SMCQL: Secure Querying for Federated Databases

Johes Bater
Northwestern University
johes@u.northwestern.edu

Gregory Elliott
Northwestern University
GregoryElliott2016@u.northwestern.edu

Craig Eggen
Northwestern University
CraigEggen2016@u.northwestern.edu

Satyender Goel
Northwestern University
s-goel@northwestern.edu

Abel Kho
Northwestern University
Abel.Kho@nm.org

Jennie Rogers
Northwestern University
jennie@eecs.northwestern.edu

ABSTRACT

People and machines are collecting data at an unprecedented rate. Despite this newfound abundance of data, progress has been slow in *sharing* it for open science, business, and other data-intensive endeavors. Many such efforts are stymied by privacy concerns and regulatory compliance issues. For example, many hospitals are interested in pooling their medical records for research, but none may disclose arbitrary patient records to researchers or other healthcare providers. In this context we propose the Private Data Network (PDN), a federated database for querying over the collective data of *mutually distrustful* parties. In a PDN, each member database does not reveal its tuples to its peers nor to the query writer. Instead, the user submits a query to an honest broker that plans and coordinates its execution over multiple private databases using secure multiparty computation (SMC). Here, each database's query execution is *oblivious*, and its program counters and memory traces are agnostic to the inputs of others.

We introduce a framework for executing PDN queries named SMCQL. This system translates SQL statements into SMC primitives to compute query results over the union of its source databases without revealing sensitive information about individual tuples to peer data providers or the honest broker. Only the honest broker and the querier receive the results of a PDN query. For fast, secure query evaluation, we explore a heuristics-driven optimizer that minimizes the PDN's use of secure computation and partitions its query evaluation into scalable slices.

1. INTRODUCTION

Federated database systems, wherein many autonomous databases are united to appear as a single engine for querying, are having a renaissance in "big data" applications. Interestingly, many such federations contain data owned by *mutually distrustful* parties who are willing to have the union of their data analyzed, but will not disclose their raw tuples. We call a database federation that spans mutually distrustful sources a *private data network* or PDN. Federations of this kind contain data that is privately held and not available for upload to the cloud.

Proceedings of the VLDB Endowment, Vol. 10, No. 6

In exploring this topic, we identified use cases for PDNs in medicine, data markets, banking, online advertising, and human rights work. Typically, PDN members either upload their data to a trusted intermediary or they use one-off privacy-preserving algorithms to mine it [15, 2]. We posit that PDNs will see broader adoption if their users express their analytics as declarative *SQL statements*.

For example, a consortium of hospitals is interested in pooling their patient records for clinical data research and each site is in charge of securing their own data. A university researcher, operating independently of the hospitals, wants to evaluate a new treatment for rare disease X. Her first step is to ask if there is a large enough cohort of X sufferers in the consortium to form a study. She writes `SELECT COUNT(DISTINCT patient_id) FROM diagnosis WHERE diag=X;` to the consortium coordinator. If this query were run in a standard database federation, the coordinator would collect and merge patient IDs for X from each site, eliminate duplicates, and counts them. This approach is undesirable because it reveals the patient IDs of individuals affected by X to the coordinator. A secure framework is needed that enables researchers to execute distributed queries without exposing information to unauthorized parties about their source data or intermediate results.

The PDN architecture is shown in Figure 1. Here, a user submits their query to the federation's *honest broker*, a neutral third party that plans and orchestrates its execution over two or more private data providers. The federation has a shared schema that is supported by all parties. The execution of a PDN query is distributed over a secure compute cluster of *private data providers*. Each provider executes a secure protocol provided by the honest broker that produces a share of the query output. The honest broker assembles the shares into output tuples and sends them to the end user. From the user's perspective, the PDN behaves exactly like a conventional federated database where one submits SQL and receives query results. In our example above, the end user is the researcher, the hospitals are private data providers, and the honest broker is the consortium coordinator.

At setup time, the PDN has a shared set of table definitions. This schema is annotated with the level of protection required for each of its attributes. For identifiers that span multiple data providers, such as patient IDs that appear in multiple hospitals, the honest broker works with the PDN members to carry out secure record linkage as in [3, 18, 22].

SMCQL is our framework for planning and executing PDN queries. It uses secure multiparty computation (SMC) to evaluate distributed queries among mutually distrustful par-

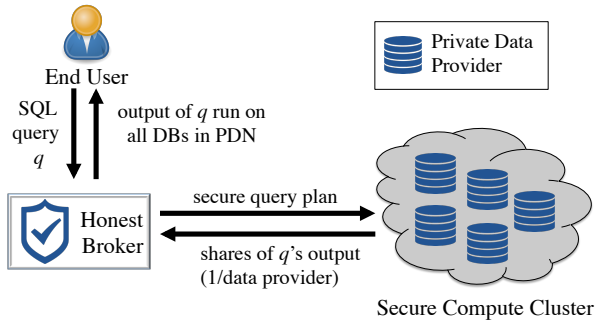


Figure **1**: Private data network architecture

ties. SMC is a subfield of cryptography that studies methods whereby two or more parties compute a function over their data while each keeps their individual inputs private. SMC makes query evaluation *oblivious* such that each party's computation is independent of the inputs of others. An example of secure function evaluation is Yao's Millionaire Problem [38]. He asked, "If Alice and Bob are millionaires who are interested in determining which one of them is richer, how can they solve for this without either party revealing their net worth?". Speaking imprecisely, SMC provides a black box within which the mutually distrustful parties combine their sensitive tuples for query evaluation.

Since SMC computes distributed query evaluation obliviously, it exacts a high performance cost in comparison to plaintext query evaluation. For example, if one runs a join using SMC, the output of this operator will be the size of the cross product of its inputs. Its output is padded with encrypted nulls to maintain the query's obliviousness. With cascades of oblivious database operators, the cardinality of their intermediate results grows rapidly. It is not uncommon to see secure functions run multiple orders of magnitude slower than their plaintext counterparts.

In our work, a PDN seamlessly translates database operators within a query execution plan into SMC primitives. It carefully manages its use of SMC so that its queries run efficiently. The query planner first identifies when SMC is needed in a query by modeling the flow of sensitive data through its operator tree. After that, it optimizes the subtree's execution using heuristics. Lastly, the system generates secure code for the optimized plan. In this paper, we present a query planner and executor for two mutually distrustful parties.

SMCQL offers an honest-but-curious threat model. In other words, we trust each data provider to faithfully execute the protocol provided by the honest broker. On the other hand, participants may attempt to deduce the sensitive data of others during a secure query execution using side channels including program counters and memory accesses. We do not address the question of malicious queriers who try to infer sensitive data by examining the output of many PDN queries. Differential privacy [8] tackles this issue by injecting controlled levels of noise into the data at query time to obscure the role of an individual in a larger dataset.

In contrast, we propose a rule-based approach to protecting sensitive data that returns precise query results. Here, the PDN's stakeholders create a policy defining the queries they will admit for execution. This policy may stipulate a minimum number of parties that must participate in each secure computation or attributes that are never accessible to end users in plaintext. These rules reflect the cultural norms and best practices of a PDN's domain. For example, when hospitals share health records, they typically protect patient privacy with the heuristics of HIPAA Safe Harbor [9].

We hypothesize that with a combination of common-sense rules and external incentives (e.g., legal remedies for bad actors), we can make a much larger set of data available for sharing. One of the questions we are pursuing with this work is how far a rule-based PDN security policy will go in a real-world setting. Going forward, if users wish to add rigorous tuple-level privacy to their SQL workflow, they may integrate differentially private querying into a PDN using the techniques described in [26].

SMCQL is a substantial departure from existing research on secure databases. Prior work used homomorphic encryption for the outsourced storage and querying of private data [30]. Our approach keeps data in the hands its originators and uses SMC to conceal the query's computation rather than the data itself. Researchers have also studied secure computation for outsourced database systems [1, 7]. In contrast, a PDN distributes secure computation among the data providers and removes the need for any trusted intermediaries beyond a lightweight honest broker for coordinating query evaluation.

There are numerous existing approaches to making SMC available to untrained users, including domain-specific programming languages [23, 25, 32] and extending existing languages [39]. Our approach differs from them in that we do not require the programmer to reason explicitly about how to combine the data of each party. Instead we use SQL's semantics to translate queries into SMC. To the best of our knowledge, this is the first system that enables users to take advantage of SMC without reasoning about the security properties of the underlying system. We decouple the security policy from the querier by providing the honest broker with a schema-based security policy at startup.

Our main contributions in this work are:

- A novel generalization of federated databases for querying over data providers that do not trust one another.
- A code generator that automatically translates SQL into secure computing primitives.
- A heuristics-based optimizer for PDN query execution
- An in-depth evaluation of this system on real-world medical data.

The remainder of this paper is organized as follows. In Section 2 we describe the basics of SMC and introduce a running example for PDNs. Section 3 provides an overview of the SMCQL system. Next, we describe our code generator in Section 4. Section 5 describes our PDN security policy and how we use it to identify the subtree of a query plan for which SMC is needed. We then explore our secure query plan optimizations. After that, we present the results our experimental evaluation over real-world data. Finally, we survey the relevant literature and conclude.

2. BACKGROUND

SMCQL leverages existing SMC primitives for secure query evaluation. In this section, we briefly describe these building blocks and give an intuition about how they work. We also introduce a running example from clinical data research. For clarity, we refer to two mutually distrustful data providers as Alice and Bob.

2.1 Secure Multiparty Computation

SMC systems allow parties to jointly compute functions while keeping each party's inputs secret. In PDNs, database operators act as secure functions that are evaluated over the sensitive data of two or more parties. SMCQL performs secure computation using garbled circuits, and conceals its access patterns of sensitive tuples with oblivious RAM. We chose these classical techniques because they are heavily optimized in existing work and easily accessible for code generation.

Garbled Circuits For secure query evaluation, we generate garbled circuits to compute database operators over the data of multiple parties. Garbled circuits are a series of logic gates (e.g., AND, XOR) that hide the program traces of their computation. These circuits use the same distribution of execution time, regardless of their inputs, to make it impossible for a party to deduce the inputs of others. In our framework, the only information available to the data providers is the number of tuples provided by each party. Garbled circuits are quite expressive, and can be used to compute any arbitrary function [6]. This technique has been extended for three or more parties [12].

Each garbled circuit securely computes a function, such as $a \geq b$ in Yao's Millionaire Problem. In order to not reveal the inputs of Alice and Bob, we execute the entire circuit in the worst-case performance scenario. If we were computing $a \geq b$ in plaintext, we'd find the most significant bit where Alice and Bob's inputs differ and use it to determine the circuit output. A garbled circuit must instead compare all of the bits in order to not disclose the most significant one where the inputs differed. In SMCQL, the first data provider, Alice, generates the garbled circuits, and the second one, Bob, evaluates them.

Oblivious RAM In addition to covering the compute traces effected by the input of another party, our engine hides the memory access patterns of a secure program. We use *oblivious RAM* (ORAM) [11] to store arrays of tuples as they move through the secure query executor. This data structure shuffles the tuple array at each read or write, thereby making all memory accesses indistinguishable from one another. This prevents attackers from using reads and writes in a secure program to learn about the underlying data. ORAM enables us to reduce the depth of our garbled circuits by creating a small circuit for each database operator and securely passing the output of one operator to the next through oblivious reads rather than evaluating the query in a single, massive circuit.

2.2 HealthLNK Running Example

Throughout the text, we use a running example of a group of hospitals that wish to mine the collective data of their electronic health record systems for research while keeping individual tuples private. We first examine the architecture of this system and then describe a set of representative queries with which we explore SMCQL.

A clinical data research network or CDRN is a consortium of healthcare sites that agree to share their data for research. CDRN data providers may be mutually distrustful parties. We examine this work in the context of HealthLNK [29], a CDRN prototype for Chicago-area healthcare sites. This repository contains records from seven Chicago-area healthcare institutions, each with their own member hospitals, from 2006 to 2012, totaling about 6 million records. The data set is selected from a diverse set of hospitals, including academic medical centers, large county hospitals, and local community health centers.

HealthLNK is the forerunner for the Chicago Area Patient-Centered Outcomes Research Network (CAPriCORN), which is itself part of a national network in the US, the Patient-Centered Outcome Research Network (PCORnet). The CAPriCORN consortium includes 481 data sources, each of which has protected health information (PHI) such as gender, timestamps, and diagnoses [19]. CAPriCORN and HealthLNK share the majority of their stakeholders and this group designed these systems to meet the needs of clinical researchers, especially ones exploring personalized medicine.

In the absence of a secure query evaluation framework like SMCQL, CDRN members resort to computing distributed queries entirely within the honest broker. Because each data provider is in charge of their own HIPAA compliance, they will often only volunteer their least sensitive database attributes for querying. Although each site will not disclose its PHI, they are willing to compute queries over sensitive data when multiple records are accessed together.

In addition to limiting the queryable attributes in a CDRN, this hub-and-spoke architecture will not scale to hundreds of data providers. This is an issue we are already seeing in the field. On the other hand, this setting makes it is possible for individual tuples to "hide in the crowd" of the data from many participating healthcare sites. This makes CDRNs a prime use case for SMCQL. We are investigating deploying a prototype of SMCQL on CAPriCORN.

2.2.1 Query Workload

We now explore the workings of SMCQL with three representative queries based on clinical data research protocols [14, 28]. In addition, we evaluate the system with this workload on de-identified medical records from the HealthLNK data repository.

Comorbidity *Clostridium difficile*, or *c. diff*, is an infection that is often antibiotic-resistant. Our first query finds the most common types of ailments that arise for *c. diff* sufferers:

```
SELECT diag, COUNT(*) cnt
FROM diagnoses
WHERE patient_id IN cdiff_cohort
GROUP BY diag
ORDER BY cnt
LIMIT 10;
```

The query selects the diagnoses of individuals in the *c. diff* cohort and counts the diagnoses for each condition, returning the most common ones to the user. With *comorbidity*, we explore practical techniques for minimizing the use of SMC in distributed query evaluation.

Recurrent C. Diff *C. diff* sufferers have a high rate of re-infection after treatment. When patients are treated at multiple hospitals, recurrent *c. diff* frequently goes undetected. This is exactly the type of problem that SMCQL is designed to solve. This query identifies a cohort of recurrent *c. diff* patients whose second infection occurs between 15 and 56 days after an initial diagnosis:

```
WITH rcd AS (
    SELECT pid, time, row_no() OVER
        (PARTITION BY pid ORDER BY time)
    FROM diagnosis
    WHERE diag=cdiff)
```

```
SELECT DISTINCT pid
FROM rcd r1 JOIN rcd r2 ON r1.pid = r2.pid
WHERE r2.time - r1.time >= 15 DAYS
     AND r2.time - r1.time <= 56 DAYS
     AND r2.row_no = r1.row_no + 1;
```

We first select for *c. diff*, and use a window aggregate to number the diagnoses of each patient in chronological order. We then compare the ith diagnoses to the $(i + 1)$th one using a self-join to find recurring infections in the prescribed date range. Lastly, the query eliminates duplicate patient IDs. We use this query to examine how fine-grained data partitioning (by patient ID) improves SMCQL's performance.

Aspirin Count In our final query, we identify the number of heart disease sufferers who were prescribed Aspirin. Here, researchers are investigating the effectiveness of Aspirin in preventing repeated heart attacks. We calculate the *Aspirin Count* of heart disease patients as:

```
SELECT COUNT(DISTINCT pid)
FROM diagnosis d JOIN medication m ON d.pid = m.pid
WHERE d.diag = hd AND m.med = aspirin
    AND d.time <= m.time;
```

The query first filters the diagnosis table for heart disease patients and the medications for Aspirin. It then joins these tables to identify patients who were prescribed Aspirin during or after a heart disease diagnosis. Lastly, we count the distinct patient IDs that meet this criterion. This query tests the SMCQL optimizer's ability to create high-performance query plans for complex sequences of operators.

3. SYSTEM OVERVIEW AND ROADMAP

In this section, we walk through the steps SMCQL takes to translate a SQL statement into a secure query execution plan, as detailed in Figure 2. The honest broker starts with a SQL statement provided by the user. The statement is written against the PDN's shared schema. The honest broker parses the statement into a directed acyclic graph (DAG) using well-known techniques [31]. This tree of database operators, such as joins and aggregates, provides the steps needed to compute a given query. The honest broker examines this tree, confirming that it is runnable within the PDN's security policy.

Next, we generate a *secure plan* that identifies the minimal subtree in the query's DAG that must run obliviously to uphold the PDN's security policy. We describe this process in Section 5.2. The planner traverses the tree bottom-up, modeling the flow of sensitive attributes through its nodes.

Next, as described in Section 6, SMCQL optimizes the secure query tree using heuristics that partition database operators into small, scalable units of secure computation. We also propose methods for reducing the secure computation performed within an operator.

Armed with a tree of optimized operator specifications, the planner generates SMC code for execution on the data providers using the techniques in Section 4. For each relational algebra operator, the code generator looks up a template for it and populates this outline with query-specific information including the width of its tuples in bits and filter predicates.

When the code generator completes we have an executable secure plan. The honest broker distributes the compiled secure code to the data providers, along with plaintext source

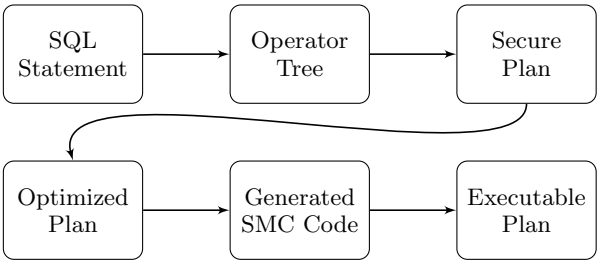


Figure **2**: SMCQL query compilation and execution

queries for generating inputs for SMC. The data providers run their source SQL queries within their databases with standard methods and coordinate with one another to generate and evaluate garbled circuits for the secure operators using the specifications provided by the SMC code.

End-to-end, SMCQL implements a wide range of SQL operators. It supports selection, projection, aggregation (including `DISTINCT`), limit, and some window aggregates. For joins, we handle equi-joins, theta joins, and cross products. At this time we do not support outer joins or set operations. We detail how the latter's semantics would be implemented in Section 5.2.

4. SECURE QUERY EXECUTOR

We now examine the process whereby SMCQL translates physical operators in a PDN plan into executable, secure code. We first prepare the plan by adding steps that combine tuples from multiple parties. After that, we generate secure code for each relational operator in the operator tree.

This secure code uses garbled circuits to jointly compute an operator over the participating parties, and stores the result in ORAM. The engine uses this secure data structure to pass intermediate results between nodes in the operator tree. The query executor carries out this secure computation–with Alice generating the circuits and Bob evaluating them–for every subsequent operator in the tree until it reaches the root node. Each party ships their garbled result from the root node to the honest broker for decoding the query's output tuples.

To create garbled circuits and ORAM for PDN query evaluation, we use a domain-specific programming language, ObliVM [23]. This language has a C-style syntax. Among other features, ObliVM offers callable functions, loops, and if-then statements. A programmer declares and accesses ORAM in ObliVM using the C language's bracket notation. This framework compiles its code in two steps. First, it translates the code into a set of logic gates and ORAM accesses. Then, at execution time, it generates the garbled circuits on the fly to prevent replay attacks. This language is backend-agnostic, so if one were to research new garbled circuit protocols, they would be able to seamlessly compile them into the code generated by SMCQL.

Each secure operator starts with a template, or a parameterized C-style program for the operator's execution logic. Templates have variables for filter predicates, input tuple widths, and for projecting their output as needed. The system has a library of operator-specific templates, including ones for the optimizations introduced in Section 6.

We show an example template for joins in Figure 3. The template's parameters are denoted with a "$". The system populates the template's variables using parameters from its relational query plan. This join function takes in two arrays

```
int$dSize[m*n] join(int$lSize[m] lhs, int$rSize[n] rhs) {

   int$dSize[m*n] dst;
   int dstIdx = 0;

    for(int i = 0; i < m; i=i+1)  {
       int$lSize l = lhs[i];
       for(int j = 0; j < n; j=j+1) {
             int$rSize r = rhs[j];
             if($filter(l, r) == 1) {
                  dst[dstIdx] = $project;
                  dstIdx = dstIdx + 1;
             }
        }
   }
   return dst;
}
```

Figure **3**: Template for secure join

of tuples, one for each input. Each array contains tuples from both parties. The variables $lSize and $rSize denote the number of bits per tuple in each input relation. The tuples per input table are stored in n and m, and the SMC executor infers these values at runtime.

After populating an operator's template, we compile it into a low-level representation with logic gates and ORAM accesses. This is the code that each data provider executes. These low-level directives automatically generate all of the garbled circuits (and their coded inputs) at runtime to protect the secure query evaluation from replay attacks.

4.1 SMC Performance Costs

All SMC techniques incur substantial overhead in comparison to their plaintext counterparts. To get an intuition for the performance costs and optimization opportunities associated with secure query evaluation, we hand-coded the three queries in Section 2.2 as oblivious programs. We ran these carefully optimized, fully-SMC implementations on a randomly selected subset of tuples that matched the query's initial selection criteria. The details of our experimental configuration are in Section 7.1. To bound the duration of our experiments, our input samples had 50 tuples per table.

Our results are shown in Table 1. We see that a purely secure query evaluation is on the order of 4–5 orders of magnitude slower than its plaintext execution. This is unacceptably slow. We see that for complex plans with cascades of operators, like *aspirin count*, the runtime explodes. In order to keep their computation oblivious, each step in the query has an output cardinality equal to its maximum possible size. For example, the output of a join is the size of the cross product of its inputs. Thus *aspirin count*'s aggregate after the join must process all of these tuples–despite many of them being nulls–to avoid revealing information about the contents of the join inputs.

Instead of naïvely using SMC on the entire query plan, we need a more nuanced approach. The SMCQL optimizer in Section 6 uses SQL semantics to minimize the work performed in SMC and it breaks this secure computation into small, scalable partitions for speedy evaluation. These optimizations are agnostic to the SMC primitives used in distributed query evaluation. Selecting the highest-performance SMC protocol for a given query is a promising direction for future work. Before we can discuss our optimizations, we first introduce a security model with which we identify query evaluation over sensitive data. This model allows us to formally describe our optimizations and verify that the system satisfies the security requirements of its stakeholders.

Table **1**: Slowdown of HealthLNK queries run with SMC.

Test	Plaintext	Secure	Slowdown
Comorbidity	158	253,894	1,609X
Recurrent C. Diff	165	159,145	967X
Aspirin Count	193	8,195,317	43,337X

5. SECURITY MODEL

We now formulate the challenge of creating oblivious query execution plans. First, we describe SMCQL's user-facing security policy for PDN queries. This attribute-level model specifies who may access the PDN's data and under what conditions. After that, we explore SMCQL's security type system, with which the planner identifies the minimal set of operators in a query plan that require oblivious evaluation to protect a PDN's sensitive data.

5.1 Security Policy

SMCQL offers a simple, yet powerful security model to protect PDN data from unauthorized access. Recall that a PDN begins with a common set of table specifications and that they define the level of protection needed for their data one column at a time. This approach mirrors how PDN stakeholders often reason about the sensitivity of their data.

Our model offers three levels of data access: public, protected, and private. By working with stakeholders, a DBA creates an annotated schema that specifies the PDN's security policy. This policy also enables the SMCQL optimizer to create efficient, secure plans.

Public attributes are readable by all parties, including the honest broker, data providers, and end users. These columns have the lowest sensitivity, and often have a low probability of being independently replicable. In HealthLNK, anonymized patient IDs, lab results, and blood pressure readings are public attributes.

Protected data is visible in their originating site and conditionally available to the end user and honest broker. SMCQL uses k-anonymity to control access to protected attributes. A selection is k-anonymous iff each of its tuples is indistinguishable in its protected attributes from at least $k-1$ records. This policy is one of many possibilities for controlling access to protected data. Any distributed query evaluation over these attributes is done securely. In our running example, protected attributes include diagnosis codes, age, and gender.

Private attributes are the most sensitive ones in a PDN, and they are not disclosed to anyone outside of the initial data provider. Computation over these attributes must be carried out obliviously. Private attributes may not appear in any results returned to the user. Timestamps and zip codes are examples of private attributes in HealthLNK.

This access control policy governs when and how a PDN uses secure computation. In addition, the PDN is configured with a query admission policy. This policy may disallow certain patterns of querying, such as repeated, but slightly modified ones designed to unmask individuals in a database. It may also enable data providers to hide in the crowd with requirements such as "at least k data providers must contribute tuples to a secure computation". A PDN may automatically reject queries that do not meet its policy, using a system such as DataLawyer [36]. Another approach is to audit query trails to determine if a sequence of queries is threatening to unmask sensitive views of the data [27].

We now address the planning and optimization of PDN queries. Since query processing takes place within the data providers, we treat protected attributes as if they are private.

5.2 Secure Information Flow

Our first tactic for optimizing a PDN query is to minimize the number of operators it runs securely. We use a security type system [17, 34, 35, 37] to analyze the flow of secure attributes through an operator tree. The planner traverses the tree bottom-up, recording the operators that will be executed obliviously to fulfill the requirements of the PDN's security policy. In each operator, we examine the provenance of its output columns and determine the protection level needed for each one by taking the maximum security policy of its source attributes.

The security type system begins with a grammar with which the planner interprets the query's operator tree. Grammar 1 shows the syntax with which the type system will analyze a query tree to determine whether each of its database operators needs oblivious evaluation. This syntax closely follows that of relational algebra.

$\langle$*phrases*$\rangle$ ρ ::= e | E | Op

$\langle$*expressions*$\rangle$ e ::= a | n | e + e' | $e \leq e'$ | $e \wedge e'$ | ...

$\langle$*sets*$\rangle$ E ::= $\{e_1, e_2, \ldots e_n\}$

$\langle$*operators*$\rangle$ Op ::= Op'(Op(.))
| Op'(Op(.), Op(.)) | `scan(E)`
| $\sigma_e(E)$ | $\pi_{E'}(E)$ | $E \bowtie_e E'$
| `agg(E)` | `limit(E)` | `sort(E)`
| $E \cup E'$ | $E \cap E'$ | $E \setminus E'$

Grammar 1: Grammar for SQL query plans.

All objects in a query plan are phrases, represented by ρ. A phrase may be an expression (e), a set of expressions (E), or a relational operator (Op). Expressions may describe attribute references (a), string and integer literals (n), arithmetic operators, comparisons or logical connectives.

We reason about a query plan as a set of relational operators. Operators are arranged in a tree and each operator has up to two children. An operator takes in a set of expressions, E, and produces a new set, E', as output. The grammar offers table scans, filters ($\sigma_e(E)$), projections ($\pi_{E'}(E)$), joins ($\bowtie$), aggregates, sorts, limit, and set operations.

$\langle$*security types*$\rangle$ τ ::= $s \in \{low, high\}$

$\langle$*phrase types*$\rangle$ ρ ::= τ | τ *set* | τ *exec*

Grammar 2: Grammar for secure information flow analysis

The security type system assigns a *label* to each phrase in a query plan. To model the flow of secure attributes through the query tree, we label each phrase as *low* or *high*. If a phrase is *low*, it does not require oblivious computing and we run it within the source databases like a conventional federated query. A *high* phrase requires oblivious evaluation. We use low to denote computing over public attributes or in a setting where secure evaluation is not needed. Private attributes are handled with *high* operators and we call the set of *high* attributes in a PDN schema h. In addition to judging each phrase as *low* or *high*, the system records the type of each phrase to show whether it is referring to an expression (the default), set of expressions, or an operator execution. This syntax is shown in Grammar 2.

E-BASE $\gamma \vdash e : high$

E-LOW $\dfrac{h \notin attrs(e)}{\gamma \vdash e : low}$

E-SET $\dfrac{E = \{e_1, \ldots, e_n\} \quad \forall i \in \{1 \ldots n\} : (\gamma \vdash e_i : \tau)}{\gamma \vdash E : \tau\ set}$

Figure **4**: Rules for information flow in relational data.

For each typing rule, we say

$$\frac{assumptions}{type\ judgement}$$

to denote the conditions under which we assign a security label to a given phrase. Each type judgement rule is of the form $\gamma \vdash \rho : \tau\ type$. This rule says that in type system γ, we judge phrase ρ as security type $\tau \in \{high, low\}$, with a phrase type of expression, set, or execution.

Figure 4 shows the rules for labeling expressions and sets thereof. In rule E-BASE we say that any phrase may be evaluated as *high*. An expression may be labelled as *low* iff none of the attributes referenced in it are *high*. In E-SET we assign a type to a set of expressions, E, by resolving its elements to a single label, τ. If E contains a mix of *low* and *high* expressions, the system uses type coercion to assign a *high* label to the set.

The type system rules for labeling relational algebra operators in a query plan are shown in Figure 5. We classify each relational algebra operator into one of two categories: tuple-at-a-time or multi-tuple evaluation. Most tuple-at-a-time operators, or ones that emit output by considering each tuple discretely, have no need for secure evaluation unless they consume data from one or more secure children. Operators of this kind are scan, SQL project (for altering the attributes in intermediate results), limit, and filters with *low* predicates. As shown in R-FILTER, a filter needs to be evaluated obliviously iff its predicate changes the operator's output cardinality based on private attributes. Scans are unconditionally public because they are executed locally and their output cardinality is not altered by any attributes.

Multi-tuple operators–including sorts, joins, and set operations–combine data from multiple source engines, and they execute at the level of their most sensitive inputs. The type judgements of this kind are R-AGGREGATE, R-DISTINCT, R-JOIN, R-SETOP, and R-SORT. This, in conjunction with the nesting rules, ensure that these operators leak no information about private data.

All additional operators are covered by R-NEST; it states that a new operator Op' executes at a security level greater than or equal to that of its child. In R-NEST-BIN, we generalize this to operators with two inputs, i.e., joins and set operations. If a binary operator has at least one *high* child, then the operator is judged as *high*. The nesting judgements

$$\text{R-AGGREGATE}\quad \frac{\gamma \vdash E : \tau\ set}{\gamma \vdash agg(E) : \tau\ exec}$$

$$\text{R-DISTINCT}\quad \frac{\gamma \vdash E : \tau\ set}{\gamma \vdash distinct(E) : \tau\ exec}$$

$$\text{R-FILTER}\quad \frac{\gamma \vdash exp : \tau}{\gamma \vdash \sigma_{exp}(E) : \tau\ exec}$$

$$\text{R-JOIN}\quad \frac{\gamma \vdash E : \tau\ set \quad \gamma \vdash E' : \tau\ set}{\gamma \vdash E \bowtie E' : \tau\ exec}$$

$$\text{R-SCAN}\quad \gamma \vdash scan(E) : low\ exec$$

$$\text{R-SETOP}\quad \frac{\gamma \vdash E : \tau\ set \quad \gamma \vdash E' : \tau\ set}{\gamma \vdash E\ op\ E' : \tau\ exec}$$

$$\text{R-SORT}\quad \frac{\gamma \vdash E : \tau\ set}{\gamma \vdash sort(E) : \tau\ exec}$$

$$\text{R-NEST}\quad \frac{\gamma \vdash Op(.) : \tau\ exec \quad \gamma \vdash Op'(.) : \tau\ exec}{\gamma \vdash Op'(Op(.)) : \tau\ exec}$$

$$\text{R-NEST-BIN}\quad \frac{\gamma \vdash Op(.) : \tau\ exec \quad \gamma \vdash Op'(.) : \tau\ exec \quad \gamma \vdash Op''(.) : \tau\ exec}{\gamma \vdash Op''(Op(.), Op'()) : \tau\ exec}$$

Figure **5**: Typing system rules for information flow in relational algebra query tree.

ensure that no subsequent operators leak information about a prior secure computation.

Security Proof We prove the security of our information flow type system by induction. The type system traverses a query tree bottom-up.

Base Case: The leafs of the query tree are all table scans. By R-SCAN every branch of the tree begins as *low*. Table scans are unconditionally oblivious–the contents of the tuples do not alter the data flow of this operator.

Step 1: There are two cases for the parent operator of a scan. The operator may process data one tuple at a time. Recall that projects, limits, and *low* filters are in this category. By R-FILTER, the filter operator ceases to be oblivious when its predicate references non-public attributes. Otherwise, the filter executes at the same security level as its child by R-NEST. Likewise, SQL project and limit operators are deterministic in the cardinality of their outputs, so they reveal no information that is not already visible from running them at the security of their source operator.

The remaining operators–aggregation, distinct, join, set operations, and sort–compute over multiple tuples. Thus, they may execute over the data of mutually distrustful parties. Each of these operators executes at the security level of the most sensitive attribute in their input expressions. By E-SET, the type system determines the security label of the expressions in an operator's fields. The resolver labels the operator with the set's security level, τ, using the operator-specific rules in Figure 5.

Step n: Each subsequent step executes at a security level greater than or equal to that of its children. By R-NEST and R-NEST-BIN, an operator is typed with the label of its source nodes. Therefore, even if an operator only references public attributes, such as `DISTINCT patient_id` in *recurrent c. diff*, the plan reveals no information about the output of secure computation performed by its source operators.

6. SECURE QUERY OPTIMIZATION

Armed with our security model, we can now examine optimizations to improve the performance of SMCQL for secure querying. The optimizer begins with a secure query plan, wherein each operator is labelled *high* or *low* using the type checker in Section 5.2. It then transforms this logical plan into a low-level physical one. Using heuristics, SMCQL reduces a secure plan's use of secure computation and partitions query evaluation into small, scalable units. The optimizations described below are generalizations of distributed query optimization [5]. Since these adjustments change the program structure of a secure plan, we extend the security proof in Section 5.2 for them.

6.1 Scalable Physical Plans

The optimizer identifies opportunities to *slice* its secure query evaluation by partitioning the input data into smaller, more manageable units of computation. This fine-grained partitioning enables us to reduce our secure code complexity, thereby speeding up the secure computation of eligible queries. Slicing also makes it possible to parallelize secure query evaluation. The optimizer assigns each operator in a secure plan to one of three execution modes:

- Plain: Operators are of type *low* and they evaluate in the source database.
- Sliced: *High* operators run securely over tuples that are horizontally partitioned by a public expression.
- Secure: *High* operators executed using a single SMC program run on the inputs of all data providers.

All paths in the tree start with plaintext scans over a database table. When the executor encounters a *high* operator, the engine switches to sliced or secure execution mode. If an operator is in sliced mode, then its ancestors must run in sliced or secure mode so that the query's computation remains oblivious.

Each sliced operator that receives plaintext data bins its input by a *slice key*, or an expression on public attributes upon which the engine divides up a secure operator's computation. Each distinct value associated with a slice key, or *slice partition*, is computed independently. For each relational algebra operator in Grammar 1, we identified if and how it is sliceable.

Joins with public predicates are runnable one slice partition at a time. Likewise, it is possible to slice sorts by all or part of their sort key. Sliced aggregates compute one group-by bin at a time and `DISTINCT` operators break up their computation by the attributes they reference. Since filters and projections work one tuple at a time, they are agnostic to slicing and assume the slice key of their parent or child operator, whichever enables more sliced evaluation. Set operations with any slice key will produce the correct outputs because a non-empty key guarantees that any tuples needing comparison will be grouped together.

Slicing is a powerful optimization because it is composable. The optimizer identifies sequences of secure operators that partition their computation on the same slice key. For example, in the *recurrent c. diff* query every operator after its initial table scan is computed securely and all oblivious operators are sliced by patient ID.

```
Function planExecution
Input: DatabaseOperator o
Output: ExecutionMode e ∈ {Plain, Sliced, Secure}
e = Plain;
if o.label = low then
   return e;
end if
for c ∈ o.children() do
   childMode = planExecution(c);
   if childMode == Secure then
      e = Secure;
   else if childMode == Sliced then
      if o.sharesSliceKey(c) and e ≠ Secure then
         e = Sliced;
      else
         e = Secure;
      end if
   end if
end for
// if o.label = high and all children computed in plaintext
if e == Plain and o.sliceKey ≠ ∅ then
   e = Sliced;
else
   e = Secure;
end if
return e
```

Algorithm 1: Method for assigning execution mode to database operators in a secure query plan.

The optimizer identifies sequences of sliced computation by traversing a query tree bottom-up using Algorithm 1. All *low* operators are run in plaintext. If a *high* operator has only plain children and a nonempty slice key, then it is assigned to slice mode. The optimizer then checks to see if the operator shares a slice key with its parent. Two operators are sliced alike if their slice key is equal, or for joins, at least one side of an equality predicate appears in the slice key of each of its descendants. If a *high* operator does not qualify for slicing, we switch to secure mode.

SECURITY PROOF Each sliced query evaluation is equivalent to running a secure operator without slicing where we insert a filter over public attributes for each distinct slice partition. The optimizer identifies partitions of computation by running `SELECT DISTINCT` <*slice key*> `FROM ...` on each data provider. By R-DISTINCT this query is public. After that, we evaluate each slice partition securely using the previous plan with an added filter for each slice partition. By R-FILTER this filter is public, thus we retain the security properties of the previous plan.

6.2 Minimizing Secure Computation

We now introduce optimizations that reduce a query's use of secure computation. The first delays its entry into SMC by partially computing an operator in a lower execution mode. The second reduces the data that the engine evaluates securely.

Split Operators A *splittable* operator may partition its execution into discrete phases, such as local plaintext followed by distributed secure computation. For example, if we are computing a `COUNT(*)`, the query executor may have each data provider calculate a local partial count and use SMC to add them up. Aggregates and filters are splittable operators.

Each splittable operator has a *low* phase and a *high* phase, and the low phase is executed first. For aggregates, the *low* phase partially computes the aggregate over the input tuples and the *high* one combines the partial aggregates. Filters with conjunctive predicates are splittable into clauses that reference sensitive attributes and ones that do not. All other operators are not trivially splittable.

SECURITY PROOF For an aggregate with no group-by its output is exactly one tuple, and it is trivially oblivious. To handle aggregates with a group-by on sensitive attributes, we take advantage of the PDN's architecture to reduce our overhead. Recall that the parties do not reveal protected or private attributes to one another and that none of the data providers have access to the output of a secure query evaluation. If an operator computes over $|A|$ partial aggregates from Alice and $|B|$ from Bob, the output cardinality of its *high* phase is unconditionally $|A| + |B|$–with null-padding as needed. This fixed output length precludes each data provider from learning the group-by values of a partial aggregate that is not their own. A split filter is analogous to creating two separate selections, one for *low* predicates followed by another for *high* ones. We perform a type judgement on each one with R-FILTER and assign its execution mode accordingly.

Secure Semi-Join We can further reduce our reliance on secure computation by performing it only on slice partitions that are present in greater than one data provider. For example, in *recurrent c. diff* if a given patient ID is present in just one hospital, we evaluate the medical records of this individual locally in plaintext, sending the output of this computation to the honest broker over an encrypted channel. Hence, the query execution plan has two tracks for sliced operators: a secure one for partitions that appear in multiple data providers and plaintext one for all others.

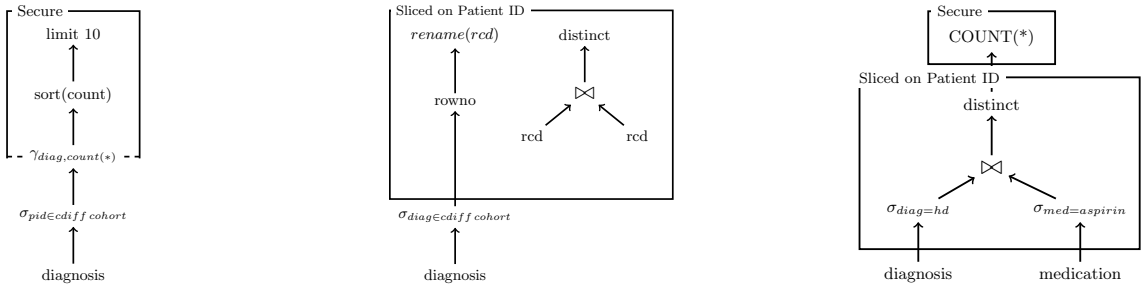
SECURITY PROOF To determine the distributed slice partitions, the honest broker collects the distinct slice partitions as in the proof in Section 6.1. It then takes the intersection of their sets. By R-SETOP the computation of an intersection of two public inputs is public. By using only public values in our slice keys, this optimization reveals no additional information in comparison to its predecessors.

6.3 Optimized Plans

Let's put this all together by examining our optimized SMCQL plans for the queries in our running example. We display the query trees in Figure 6.

Comorbidity starts by executing the diagnosis scan in plaintext on each host, filtering its input tuples for ones in the *c. diff* patient registry. Since this query's slice key is the diagnosis code, a protected attribute, its execution cannot be trivially partitioned. Next, it performs a split aggregate, wherein each host computes a partial diagnosis count for each condition locally. Each site feeds its partial counts into a single unit of secure computation and the parties work together to sum up their counts for overlapping diagnoses, sort them, and take the most common ones.

Recurrent c. diff runs almost entirely in sliced mode. The filter–its first *high* operator–checks for an infection diagnosis. It takes on the slice key of its parent, a window aggregate partitioned on patient ID. The window aggregate numbers the infection diagnoses of each patient, where the diagnoses



(a) Comorbidity (b) Recurrent *C.Diff* (c) Aspirin Count

Figure **6**: Optimized PDN query execution plans for running example.

are sorted on timestamp. We then self-join this table one patient at a time to identify the ones with a recurring diagnosis, and eliminate duplicates using the same slice key.

Aspirin count begins in plaintext with scans on the medication and diagnosis tables. Next, we filter on a protected attribute, and this step is sliced on patient ID. We then join the two tables to identify heart disease patients who received an Aspirin prescription. After that, we eliminate duplicate patient IDs one slice at a time. Finally, we switch to secure mode to count up the patient IDs over all slices.

In summary, we optimize our use of secure computation in three ways. First, SMCQL horizontally partitions the data over public attributes to reduce the time and complexity of our oblivious computing. Second, the query evaluator splits up query operators to prolong its time in a less expensive execution mode. Lastly, the optimizer identifies tuples that do not require distributed secure computation and evaluates them within their source DBMS.

7. RESULTS

We now verify that SMCQL produces efficient query plans using the workload introduced in Section 2.2. We first review our experimental design. Next, we explore the effectiveness of this system's heuristics-based optimizer at managing our use of SMC. Then we examine the impact of SMC on assorted database operators. After that, we test the scalability of SMCQL as it executes over data of increasing size. Lastly, we reveal the performance profile of this system in comparison to a hypothetical federated database where the parties trust one another.

7.1 Experimental Setup

We evaluate SMCQL on medical data from two Chicago-area hospitals in the HealthLNK data repository [29] over one year of data. This dataset has 500,000 patient records, or 15 GB of data. To simplify our experiments, we use a public patient registry for common diseases that maintains a list of anonymized patient identifiers associated with these conditions. We filter our query inputs using this registry.

SMCQL's query executor is built atop PostgreSQL 9.5 running on Ubuntu Linux. We evaluated our two-party prototype on 8 servers running in pairs. The servers each have 64 GB of memory, 7200 RPM NL-SAS hard drives, and are on a dedicated 10 Gb/s network. Our results report the average of three runs per experiment. Unless otherwise specified, the results show the end-to-end runtime of a query, including its plaintext and secure execution. All figures display their runtimes on a logarithmic scale.

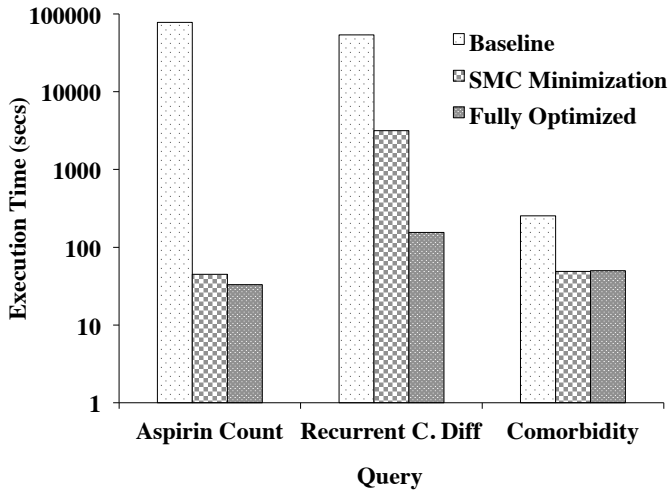


Figure **7**: System performance on sampled data.

7.2 Secure Query Optimization

We now examine the role of SMCQL's optimizations in making PDN queries run with efficiency and scalability. The results in this section are from query executions over samples of HealthLNK data with 50 tuples per data provider. The samples were taken uniformly at random, with the restriction that each sample has at least one distributed slice partition so that the query uses secure computation.

We evaluate the optimizer's heuristics with three tests. First we use a baseline of fully secure execution with no optimizations. This test is analogous to a query execution where all of the attributes in a PDN's schema are protected. The baseline has the same configuration as the results in Section 4.1. The second approach, SMC minimization, evaluates optimizations that reduce the subtree of a query's plan that is executed securely and the data processed therein. For *comorbidity*, this tests split operators. In *aspirin count* and *recurrent c. diff*, it showcases the secure semi-join. Lastly, we measure the system's performance when fully optimized. These results show the system performance with the previous optimizations plus sliced execution.

Figure 7 displays the runtime for each query end-to-end. It is clear that our baseline execution is very slow, even for modest data sizes. Leveraging the PDN's security policy is important for efficient query evaluation in this setting. The SMC minimization techniques substantially improve the system's performance for all queries. With the split operator evaluation, *comorbidity* runs 5X faster than the baseline.

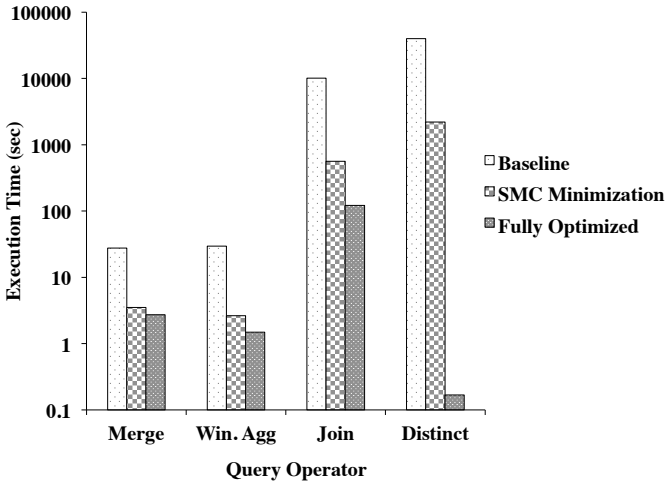


Figure **8**: Optimized performance of *recurrent c. diff*.

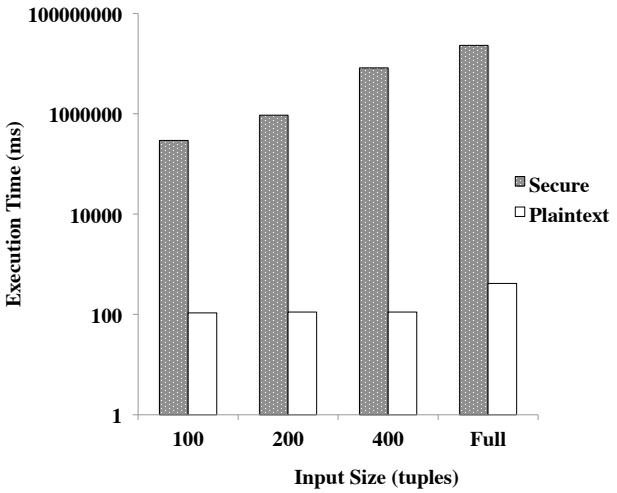


Figure **9**: Runtime of *comorbidity* on increasing data sizes.

This query has no sliced operators, so it realizes no additional speedup in the fully optimized test.

We see the most dramatic performance improvement from *aspirin count*. The secure semi-join heuristic substantially reduces the number of tuples processed in this query and it has a speedup of 1700X over the baseline. This query benefits less from sliced execution because its tuples are not evenly distributed over the slice partitions. This skewed distribution reduces the effectiveness of slicing because its largest partitions are comparable in size to the single partition run in the SMC minimization experiment.

In contrast, *recurrent c. diff*'s SMC minimization has a less pronounced improvement of 17X over the baseline. This query has fewer opportunities for optimization because its plan has a simpler tree of operators in comparison to *aspirin count*. On the other hand, *recurrent c. diff* responds better to slicing owing to its tuples having an even distribution over the query's partition keys. Its fully-optimized version is nearly 350X faster than the baseline.

Let's drill down to how SMCQL's optimizations improve the performance of *recurrent c. diff*. We show the breakdown of this query's runtime on each of its secure operators in Figure 8. We see a gradual performance improvement as we add more optimizations. The merge and window aggregate operators both make simple passes over the data, and their oblivious runtimes are proportional to the size of their inputs. The unoptimized join is costly because it computes the cross product of its input relations. The `DISTINCT` operator has a substantial SMC-induced performance penalty owing to its large input from the exhaustive join.

Secure semi-join noticeably improves the performance of each operator. By reducing the cardinality of the secure input data, the first two operators run much faster. This is primarily owing to the reduced I/O costs associated its oblivious memory accesses on smaller arrays of data. The join shows a reduction in runtime due to its performing fewer tuple comparisons. Likewise, the final distinct operator has a strong performance improvement over the baseline owing to its smaller input data.

We see that slicing bolsters the performance of the *recurrent c. diff* query. Building multiple small oblivious tuple arrays speeds up our data ingest. Window aggregate enjoys even greater gains since slicing partitions the data by its group-by clause–further simplifying its secure code. The join also becomes faster because it is computing the cross product over fewer tuples. The cost of finding the distinct patient IDs goes to less than a second because each slice partition simply checks if it has a non-empty array of tuples.

7.3 System Scale Up

We now test the performance of SMCQL as it scales to larger input data. For this experiment, we ran the *comorbidity* query. We artificially limited the size of the SMC input–the partial counts of comorbid conditions in the *c. diff* cohort–to 100, 200, and 400 tuples. We ran also ran this test on the full dataset; it had nearly 800 diagnosis codes with each party supplying around 650 partial counts. We report runtimes for this experiment in milliseconds.

Figure 9 shows our scale up results. The plaintext execution time of this query grows slowly as we scale up. The dominant cost of plaintext *comorbidity* is in local computing, wherein it scans each table, filters for the cohort, and computes the local partial count.

In contrast, *comorbidity*'s secure execution shows a substantial increase in duration as the framework computes over more data. This is primarily caused by increasing costs associated with larger oblivious tuple arrays instances for storing and accessing the data. Because the secure array shuffles the data every time we access an element, its cost increases in proportional to its input size. In addition, the query execution takes longer because its secure operators perform more comparisons among the inputs to sum up partial counts.

The smallest scale run in this analysis has a slowdown of 2,700X in comparison to its plaintext execution. With an input size of 400 tuples, the query's duration jumps to nearly 74,000X its conventionally executed counterpart. When we run over the entire SMC input, our runtime goes to 6.5 hours or 56,000X the baseline.

Zooming out to look at the system's end-to-end performance, we now appraise the effects of its entire query optimization system. For this experiment, we used a full year of health records. Our results are shown in Figure 10. These results are a scaled-up evaluation of the fully optimized test in Figure 7.

All of these queries run 5–6 orders of magnitude more slowly than their plaintext counterparts. SMCQL executes

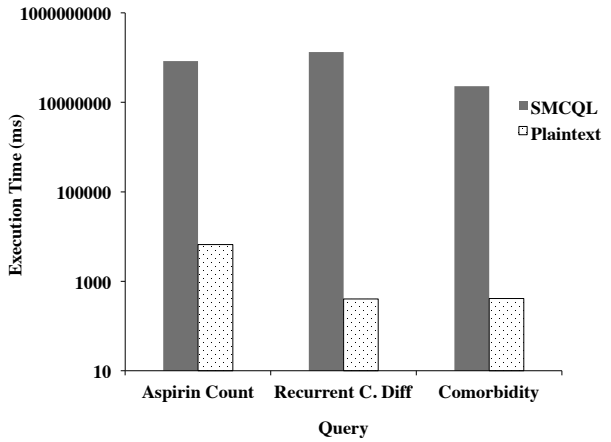


Figure **10**: Runtime on full year of HealthLNK records.

recurrent c. diff in 37 hours, *aspirin count* in 23 hours, and *comorbidity* in 6 hours. These queries are accessing a collective 42 million diagnoses and 23 million medication records. By using fine-grained partitions of the underlying dataset, the SQL operators scale reasonably to large volumes of data.

We see noticeably higher runtimes for *aspirin count* and *recurrent c. diff*. Both of these take upwards of a single day to run. The primary source of the slowdown arises from their `join` operators that have hundreds of input tuples and they perform $O(n^2)$ comparisons for oblivious evaluation. More work is needed to manage this challenge of securing large in-memory data structures.

One potential solution for scaling SMCQL to large datasets is to partition the work into smaller units and use the honest broker to assemble the results. This approach would require an analytical cost model to identify how finely to partition the work based on the rate at which the honest broker can accept and assemble the final results. It would also require the planner to analyze the cardinality of intermediate results to ensure that the honest broker does not receive unauthorized access to protected or private attributes. In our running example, this would mean ensuring that intermediate results received by the honest broker were k-anonymous in their protected attributes and had no private data.

In summary, our results demonstrate that SMCQL provides practical secure query evaluation over SQL for mutually distrustful parties. We found that joins are the most costly oblivious operator and that their large output cardinalities impact the execution time of their parent operators. Slicing reduces the slowdown associated with secure query execution, and it is most effective for datasets where the tuples are evenly partitioned over the slice key space. Our results show that query runtime grows rapidly as the system scales up to larger datasets.

8. RELATED WORK

Several approaches to secure query processing exist in the literature. Most of this work is designed for outsourced computation [20, 21, 30]. In contrast, a PDN keeps query processing in the database as much as possible and performs SMC within the hosts from which the data originated. This enables data providers who do not trust the cloud (or some other third party) to share information.

CryptDB [30] stores its data on a remote server using homomorphic encryption. Also, in [1], the authors proposed outsourcing database query evaluation by storing the entire database as secret shares spread over two or more cloud providers. The work of [20, 21] computes database filters and joins using SMC that is run on trusted third-party servers. Sharemind [4] supports federated databases, but does not allow for arbitrary SQL queries and uses encrypted cloud storage for their data. Because SMCQL runs SMC locally on the hosts that provide the data, we make our queries faster by minimizing our use of SMC. This is not possible in outsourced systems since they do not have access to the unencrypted input data.

Chow, et al. [7] proposed outsourced SMC for SQL queries over data from multiple source systems. Our work removes the reliance on trusted third parties by performing the secure computation within the hosts of each data provider. In addition, the existing work did not support arbitrary tuple comparisons (e.g., $\leq$, $\geq$) and it leaked information for nested queries. SMCQL has both of these features.

There is a plethora of domain-specific programming languages for working with SMC, including ObliVM [23], VMCrypt [24], TASTY [13], and FAIRPLAY [25]. They all generate secure code for procedural programs. These systems rely on the user explicitly specifying how to manage and compare the data from each party. In contrast, SMCQL seamlessly injects SMC into an existing declarative language, SQL.

Kerschbaum [16, 17] broke ground by optimizing the use of SMC in imperative programs using a secure type flow system. We extend their approach with a security type system for SQL. This existing work operated in the context of all parties learning the plaintext output of secure computation. The authors use this information to deduce the intermediate states of a SMC program that are safe to reveal to the data providers. Rastogi [33] generalizes and expands upon this work using a knowledge inference approach. In our research, PDN members do not have access to the output of secure computation. The optimization techniques used here are more conservative than their predecessors in how they handle intermediate results.

9. CONCLUSIONS AND FUTURE WORK

In this work, we introduce the private data network (PDN), a novel generalization of federated database systems for mutually distrustfully parties. We propose SMCQL, a framework for translating SQL queries into secure multiparty computation primitives for evaluating PDN queries. We introduce several strategies for optimizing PDN query plans, including partitioning data before securely computing on it and partially evaluating database operators in plaintext.

Our results demonstrate that by partitioning query processing at a fine granularity, we offer usable performance for complex PDN workloads. Our PDN queries, that are based on a real medical use case and evaluated on de-identified medical records, complete within a reasonable time even in the presence of large datasets. We designed this system to be practical. To this end, we are preparing SMCQL for an open source release. In addition, we are collaborating with stakeholders in a clinical data research network to start testing it in the field.

There are numerous opportunities for future research on PDNs. We are investigating how to generalize SMCQL to

three or more parties. Scaling out to more parties requires changes to our SMC protocols, as well additional algorithms for assigning work to PDN nodes. PDNs with a large number of parties also introduce opportunities for parallelizing their plans. Another future research direction is to identify automatic SQL rewrite rules that further delay our entry into SMC by reordering commutative database operators. Optimizing the SMC primitives and protocols used in a PDN plan is another future avenue of inquiry. There are many choices for how to implement the secure computation within a PDN, including randomized key protocols [7] and garbled ORAMs [10]. In addition, there are garbled circuit protocols optimized for reduced CPU time, minimizing network bandwidth, and for scaling to a large number of parties. Lastly, extensive research exists on SMC protocols for specific algorithms, such as linear regression and matrix multiplication, but we are aware of no work on improving the performance of secure SQL operators.

10. ACKNOWLEDGMENTS

We thank Katie Jackson and Jess Joseph Behrens for their guidance and assistance with CAPriCORN and HealthLNK data. We appreciate the HealthLNK team for sharing de-identified electronic health record data for this study. We are grateful to Ben Slivka and Mike Stonebraker for their feedback on early drafts of this work.

11. REFERENCES

[1] G. Aggarwal, M. Bawa, P. Ganesan, H. Garcia-Molina, K. Kenthapadi, R. Motwani, U. Srivastava, D. Thomas, and Y. Xu. Two can keep a secret: A distributed architecture for secure database services. *CIDR*, 2005.

[2] R. Agrawal and R. Srikant. Privacy-preserving data mining. *2000 SIGMOD*, 29(2):439–450, 2000.

[3] A. Al-Lawati, D. Lee, and P. McDaniel. Blocking-aware private record linkage. In *Proceedings of the 2nd international workshop on Information quality in information systems*, pages 59–68. ACM, 2005.

[4] D. Bogdanov, M. J. oemets, S. Siim, and M. Vaht. Privacy-preserving tax fraud detection in the cloud with realistic data volumes. Technical Report T-4-24, Cybernetica AS, 2016.

[5] S. Chaudhuri. An overview of query optimization in relational systems. In *ACM PODS*, pages 34–43. ACM, 1998.

[6] D. Chaum, C. Crépeau, and I. Damgård. Multiparty Unconditionally Secure Protocols. *STOC*, pages 11–19, 1988.

[7] S. S. Chow, J.-H. Lee, and L. Subramanian. Two-party computation model for privacy-preserving queries over distributed databases. In *NDSS*, 2009.

[8] C. Dwork. Differential privacy. *Proceedings of the 33rd International Colloquium on Automata, Languages and Programming*, pages 1–12, 2006.

[9] K. E. Emam. Heuristics for de-identifying health data. *IEEE Security & Privacy*, 6(4):58–61, 2008.

[10] C. Gentry, S. Halevi, S. Lu, R. Ostrovsky, M. Raykova, and D. Wichs. Garbled ram revisited. In *Annual International Conference on the Theory and Applications of Cryptographic Techniques*, pages 405–422. Springer, 2014.

[11] O. Goldreich. Towards a theory of software protection and simulation by oblivious rams. In *STOC*, pages 182–194, New York, NY, USA, 1987. ACM.

[12] O. Goldreich, S. Micali, and A. Wigderson. How to Play Any Mental Game. *Stoc '87*, pages 218–229, 1987.

[13] W. Henecka, A.-R. Sadeghi, T. Schneider, I. Wehrenberg, et al. Tasty: tool for automating secure two-party computations. In *CCS*, pages 451–462. ACM, 2010.

[14] A. F. Hernandez, R. L. Fleurence, and R. L. Rothman. The ADAPTABLE Trial and PCORnet: shining light on a new research paradigm. *Annals of internal medicine*, 163(8):635–636, 2015.

[15] A. F. Karr, X. Lin, A. P. Sanil, and J. P. Reiter. Secure Regression on Distributed Databases. *Journal of Computational and Graphical Statistics*, 14(2):263–279, 2005.

[16] F. Kerschbaum. Automatically optimizing secure computation. In *ACM CCS*, pages 703–714. ACM, 2011.

[17] F. Kerschbaum. An information-flow type-system for mixed protocol secure computation. *Proceedings of the 8th ACM SIGSAC symposium on Information, computer and communications security - ASIA CCS '13*, page 393, 2013.

[18] A. Kho, J. Cashy, K. Jackson, A. Pah, S. Goel, J. Boehnke, J. Humphries, S. Kominers, B. Hota, S. Sims, B. Malin, D. French, T. Walunas, D. Meltzer, E. Kaleba, R. Jones, and W. Galanter. Design and implementation of a privacy preserving electronic health record linkage tool in chicago. *Journal of the American Medical Informatics Association*, 22(5):1072–1080, 2015.

[19] A. N. Kho, D. M. D. Hynes, S. Goel, A. E. Solomonides, R. Price, B. Hota, S. A. Sims, N. Bahroos, F. Angulo, W. E. Trick, and Others. CAPriCORN: Chicago Area Patient-Centered Outcomes Research Network. *Journal of the American Medical Informatics Association*, 21(4):607–611, 2014.

[20] S. Laur, R. Talviste, and J. Willemson. From oblivious AES to efficient and secure database join in the multiparty setting. *Lecture Notes in Computer Science*, 7954 LNCS:84–101, 2013.

[21] S. Laur, J. Willemson, and B. Zhang. Round-efficient Oblivious Database Manipulation. *ISC'11*, pages 262–277, 2011.

[22] I. Lazrig, T. Moataz, I. Ray, I. Ray, T. Ong, M. Kahn, F. Cuppens, and N. Cuppens. Privacy preserving record matching using automated semi-trusted broker. In *IFIP Annual Conference on Data and Applications Security and Privacy*, pages 103–118. Springer, 2015.

[23] C. Liu, X. S. Wang, K. Nayak, Y. Huang, and E. Shi. ObliVM : A Programming Framework for Secure Computation. *Oakland*, pages 359–376, 2015.

[24] L. Malka. VMCrypt: modular software architecture for scalable secure computation. In *CCS*, pages 715–724. ACM, 2011.

[25] D. Malkhi, N. Nisan, B. Pinkas, Y. Sella, et al. Fairplay-secure two-party computation system. In *USENIX Security Symposium*, volume 4. San Diego, CA, USA, 2004.

[26] F. D. McSherry. Privacy integrated queries: an extensible platform for privacy-preserving data analysis. In *2009 ACM SIGMOD*, pages 19–30. ACM, 2009.

[27] R. Motwani, S. U. Nabar, and D. Thomas. Auditing SQL queries. In *ICDE*, pages 287–296. IEEE, 2008.

[28] PCORI. Characterizing the Effects of Recurrent Clostridium Difficile Infection on Patients. *IRB Protocol*, ORA: 14122, 2015.

[29] PCORI. Exchanging de-identified data between hospitals for city-wide health analysis in the Chicago Area HealthLNK data repository (HDR). *IRB Protocol*, 2015.

[30] R. Popa and C. Redfield. CryptDB: protecting confidentiality with encrypted query processing. *SOSP*, pages 85–100, 2011.

[31] R. Ramakrishnan and J. Gehrke. *Database management systems*. McGraw-Hill, 2000.

[32] A. Rastogi, M. A. Hammer, and M. Hicks. Wysteria: A programming language for generic, mixed-mode multiparty computations. In *2014 IEEE Symposium on Security and Privacy*, pages 655–670. IEEE, 2014.

[33] A. Rastogi, P. Mardziel, M. Hicks, and M. A. Hammer. Knowledge inference for optimizing secure multi-party computation. In *Eighth ACM SIGPLAN workshop on Programming languages and analysis for security*, pages 3–14. ACM, 2013.

[34] A. Sabelfeld and A. C. Myers. Language-based information-flow security. *IEEE Journal on selected areas in communications*, 21(1):5–19, 2003.

[35] F. B. Schneider, G. Morrisett, and R. Harper. A language-based approach to security. In *Informatics*, pages 86–101. Springer, 2001.

[36] P. Upadhyaya, M. Balazinska, and D. Suciu. Automatic enforcement of data use policies with DataLawyer. In *SIGMOD*, pages 213–225. ACM, 2015.

[37] D. Volpano, C. Irvine, and G. Smith. A sound type system for secure flow analysis. *Journal of computer security*, 4(2-3):167–187, 1996.

[38] A. C. Yao. Protocols for secure computations. *FoCS*, pages 1–5, 1982.

[39] S. Zahur and D. Evans. Obliv-C: A language for extensible data-oblivious computation. *Cryptology ePrint Archive, Report 2015/1153*, 2015.