure 17 and Figure 18 show run time and memory consumption for the company benchmark with precise selectivity values relative to the estimated values from Section 7.2. The improvements are influenced by the setups, but minimally by the cost function. The run time decreases for S_h on average by 27% (SD 40%) and for S_v by 9% (SD 26%). In S_h twice as many executions are faster compared to estimated selectivity values; in S_v the numbers are equal. Figure 17 shows that the improvement is often significant, while the few changes for the worse are rather small. All tests but the $P_{optimal}$ placements of Q5 with ΠBL and ΠBP in the S_v setup are similarly fast or faster than the client-only placement. Similar considerations apply to the total CPU time – which we omit. Memory consumption exhibits no difference among cost functions, but does between setups. The measurements for S_h show a reduction of memory growth on average by 5% (SD 9%) and for S_v by only 1% (SD 5%). In S_h the memory growth is reduced 3-times more often than increased and twice as often reduced in S_v.

Discussion. Our experiments show that precise selectivity values improve the performance of placements for all cost functions. This is explained by the fact that all cost functions, to different extents, include $cost_{bw}$, which is heavily influenced by selectivity values. In a few cases, performance slightly degrades, because our models are approximate, e.g., the definition of $data(o)$ assumes that input relations produce similar numbers of events. The results also show that the impact of the precision depends on the setup configuration. This has to be noticed in the context that, with estimated values, placement achieves better results in S_v than in S_h (Section 7.2), hence the room for improvement of run time is bigger in S_h, with precise selectivity.

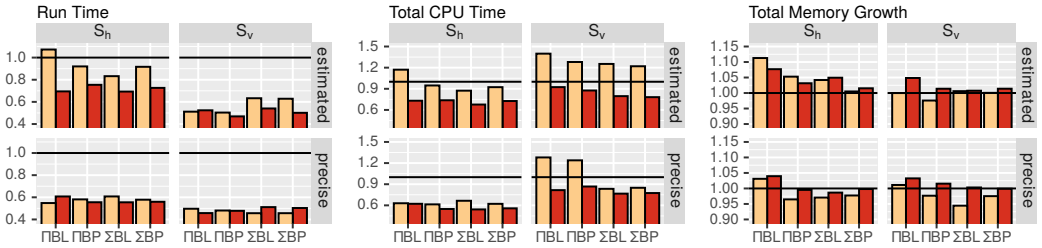


Fig. 19. Average performance over all company benchmark queries divided by cost function, placement (orange: $P_{optimal}$, red: $P_{private}$), setup and selectivity value precision relative to P_{client}.

7.4 Evaluation Summary

Figure 19 shows an average comparison among the cost model functions in all tested setups. The performance for the private case is often similar to the optimal case. The total required CPU time of a deployment heavily depends on the amount of data sent over the network. Therefore, $P_{private}$ requires mostly less CPU time than $P_{optimal}$ because of the privacy restrictions, which lead to less data transmission between hosts. The measurements confirm that network transmission is in most cases the reason for higher run times, even if in all tests the network is fast. For worse connections, e.g., over the Internet, this factor would increase even more. For this reason, in many cases, the bandwidth reduction without any additional objective leads already to comparatively good placements as ΣBP in S_h shows. The similar average performance over all queries for all cost functions confirms that – otherwise we would see a significant difference between ΠBP and ΣBP in S_h (model reduction to bandwidth reduction only) compared to the other tests. Yet, the difference among queries might be significant: ΠBL achieves better load distribution when simple bandwidth reduction leads to local CPU bottlenecks. The average results show that the precision of operator selectivity has strong impact on our model's performance.

When operator selectivity is uncertain, the cost function ΣBP generally leads to better placements with decisions that are easier to understand. On the other hand, ΠBL or ΣBL lead to more efficient placements, e.g., if local CPU bottlenecks are present. ΠBL and ΠBP, model equal objective priority. They should be applied with care if the selectivity values are estimated. These cost functions might compromise too much on the central bandwidth objective, hindering performance.

Although our models do not guarantee best performance in all cases, we consider it as a more realistic placement solution than a more accurate performance model. We believe that approximate models are more likely to be applied in practice, since a perfect model becomes inaccurate as soon as the system evolves.

8 RELATED WORK

Privacy in Big Data. Recent research has addressed the issue of confidentiality in big data systems. An overview of the most recent development is given by Derbeko et al. (2016).

Sen et al. (2014) propose a workflow for enforcing privacy policies in big data systems. Similar to our proposal, users specify privacy requirements for data streams and access rights for nodes. However, the goal is not to find a proper placement but check that the big data processing system does not violate a policy. GUARDMR (Ulusoy et al. 2015) is a framework that enforces security policies at the key-value level in MapReduce (Dean and Ghemawat 2010). Authorized views of the target dataset are expressed in the object constraint language (OCL). The implementation of the policies is automatically generated and integrated in the system via aspect weaving using AspectJ. In contrast to our work placement is defined by the MapReduce scheduler and is not driven by privacy enforcement.

A number of systems have been designed to operate both on the private cloud for security reasons and on a public one to improve performance in the case of non sensitive data. HybrEx (Ko et al. 2011) assigns MapReduce computations to hosts based on data sensitivity. In HybrEx, data is divided into sensitive and non-sensitive data, which are sent to public clouds, respectively, private cloud. Sedic (Zhang et al. 2011) automatically partitions MapReduce jobs by following security levels of data and distributes jobs between private and public clouds. Code analysis and transformation is used to reduce communication between the public and the private cloud. SEMROD (Oktay et al. 2015) overcomes Sedic's limitation of shifting all reducers to private machines employing schedulers that can assign jobs on non-sensitive data to the public cloud even in the reduce phase.

Several approaches exploit partial homomorphic encryption to enable query processing on encrypted data in the cloud (Popa et al. 2011; Tetali et al. 2013). Others (Arasu et al. 2013; Bajaj and Sion 2014) use trusted hardware to process database queries in a protected environment. These approaches are promising, but performance overhead and/or costs of trusted hardware are very high, which makes the privacy-aware operator placements a valuable alternative to consider. MONOMI (Tu et al. 2013) addresses performance issues with homomorphic encryption by introducing split client/server execution of complex queries: as much of the query as is practical over encrypted data is performed on the server, the remaining components are performed on a trusted client, which decrypts data and processes queries normally. None of the above systems support information-flow analysis or placement for aggregated data belonging to multiple security classes and only separate between public and private data.

The ideas above are also applied to the evaluation of distributed SQL-like queries: Oktay et al. (2017) split an acyclic query Q on private and public data in a hybrid cloud into the subqueries Q_{priv} and Q_{pub}. These subqueries are evaluated independently, but the system attempts to execute the maximum amount of work of Q with Q_{pub} which only requires public data, allowing its execution on a cheap and scalable public cloud. In contrast, Q_{priv} has to be evaluated on expensive trusted resources. Finally, the results are merged to obtain the result of Q. This approach only supports two security classes which can be defined for relation attributes or sets of tuples. While Oktay et al. (2017) consider only two privacy levels, Zeng et al. (2015) enable defining policies pairwise between multiple authorities, supporting a query plan generation algorithm for distributed multi-party queries. The algorithm generates optimized distributed queries, enforces the necessary access restrictions and also keeps the authorizations themselves confidential. Other work focuses on centralized query planning with attribute-level authorizations for distributed multi-provider setups (De Capitani di Vimercati et al. 2017). The *only encrypted* authorization mode enables access to an on-the-fly encrypted version of data to support secure operations on encrypted data for authorities that are not fully trusted, e.g., in cloud scenarios. Authorizations are enforced after query optimization, by integrating them into the physical plan generation. Such separation of optimization and privacy enforcement can lead to non-optimal placement decisions. Therefore, Dimitrova et al. (2019) embed the privacy enforcement into the optimization phase. This approach achieves better placement decisions as well as reduced optimization cost through early pruning of query plans, which violate privacy constraints. In contrast to all the solutions above, SecQL leverages programming language techniques and enables reasoning about privacy preservation based on the syntactic structure of the query.

ZQL (Fournet et al. 2013) is a compiler for privacy-preserving data processing expressed in a subset of SQL. Server-side computations are performed at the client that owns the data and the integrity of the computations/results is ensured by cryptographic protocols such as zero-knowledge proofs. ZQL addresses an orthogonal problem to ours. All operators are placed in one host (the host that owns all the data sources) and no algorithm for privacy-aware distributed placement is

provided. On the other hand, our approach does not address the integrity of the distributed partial computations and could be combined with ZQL to do so.

Information-Flow Policies. Access control protects confidential data: documents and entities (physical person, user, process, etc.) are assigned a security class. Access to a resource is granted, depending on the matching of security classes. In Bell-LaPadula (Bell and LaPadula 1973), security classes are ordered and a user may only read documents with equal or lower security class than his clearance (no read up). As information can be leaked by copying secret data to unclassified documents available to users with low clearance, write down is also prohibited.

Denning (1976) has generalized the multi-level security model (i.e., an ordered set of classes) to a lattice. Values and variables are assigned a security class. Information flows from a variable x to a variable y whenever information stored in x is transferred to, or used to derive information transferred to, y (Denning and Denning 1977). Goguen and Meseguer have introduced the concept of noninterference to formally state that a program does not leak data (Goguen and Meseguer 1982). For any class, any information in a higher class should not interfere with any value in a lower class. Volpano et al. (1996) propose a minimal programming language with a sound secure flow type system: programs can be type checked for secure information flow ahead of execution. Types encode the security class of variables and the clearance of programming language constructs. Typing rules check that there are no prohibited flows.

Language-integrated Policies. JFlow (Myers 1999) applies these principles to Java. Values are statically labeled with a security class. The compiler can erase most of the security labels when translating to Java code, resulting in minimal run-time overhead. More recent work includes formalisation of dynamic flow policies (Broberg and Sands 2006), tracking data flow across the application-database interface (e.g., by Schoepe et al. (2014)) and policy agnostic programming (Yang et al. 2016) to factor out information policies allowing programmers to implement them only once instead of repeatedly check them across the code. Our approach builds on work in information flow control but enforces flow control via deployment on a distributed system rather than via dynamically or statically checking a policy.

Secure Program Partitioning (Zdancewic et al. 2001) is a method to enforce system-wide security policies. In this approach, the code and data of the computation are partitioned across the available hosts in accordance with a security specification. In a follow up work (Zheng et al. 2003), this mechanism in complemented with the use of replication to ensure integrity – compromising data requires to compromise all replicas. Fabric (Liu et al. 2017) is a fully decentralized distributed system that integrates information and computation from independent administrative domains with different security requirements. Fabric tackles a number of new problems compared to the approaches above, including mobile code, and distributed transactions over *mutually not trusting* hosts. In contrast to our approach, these systems do not target event streams and distributed SQL-like queries, they do not perform run-time optimizations (enabled by LMS in SecQL) and do not allow reasoning about optimal performance of alternative deployment strategies as our cost optimization phase does.

The work by Farnan et al. (2010) tackles the problem of leaking *query* information (rather than *data* information, like in our case). Query leaking can occur when a subtree is offloaded to other hosts. Because they receive a (sub)query, such hosts may gain information about the original query. We consider this work complementary to ours. Combining the two is a promising line of future work, as such combination would enable reasoning about data privacy and query privacy together.

Previous research has investigated mechanisms to reason about declassification. Zdancewic (2013) proposes to add to security labels "integrity labels" that specify a degree of trust in data to ensure

that the decision to do the declassification is *sufficiently trusted*. Cruz et al. (2017) propose a type-based approach to declassification in an object-oriented setting which integrates the solution above with OO programming, defining a simple notion of label ordering based on subtyping, supporting recursive declassification policies, and enabling modular reasoning for relaxed noninterference.

We believe that our type system can be easily extended with the additional labelling mechanisms above. The *placement space reduction* phase in our work would then not only generate candidate deployment where privacy is enforced, but also deployments with a sufficient trust level.

Operator Placement. The operator placement problem consists of finding the *best* host on which each operator should be deployed in a distributed system by maximizing a certain metric, such as throughput (Cugola and Margara 2013; Lakshmanan et al. 2008). Previous research proposed to distribute operators to achieve load balancing for streaming systems in a cluster (Cherniack et al. 2003). The PIER distributed database relies on an overlay network where operators are assigned to hosts via random selection (Huebsch et al. 2003). More recent approaches are based on an abstract representation of the network to account for latency between hosts in the model. Since latency becomes a large factor in the Internet (in contrast to dedicated clusters), this solution enables placement for streaming systems on world-scale networks. An example of such algorithms using decentralized, dynamic optimization decisions is (Pietzuch et al. 2006). Similar to our approach these systems adopt operators as the deployment unit. To reduce the load of the placement algorithm, Zhou et al. propose a coarser granularity (Zhou et al. 2006) and deploy *query fragments*, i.e., groups of operators. AdaptiveCEP (Weisenburger et al. 2017) enables developers to define their own placement strategies. In contrast to theses solutions and others (Tian and DeWitt 2003; Xing et al. 2005) we adopt static operator placement because we derive placement from privacy constraints which are statically defined.

Query Languages. While SecQL is limited to the essential relational algebra operators, many query languages, e.g., T-SQL for MS SQL Server, offer a number of additional features. Our language is easier to study and it is close to the formalization we propose in this paper. Yet, additional features can raise new issues. For aggregation operations, if these introduce closed sets on the stream, they can become applicable even without the need to send incremental updates to the result stream. Without incremental updates or closed sets, result information could never be released, because results might change when additional input information arrives. Procedural programming features are currently only available in the expressions for selection and projection and could be part of aggregation, too. For these features, the privacy level would be over-approximated in the current system. Results are only sent to hosts that have sufficient permissions for all input source data, even if the applied logic would permit safely reducing the privacy level of the results. SecQL could benefit here from other work, e.g., on disclosure-lattices-based analyses (Bender et al. 2014; Guarnieri et al. 2019). Another language feature to consider is recursion, which poses a conceptual challenge for the denotation of SecQL, because it requires the fixpoint semantics to be well founded.

9 CONCLUSION

In this paper, we presented a novel approach to reason about deployment, privacy, and optimization in distributed query processing systems based on programming language techniques for information-flow control. Our main contribution is a two-step operator placement algorithm. The first step, placement space reduction, is formulated as a program transformation that we prove complete and secure according to an information-flow type system. The second step, cost model placement, supports reasoning and optimization based on performance metrics. The evaluation shows that our approach is effective in improving the performance of a distributed application while preserving privacy requirements.

Appendix A PROOFS

A.1 Type-Preserving Transformation

Proof for 4.7. We use induction on the derivations of the placement algorithm (Figure 10).

Case Tr-Table. A table is transformed into itself, hence it has the same type.

Case Tr-UnOp1. From the rule, we know that $P, C \vdash \mathsf{unOp}(q, e) \rightsquigarrow \mathsf{unOp}(q', e) \mid h$ for some h. From the induction hypothesis, we know that $P, C \vdash q \rightsquigarrow q' \mid h$ is type preserving. The transformation of Tr-UnOp1 does not change the *unOp* term, but only substitutes an expression with the same type: The type is preserved. Tr-BinOp-1 is analogous.

Case Tr-UnOp-2. The transformation adds the remote() term. For the induction hypothesis, $P, C \vdash q \rightsquigarrow q' \mid h$ is type preserving. T-Remote tells us that if $P, C \vdash q : T$ then $P, C \vdash \mathsf{remote}(h', q) : T$. Again, the transformation substitutes an expression with the same type into a term that does not change: The type is preserved. Tr-BinOp-2, Tr-BinOp-3 and Tr-BinOp-4 are analogous. □

A.2 Sound Deployment

Proof for 4.8. We start showing that the deployment is secure. By induction on the derivations of the placement algorithm (Figure 10).

Case Tr-Table. In this case, the query is $\mathsf{tableref}(x, h)$. The deployment is secure because T-Table ensures that $C(x) = T$ and $\mathsf{taint}(T) \subseteq P(h)$.

Case Tr-UnOp1. By the induction hypothesis h is a sound deployment for $q' : T$, hence $\mathsf{taint}(T) \subseteq P(h)$. Because of the premise of the theorem, the term $\mathsf{unOp}(q, e)$ is well typed. From Theorem 4.7, we know that the type of $\mathsf{unOp}(q, e)$ and $\mathsf{unOp}(q', e)$ is the same. Since $\mathsf{unOp} \in \{\mathsf{select}, \mathsf{project}, \mathsf{declass}\}$, we have three different possibile type rules for unOp: (1) In the case of select, we know that $\mathsf{select}(q, e)$ has been typed with T-Select. From T-Select, we know that $P, C \vdash \mathsf{select}(q, e) : T + A$, where $A = \mathsf{taintOfBase}(e, T)$. Note that $\mathsf{taintOfBase}(e, T) \subseteq \mathsf{taint}(T)$ and thus $\mathsf{taint}(T+A) = \mathsf{taint}(T)$. Since $\mathsf{taint}(T) \subseteq P(h)$, then $\mathsf{taint}(T+A) \subseteq P(h)$. Because of Theorem 4.7, the type of the transformed term is the same, and $A = T$ we know that h is a feasible deployment for $\mathsf{select}(q', e)$. (2) In the case of project, we know that $\mathsf{project}(q, e)$ has been typed with T-Project. From T-Project, we have $C \vdash \mathsf{project}(q, e) : T'$, where $T' = \{\overline{l : S_l}\}_{l \in \mathsf{fields}(R)}$. From $taint(R.l) = \emptyset$ and T-Proj, we know that $\mathsf{taint}(S_l) = \mathsf{taint}(A_l)$, where $A_l = \mathsf{taintOfBase}(\lambda v.\, e(v).l, T)$. We know that $\mathsf{taintOfBase}(\lambda v.\, e(v).l, T) \subseteq \mathsf{taint}(T)$, thus $\mathsf{taint}(S_l) \subseteq \mathsf{taint}(T)$ and $\mathsf{taint}(T') \subseteq \mathsf{taint}(T)$. Since $\mathsf{taint}(T) \subseteq P(h)$, then $T' \subseteq P(h)$. Because of Theorem 4.7, the type of the transformed term is the same, and $T' \subseteq P(h)$ we know that h is a feasible deployment for $\mathsf{project}(q', e)$. (3) The last case declass is analogous to select.

Case Tr-BinOp-1, Tr-BinOp-2, Tr-BinOp-3, Tr-BinOp-4. For all four rules, the induction hypothesis states that h_1 is a sound deployment for $q_1' : T_1$, and h_2 is a sound deployment for $q_2' : T_2$, hence, $\mathsf{taint}(T_1) \subseteq P(h_1)$ and $\mathsf{taint}(T_2) \subseteq P(h_2)$. Using Theorem 4.7, we can know more about T_1 and T_2 by looking at how $\mathsf{binOp}(q_1, q_2)$ is typed. As $\mathsf{binOp} \in \{\mathsf{crossProduct}, \mathsf{union}, \mathsf{intersect}, \mathsf{difference}\}$, we have four cases: $P, C \vdash \mathsf{crossProduct}(q_1, q_2) : T_1 \cup T_2$, $P, C \vdash \mathsf{union}(q_1, q_2) : T_1 \uplus T_2$, $P, C \vdash \mathsf{intersect}(q_1, q_2) : T_1 \uplus T_2$ and $P, C \vdash \mathsf{difference}(q_1, q_2) : T_1 \uplus T_2$. Theorem 4.7 tells us that these properties hold for the transformed term $\mathsf{binOp}(q_1', q_2')$ too. In the case of Tr-BinOp-1, $h_1 = h_2$. In the notation, we use h_1 in the rest. The induction hypothesis becomes: h_1 is a sound deployment for $q_1' : T_1$, and h_1 is a sound deployment for $q_2' : T_2$, hence, $\mathsf{taint}(T_1) \subseteq P(h_1)$ and $\mathsf{taint}(T_2) \subseteq P(h_1)$. But then $\mathsf{taint}(T_1 \cup T_2) \subseteq P(h_1)$ and $\mathsf{taint}(T_1 \uplus T_2) \subseteq P(h_1)$. Hence, h_1 is a secure deployment for $\mathsf{binOp}(q_1', q_2')$. In the case of Tr-BinOp-3 (Tr-BinOp-2 is symmetrical), we know that $P, C \vdash q_1 : T_1$, $\mathsf{taint}(T_1) \subseteq P(h_2)$, and $P, C \vdash q_1 \rightsquigarrow q_1' \mid h_1$. From Theorem 4.7, we get that $P, C \vdash q_1' : T_1$ (similarly $P, C \vdash q_2' : T_2$). Now, T-Remote tells us that $P, C \vdash \mathsf{remote}(h_2, q_1') : T_1$. Now, similar to the previous case, $\mathsf{binOp} \in \{\mathsf{crossProduct}, \mathsf{union}, \mathsf{intersect}, \mathsf{difference}\}$. But then,

$\text{taint}(T_1 \cup T_2) \subseteq P(h_2)$ and $\text{taint}(T_1 \uplus T_2) \subseteq P(h_2)$, because we know $\text{taint}(T_1) \subseteq P(h_2)$ from the rule. Following the same consideration as before, h_2 is a secure deployment for $\text{binOp}(q_1', q_2')$. Finally, the case Tr-BinOp-4 is similar to the previous case except that reasoning must be applied to both q_1' and q_2' and h' is selected such that $P(h') \supseteq \text{taint}(T_1) \cup \text{taint}(T_2)$. Tr-UnOp-2 is analogous.

We now show that the deployment is valid. The proof proceeds by induction on the derivations of the placement algorithm (Figure 10).

Case Tr-Table. The deployment is valid, because the table is placed on its defined host h.

Case Tr-UnOp-1. By induction, the deployment of q' is valid with $d(q') = h$ for some h. From the inference rule, we know that $d(q') = d(\text{unOp}(q', e)) = h$. Hence (iii) holds. Tr-BinOp-1 is analogous.

Case Tr-BinOp-2. By induction, the deployment d is valid for q_1' and q_2'. As d is valid and from the inference rule, we have $d(q_1') = h_1$ and $d(q_2') = h_2$. Further, the deployment produced by the placement algorithm has $d(\text{remote}(h_1, q_2')) = h_1$. Hence, d is valid for $\text{remote}(h_1, q_2')$, since (ii) holds. Since, d is also valid for q_1 (from the induction hypothesis), we conclude that d is valid for $q = \text{binOp}(q_1', \text{remote}(h_1, q_2'))$ as (iii) holds. Tr-UnOp-2, Tr-BinOp-3 and Tr-BinOp-4 are analogous. □

A.3 Complete Deployment

Proof for 4.9. By contradiction inspecting the rules in Figure 10 showing that if the premises of the theorem hold, the algorithm does not get stuck. We proceed by induction on the derivations of the placement algorithm (Figure 10) and assume that the algorithm did not get stuck until the previous step.

Case Tr-Table. Base case, there are no premises in the rule – hence, this step cannot get stuck.

Case Tr-UnOp-1. By the induction hypothesis the premise holds.

Case Tr-UnOp-2. See Tr-UnOp-1 as we can always apply that rule instead.

Case Tr-BinOp-1, Tr-BinOp-2 and Tr-BinOp-3. The premises $P, C \vdash q_1 \rightsquigarrow q_1' \mid h_1$ and $P, C \vdash q_2 \rightsquigarrow q_2' \mid h_2$ hold by induction. $P, C \vdash q_2 : T_2$ and $P, C \vdash q_1 : T_1$ hold by hypothesis of the theorem. The other premises in the three rules cover all cases except for $h_1 \neq h_2$, $\text{taint}(T_1) \nsubseteq P(h_2)$, $\text{taint}(T_2) \nsubseteq P(h_1)$, which we cover in the next case.

Case Tr-BinOp-4. We have four cases of typing $\text{binOp}(q_1, q_2)$: $P, C \vdash \text{crossProduct}(q_1, q_2) : T_1 \cup T_2$, $P, C \vdash \text{union}(q_1, q_2) : T_1 \uplus T_2$, $P, C \vdash \text{intersect}(q_1, q_2) : T_1 \uplus T_2$ and $P, C \vdash \text{difference}(q_1, q_2) : T_1 \uplus T_2$. Since $\text{taint}(T) = \text{taint}(T_1 \cup T_2) = \text{taint}(T_1 \uplus T_2) = \text{taint}(T_1) \cup \text{taint}(T_2)$, and by the definition of feasible P we know that $P, C \vdash \text{binOp}(q_1, q_2) : T$ and $\exists h.\text{taint}(T) \subseteq P(h)$. We conclude that $\exists h' \in P \; s.t. \; P(h') \supseteq \text{taint}(T_1) \cup \text{taint}(T_2)$. □

A.4 Semantics Preservation

Proof for 4.10. By induction over the structure of the query, showing that the denotation (Figure 6) for the last step does not change. Tr-Table, Tr-UnOp-1 and Tr-BinOp-1, do not change the query, so the theorem trivially holds. Tr-UnOp-2, Tr-BinOp-2, Tr-BinOp-3 and Tr-BinOp-4 introduce remote() terms that do not change the denotation of the query: $[\![\text{remote}(h, q)]\!]_R = \lambda\rho.\{[\![q]\!]_R(\rho)\}$. □

ACKNOWLEDGMENTS

This work has been supported by the German Research Foundation (DFG) within the Collaborative Research Center (CRC) 1053 MAKI and 1119 CROSSING, by the DFG projects SA 2918/2-1 and SA 2918/3-1, by the Hessian LOEWE initiative within the Software-Factory 4.0 project, by the German Federal Ministry of Education and Research, by the Hessian Ministry of Science and the Arts within CRISP, and by the AWS Cloud Credits for Research program. This work has been performed in the context of the LOEWE centre emergenCITY.

REFERENCES

Akka. 2019. Akka toolkit and runtime. http://akka.io.

Arvind Arasu, Spyros Blanas, Ken Eguro, Manas Joglekar, Raghav Kaushik, Donald Kossmann, Ravi Ramamurthy, Prasang Upadhyaya, and Ramarathnam Venkatesan. 2013. Secure Database-as-a-service with Cipherbase. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data (SIGMOD '13)*. ACM, New York, NY, USA, 1033–1036. https://doi.org/10.1145/2463676.2467797

AWS. 2019. AWS Fargate. https://aws.amazon.com/de/fargate/.

S. Bajaj and R. Sion. 2014. TrustedDB: A Trusted Hardware-Based Database with Privacy and Data Confidentiality. *IEEE Transactions on Knowledge and Data Engineering* 26, 3, 752–765. https://doi.org/10.1109/TKDE.2013.38

D. Bell and L. LaPadula. 1973. *Secure Computer Systems: Mathematical Foundations.* Technical Report ESD-TR-73-278. MITRE Corporation.

Gabriel Bender, Lucja Kot, and Johannes Gehrke. 2014. Explainable Security for Relational Databases. In *Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data (SIGMOD '14)*. ACM, New York, NY, USA, 1411–1422. https://doi.org/10.1145/2588555.2593663 event-place: Snowbird, Utah, USA.

Niklas Broberg and David Sands. 2006. Flow Locks: Towards a Core Calculus for Dynamic Flow Policies. In *Proceedings of the 15th European Conference on Programming Languages and Systems (ESOP'06)*. Springer-Verlag, Berlin, Heidelberg, 180–196. https://doi.org/10.1007/11693024_13

Valeria Cardellini, Vincenzo Grassi, Francesco Lo Presti, and Matteo Nardelli. 2017. Optimal Operator Replication and Placement for Distributed Stream Processing Systems. *SIGMETRICS Perform. Eval. Rev.* 44, 4 (May 2017), 11–22. https://doi.org/10.1145/3092819.3092823

James Cheney, Sam Lindley, and Philip Wadler. 2013. A Practical Theory of Language-integrated Query. In *Proceedings of the 18th ACM SIGPLAN International Conference on Functional Programming (ICFP '13)*. ACM, New York, NY, USA, 403–416. https://doi.org/10.1145/2500365.2500586

Mitch Cherniack, Hari Balakrishnan, Magdalena Balazinska, Don Carney, Uğur Çetintemel, Ying Xing, and Stan Zdonik. 2003. Scalable distributed stream processing. In *In CIDR*. Asilomar, CA.

George Copeland and David Maier. 1984. Making Smalltalk a Database System. In *Proceedings of the 1984 ACM SIGMOD International Conference on Management of Data (SIGMOD '84)*. ACM, New York, NY, USA, 316–325. https://doi.org/10.1145/602259.602300

Raimil Cruz, Tamara Rezk, Bernard Serpette, and Éric Tanter. 2017. Type Abstraction for Relaxed Noninterference. In *31st European Conference on Object-Oriented Programming (ECOOP 2017) (Leibniz International Proceedings in Informatics (LIPIcs))*, Peter Müller (Ed.), Vol. 74. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, Dagstuhl, Germany, 7:1–7:27. https://doi.org/10.4230/LIPIcs.ECOOP.2017.7

Gianpaolo Cugola and Alessandro Margara. 2013. Deployment strategies for distributed complex event processing. *Computing* 95, 2 (01 Feb 2013), 129–156. https://doi.org/10.1007/s00607-012-0217-9

Sabrina De Capitani di Vimercati, Sara Foresti, Sushil Jajodia, Giovanni Livraga, Stefano Paraboschi, and Pierangela Samarati. 2017. An Authorization Model for Multi Provider Queries. *Proc. VLDB Endow.* 11, 3 (Nov. 2017), 256–268. https://doi.org/10.14778/3157794.3157796

Jeffrey Dean and Sanjay Ghemawat. 2010. MapReduce: A Flexible Data Processing Tool. *Commun. ACM* 53, 1 (Jan. 2010), 72–77. https://doi.org/10.1145/1629175.1629198

Dorothy E. Denning. 1976. A Lattice Model of Secure Information Flow. *Commun. ACM* 19, 5 (May 1976), 236–243. https://doi.org/10.1145/360051.360056

Dorothy E. Denning and Peter J. Denning. 1977. Certification of Programs for Secure Information Flow. *Commun. ACM* 20, 7 (July 1977), 504–513. https://doi.org/10.1145/359636.359712

Philip Derbeko, Shlomi Dolev, Ehud Gudes, and Shantanu Sharma. 2016. Security and privacy aspects in MapReduce on clouds: A survey. *Computer Science Review* 20 (May 2016), 1–28. https://doi.org/10.1016/j.cosrev.2016.05.001

Ekaterina B. Dimitrova, Panos K. Chrysanthis, and Adam J. Lee. 2019. Authorization-aware Optimization for Multi-provider Queries. In *Proceedings of the 34th ACM/SIGAPP Symposium on Applied Computing (SAC '19)*. ACM, New York, NY, USA, 431–438. https://doi.org/10.1145/3297280.3299731 event-place: Limassol, Cyprus.

Curtis E. Dyreson and Richard Thomas Snodgrass. 1998. Supporting Valid-time Indeterminacy. *ACM Trans. Database Syst.* 23, 1 (March 1998), 1–57. https://doi.org/10.1145/288086.288087

Nicholas L. Farnan, Adam J. Lee, and Ting Yu. 2010. Investigating Privacy-aware Distributed Query Evaluation. In *Proceedings of the 9th Annual ACM Workshop on Privacy in the Electronic Society (WPES '10)*. ACM, New York, NY, USA, 43–52. https://doi.org/10.1145/1866919.1866926

Bent Flyvbjerg. 2006. Five Misunderstandings About Case-Study Research. *Qualitative Inquiry* 12, 2 (2006), 219–245. https://doi.org/10.1177/1077800405284363 arXiv:https://doi.org/10.1177/1077800405284363

Cédric Fournet, Markulf Kohlweiss, George Danezis, and Zhengqin Luo. 2013. ZQL: A Compiler for Privacy-preserving Data Processing. In *Proceedings of the 22Nd USENIX Conference on Security (SEC'13)*. USENIX Association, Berkeley, CA,

USA, 163–178. http://dl.acm.org/citation.cfm?id=2534766.2534781

Reed M. Gardner, T.Allan Pryor, and Homer R. Warner. 1999. The HELP hospital information system: update 1998. *International Journal of Medical Informatics* 54, 3 (1999), 169 – 182. https://doi.org/10.1016/S1386-5056(99)00013-1

J. A. Goguen and J. Meseguer. 1982. Security Policies and Security Models. In *1982 IEEE Symposium on Security and Privacy*. 11–11. https://doi.org/10.1109/SP.1982.10014

Marco Guarnieri, Musard Balliu, Daniel Schoepe, David Basin, and Andrei Sabelfeld. 2019. Information-Flow Control for Database-Backed Applications. In *2019 IEEE European Symposium on Security and Privacy (EuroS P)*. 79–94. https://doi.org/10.1109/EuroSP.2019.00016

Ryan Huebsch, Joseph M. Hellerstein, Nick Lanham, Boon Thau Loo, Scott Shenker, and Ion Stoica. 2003. Querying the Internet with PIER. In *Proceedings of the 29th International Conference on Very Large Data Bases - Volume 29 (VLDB '03)*. VLDB Endowment, Berlin, Germany, 321–332. http://dl.acm.org/citation.cfm?id=1315451.1315480

Steven Y. Ko, Kyungho Jeon, and Ramsés Morales. 2011. *The HybrEx Model for Confidentiality and Privacy in Cloud Computing*. USENIX Association, Berkeley, CA, USA. 8–8 pages. http://dl.acm.org/citation.cfm?id=2170444.2170452

Krzysztof Kuchcinski and Radoslaw Szymanek. 2017. JaCoP - Java Constraint Programming solver. https://osolpro.atlassian.net/wiki/display/JACOP/.

Geetika T. Lakshmanan, Ying Li, and Rob Strom. 2008. Placement Strategies for Internet-Scale Data Stream Systems. *IEEE Internet Computing* 12, 6 (Nov 2008), 50–60. https://doi.org/10.1109/MIC.2008.129

Jed Liu, Owen Arden, Michael D. George, and Andrew C. Myers. 2017. Fabric: Building open distributed systems securely by construction. *Journal of Computer Security* 25, 4-5 (2017), 367–426. https://doi.org/10.3233/JCS-15805

Ralf Mitschke, Sebastian Erdweg, Mirko Köhler, Mira Mezini, and Guido Salvaneschi. 2014. I3QL: Language-integrated Live Data Views. In *Proceedings of the 2014 ACM International Conference on Object Oriented Programming Systems Languages & Applications (OOPSLA '14)*. ACM, New York, NY, USA, 417–432. https://doi.org/10.1145/2660193.2660242

Andrew C. Myers. 1999. JFlow: Practical Mostly-static Information Flow Control. In *Proceedings of the 26th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '99)*. ACM, New York, NY, USA, 228–241. https://doi.org/10.1145/292540.292561

Andrew C. Myers and Barbara Liskov. 1997. A Decentralized Model for Information Flow Control. In *Proceedings of the Sixteenth ACM Symposium on Operating Systems Principles (SOSP '97)*. ACM, New York, NY, USA, 129–142. https://doi.org/10.1145/268998.266669

K. Y. Oktay, M. Kantarcioglu, and S. Mehrotra. 2017. Secure and Efficient Query Processing over Hybrid Clouds. In *2017 IEEE 33rd International Conference on Data Engineering (ICDE)*. ieee, San Diego, CA, USA, 733–744. https://doi.org/10.1109/ICDE.2017.125

Kerim Yasin Oktay, Sharad Mehrotra, Vaibhav Khadilkar, and Murat Kantarcioglu. 2015. SEMROD: Secure and Efficient MapReduce Over HybriD Clouds. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data (SIGMOD '15)*. ACM, New York, NY, USA, 153–166. https://doi.org/10.1145/2723372.2723741

OpenClinical. 2004. HELP - Health Evaluation Through Logical Processing. http://www.openclinical.org/aisp_help.html.

Peter Pietzuch, Jonathan Ledlie, Jeffrey Shneidman, Mema Roussopoulos, Matt Welsh, and Margo Seltzer. 2006. Network-Aware Operator Placement for Stream-Processing Systems. In *Proceedings of the 22Nd International Conference on Data Engineering (ICDE '06)*. IEEE Computer Society, Washington, DC, USA, 49–60. https://doi.org/10.1109/ICDE.2006.105

Raluca Ada Popa, Catherine M. S. Redfield, Nickolai Zeldovich, and Hari Balakrishnan. 2011. CryptDB: Protecting Confidentiality with Encrypted Query Processing. In *ACM Symposium on Operating Systems Principles (SOSP '11)*. ACM, New York, NY, USA, 85–100. https://doi.org/10.1145/2043556.2043566

Tiark Rompf and Martin Odersky. 2012. Lightweight modular staging: a pragmatic approach to runtime code generation and compiled DSLs. *Commun. ACM* 55, 6 (2012), 121–130.

Daniel Schoepe, Daniel Hedin, and Andrei Sabelfeld. 2014. SeLINQ: Tracking Information Across Application-database Boundaries. In *Proceedings of the 19th ACM SIGPLAN International Conference on Functional Programming (ICFP '14)*. ACM, New York, NY, USA, 25–38. https://doi.org/10.1145/2628136.2628151

Shayak Sen, Saikat Guha, Anupam Datta, Sriram K. Rajamani, Janice Tsai, and Jeannette M. Wing. 2014. Bootstrapping Privacy Compliance in Big Data Systems. In *Proceedings of the 2014 IEEE Symposium on Security and Privacy (SP '14)*. IEEE Computer Society, Washington, DC, USA, 327–342. https://doi.org/10.1109/SP.2014.28

Sai Deep Tetali, Mohsen Lesani, Rupak Majumdar, and Todd Millstein. 2013. MrCrypt: Static Analysis for Secure Cloud Computations. In *International Conference on Object Oriented Programming Systems Languages and Applications (OOPSLA '13)*. ACM, New York, NY, USA, 271–286. https://doi.org/10.1145/2509136.2509554

C. Thoma, A. Labrinidis, and A. J. Lee. 2014. Automated operator placement in distributed Data Stream Management Systems subject to user constraints. In *2014 IEEE 30th International Conference on Data Engineering Workshops*. 310–316. https://doi.org/10.1109/ICDEW.2014.6818346

Feng Tian and David J. DeWitt. 2003. Tuple Routing Strategies for Distributed Eddies. In *Proceedings of the 29th International Conference on Very Large Data Bases - Volume 29 (VLDB '03)*. VLDB Endowment, Berlin, Germany, 333–344. http:

//dl.acm.org/citation.cfm?id=1315451.1315481

TPC. 2019. TPC-H Benchmark Specification. http://www.tpc.org/tpch.

Stephen Tu, M. Frans Kaashoek, Samuel Madden, and Nickolai Zeldovich. 2013. Processing analytical queries over encrypted data. In *Proceedings of the 39th international conference on Very Large Data Bases (PVLDB'13)*. VLDB Endowment, 289–300. http://dl.acm.org/citation.cfm?id=2488335.2488336

Huseyin Ulusoy, Pietro Colombo, Elena Ferrari, Murat Kantarcioglu, and Erman Pattuk. 2015. GuardMR: Fine-grained Security Policy Enforcement for MapReduce Systems. In *Proceedings of the 10th ACM Symposium on Information, Computer and Communications Security (ASIA CCS '15)*. ACM, New York, NY, USA, 285–296. https://doi.org/10.1145/2714576.2714624

Yannis Vassiliou. 1979. Null Values in Data Base Management: A Denotational Semantics Approach. In *Proceedings of the 1979 ACM SIGMOD International Conference on Management of Data (SIGMOD '79)*. ACM, New York, NY, USA, 162–169. https://doi.org/10.1145/582095.582123

Dennis Volpano, Cynthia Irvine, and Geoffrey Smith. 1996. A Sound Type System for Secure Flow Analysis. *J. Comput. Secur.* 4, 2-3 (Jan. 1996), 167–187. http://dl.acm.org/citation.cfm?id=353629.353648

Pascal Weisenburger, Manisha Luthra, Boris Koldehofe, and Guido Salvaneschi. 2017. Quality-aware Runtime Adaptation in Complex Event Processing. In *Proceedings of the 12th International Symposium on Software Engineering for Adaptive and Self-Managing Systems* (**SEAMS'17**). IEEE Press, Piscataway, NJ, USA, 140–151. https://doi.org/10.1109/SEAMS.2017.10

Ying Xing, Stan Zdonik, and Jeong-Hyon Hwang. 2005. Dynamic Load Distribution in the Borealis Stream Processor. In *Proceedings of the 21st International Conference on Data Engineering (ICDE '05)*. IEEE Computer Society, Washington, DC, USA, 791–802. https://doi.org/10.1109/ICDE.2005.53

Jean Yang, Travis Hance, Thomas H. Austin, Armando Solar-Lezama, Cormac Flanagan, and Stephen Chong. 2016. Precise, Dynamic Information Flow for Database-backed Applications. In *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '16)*. ACM, New York, NY, USA, 631–647. https://doi.org/10.1145/2908080.2908098

Steve Zdancewic. 2013. A Type System for Robust Declassification. *Electron. Notes Theor. Comput. Sci.* 83 (Jan. 2013), 263–277. https://doi.org/10.1016/S1571-0661(03)50014-7

Steve Zdancewic, Lantian Zheng, Nathaniel Nystrom, and Andrew C. Myers. 2001. Untrusted Hosts and Confidentiality: Secure Program Partitioning. In *Proceedings of the Eighteenth ACM Symposium on Operating Systems Principles (SOSP '01)*. ACM, New York, NY, USA, 1–14. https://doi.org/10.1145/502034.502036

Q. Zeng, M. Zhao, P. Liu, P. Yadav, S. Calo, and J. Lobo. 2015. Enforcement of Autonomous Authorizations in Collaborative Distributed Query Evaluation. *IEEE Transactions on Knowledge and Data Engineering* 27, 4 (April 2015), 979–992. https://doi.org/10.1109/TKDE.2014.2357018

Kehuan Zhang, Xiaoyong Zhou, Yangyi Chen, XiaoFeng Wang, and Yaoping Ruan. 2011. Sedic: Privacy-aware Data Intensive Computing on Hybrid Clouds. In *Proceedings of the 18th ACM Conference on Computer and Communications Security (CCS '11)*. ACM, New York, NY, USA, 515–526. https://doi.org/10.1145/2046707.2046767

Lantian Zheng, Stephen Chong, Andrew C. Myers, and Steve Zdancewic. 2003. Using Replication and Partitioning to Build Secure Distributed Systems. In *Proceedings of the 2003 IEEE Symposium on Security and Privacy (SP '03)*. IEEE Computer Society, Washington, DC, USA, 236–. http://dl.acm.org/citation.cfm?id=829515.830549

Yongluan Zhou, Beng Chin Ooi, Kian-Lee Tan, and Ji Wu. 2006. Efficient Dynamic Operator Placement in a Locally Distributed Continuous Query System. In *Proceedings of the 2006 Confederated International Conference on On the Move to Meaningful Internet Systems: CoopIS, DOA, GADA, and ODBASE - Volume Part I (ODBASE'06/OTM'06)*. Springer-Verlag, Berlin, Heidelberg, 54–71. https://doi.org/10.1007/11914853_5