Enabling Personal Consent in Databases

George Konstantinidis
Electronics and Computer Science
University of Southampton, UK
g.konstantinidis@soton.ac.uk

Jet Holt
Electronics and Computer Science
University of Southampton, UK
hello@jetholt.com

Adriane Chapman
Electronics and Computer Science
University of Southampton, UK
adriane.chapman@soton.ac.uk

ABSTRACT

Users have the right to consent to the use of their data, but current methods are limited to very coarse-grained expressions of consent, as "opt-in/opt-out" choices for certain uses. In this paper we identify the need for fine-grained consent management and formalize how to express and manage user consent and personal contracts of data usage in relational databases. Unlike privacy approaches, our focus is not on preserving confidentiality against an adversary, but rather cooperate with a *trusted* service provider to abide by user preferences in an algorithmic way. Our approach enables data owners to express the intended data usage in formal specifications, that we call *consent constraints*, and enables a service provider that wants to honor these constraints, to automatically do so by filtering query results that violate consent; rather than both sides relying on "terms of use" agreements written in natural language. We provide formal foundations (based on provenance), algorithms (based on unification and query rewriting), connections to data privacy, and complexity results for supporting consent in databases. We implement our framework in an open source RDBMS, and provide an evaluation against the most relevant privacy approach using the TPC-H benchmark, and on a real dataset of ICU data.

PVLDB Reference Format:
George Konstantinidis, Jet Holt, and Adriane Chapman. Enabling Personal Consent in Databases. PVLDB, 15(2): 375 - 387, 2022.
doi:10.14778/3489496.3489516

PVLDB Artifact Availability:
The source code, data, and/or other artifacts have been made available at https://github.com/georgeKon/enabling-personal-consent/.

1 INTRODUCTION

We are witnessing an emerging business, governmental, technological and cultural interest in developing frameworks for allowing citizens to choose how their personal data is used. Traditional security, data privacy or access control approaches, such as [14, 15, 25, 35, 36, 38, 45], protect personal information or ensure confidentiality of individuals' identities against an *adversary*. Nevertheless, we increasingly see privacy breaches that are due to failures to implement privacy agreements between relatively *trusted, non-adversarial* parties. We refer to this setting, where data is released to a non-adversary party who has incentive to abide by the privacy or process agreement, as *collaborative privacy*.

This work is licensed under the Creative Commons BY-NC-ND 4.0 International License. Visit https://creativecommons.org/licenses/by-nc-nd/4.0/ to view a copy of this license. For any use beyond those covered by this license, obtain permission by emailing info@vldb.org. Copyright is held by the owner/author(s). Publication rights licensed to the VLDB Endowment.
Proceedings of the VLDB Endowment, Vol. 15, No. 2 ISSN 2150-8097.
doi:10.14778/3489496.3489516

Today, the need for collaborative privacy is omnipresent: an individual gives her own data to a social media provider and trusts that the provider will respect her privacy settings, patients give data to clinical trials and trust the researchers to respect their preferences, customers register data in commercial websites and trust that the website will respect their privacy, etc.. Such privacy enforcement does not rely on encryption or partial revelation of data. Instead, users give their personal data in its entirety and trust the service providers to respect their privacy as described in "Terms and Conditions" documents or other custom agreements.

Commonly, these agreements are monolithic and top-down policies which protect the interest of organizations, rather than data subjects. They are written in natural language and they are enforced in an "extra-algorithmic", ad-hoc manner. For example, in clinical trials, researchers elicit consent from patients through questionnaires and then give the list of patient's terms to a technical team to explicitly implement these preferences in the data workflow. There lies a technological gap, between the agreement in natural language and its implementation into code. Indeed, the only automation or automatic customisation that is usually taking place is in the form of predefined, coarse-grained opt-in/out choices.

Starting from query answering in RDBMSes, we advocate for the need to support data usage agreements that are machine-processable, fine-grained, and bottom-up. In order to do this, we have to revise, and can not directly re-use, classic data privacy technologies since these have been designed to protect against collusions and avoid returning (even non-private) data if this could be used for a subsequent privacy breach. For example, a clinical patient might hand over their email to find out the results of a clinical research but not for other purposes. Data privacy would not release the email at all; collaborative privacy, on the other hand, intends to automate organisations willingly enforcing consent and privacy preferences.

Thus we propose the notion of personal, fine-grained consent that allows individuals to express their own data sharing policies. In particular, we allow users to describe combinations of personal information for which consent is not given. We call these statements *consent constraints*. In Figure 1, users with ID numbers '4872' and '2321' trust the service provider with all their data, but do not consent for their Birthdate to be associated with their disease Diagnosis. They do not mind however sharing each of these attributes in isolation or in combination with other attributes, again with the understanding that these two attributes are not going to be combined to violate their consent later. We develop an algorithm and a system that service providers can use to honor these constraints, by removing non-consented tuples from their queries. That is, rather than rejecting, we partially answer a query to the fullest extent that it does not explicitly violate consent.

We focus on expressing both our consent constraints and the service providers queries in the language of conjunctive queries (CQs),

Don’t give back (Birthdate, Disease) associations

Patients

ID	Name	Gender	Birthdate	PhoneNo	DiagnosisYear	Disease
4872	Smith	M	28/02/89	2153409001	2017	Hepatitis
2321	Jones	M	04/04/78	3456008984	2014	Heart Dis.
1312	Harris	F	23/02/07	2329345674	2007	Heart Dis.
7463	Johnson	F	02/06/82	4956732833	2018	Flu
2322	Walker	M	12/02/76	5457853322	2014	HIV

Figure 1: A constraint that can be represented by a query selecting the users with IDs 4872 or 2321 and then projecting on ‘Birthdate’ and ‘Disease’.

which corresponds to the “core” (SELECT-PROJECT-JOIN) fragment of SQL, and is powerful enough to express fine-grained personal consent and top-down policies. CQs is a natural first generalization of opt-in/out choices (the only-machine processable consent language so far), and has been employed as a privacy language in multiple works and systems [7, 10, 11, 22, 23, 33, 36, 39, 42]. It is important to note that focusing on CQs for the service provider’s language allows for more expressive solutions as well. A query containing an aggregate operation can be checked and filtered for consent only on its CQ part before the aggregation takes place.

Our contributions include the following. In Section 3, we formalize the semantics of consent constraints and consent-abiding query answers via a novel usage of provenance annotations; we also lay a formal connection to data privacy and the guarantees our framework provides. In Section 4, we present an algorithm to implement our semantics without relying on annotations, but via query rewriting: given a CQ and a set of constraints we produce an SQL query that computes the consent-abiding answer of our original query on *any* database; we then discuss strategies to propagate consent. We implement and evaluate our algorithms on top of PostgreSQL, and compare our framework (in Sec. 5) with the most relevant "opt-in/out" work from data privacy [33] exhibiting a comparably efficient but more expressive framework. We experiment with both TPC-H and real data scaling our system to answering queries in the face of thousands of constraints within seconds.

2 PRELIMINARIES

We use the well-known mathematical logic and relational calculus notions of *constants*, *variables*, *predicates*, *terms* (which are either constants or variables), *attributes* and *tuples* [4]. For any expression e, $vars(e)$, and $terms(e)$ denote the sets of variables and terms that appear in e respectively. Atoms are of the form $P(\vec{t})$ with P a predicate/relation name. By $\vec{t}[i]$ we denote the ith element of tuple $\vec{t}$. A ground atom, with only constant terms, is a *fact*. A database *relation* is a set of facts. A relation name R and its arity, is a *relation schema*. A set of relation schemas is a *database schema*. A a set of facts of different relations is *database instance*.

A *substitution* $\sigma = \{v_1 \rightarrow t_1, ..., v_n \rightarrow t_n\}$ is a mapping of variables to terms. For any expression e, such as a tuple, an atom, or a set of atoms, $\sigma(e)$ is obtained by simultaneously replacing each occurrence of a variable v_i in e, that also occurs in the domain of σ, with $\sigma(v_i) = t_i$; variables outside the domain of σ remain unchanged (as, obviously, do constants). For example, consider atom $P(v_1, v_2, v_3, v_4, \text{“John”})$ and $\sigma = \{v_1 \rightarrow v_2, v_2 \rightarrow v_1, v_3 \rightarrow v_4\}$, then $\sigma(P(v_1, v_2, v_3, v_4, \text{“John”})) = P(v_2, v_1, v_4, v_4, \text{“John”})$. A substitution σ is a *homomorphism* of a set of atoms S_1 into a set S_2, if the domain of σ is the set of all variables occurring in S_1 and $\sigma(S_1) \subseteq S_2$.

We write CQs in the rule form $q(\vec{v}, \vec{c}) \leftarrow P_1(\vec{u_1}, \vec{c_1}), \ldots, P_n(\vec{u_n}, \vec{c_n})$ where q is the name of the answer relation, $P_1, ..., P_n$ are database relation names, $\vec{v}, \vec{u_i}$ are tuples of variables, $\vec{c}, \vec{c_i}$ are tuples of constants and the query is *safe*, i.e., $\vec{v} \subseteq \bigcup_{i=1}^{n} \vec{u_i}$ and $\vec{c} \subseteq \bigcup_{i=1}^{n} \vec{c_i}$. Atom $q(\vec{v}, \vec{c})$, is called the *head* of the query, denoted $head(q)$, while the *body* is $body(q) = \{P_1(\vec{u_1}, \vec{c_1}), ..., P_n(\vec{u_n}, \vec{c_n})\}$. The vector of head variables, $\vec{v}$, is the tuple of *free* or *distinguished* variables of the query. Notice that, for technical reasons we allow query constants to also appear in the head. We denote joins with the same variable repeated in different atoms. We refer to a CQ with head terms $\vec{x}$ by its head $q(\vec{x})$ or even by q if its head terms are not important. A query with no head terms is *boolean*, and it is false if the result is empty or true if the result contains the empty tuple. For two queries $q(\vec{x})$ and $p(\vec{z})$ of arity n, we say that q and p have the same *result schema* if, for all $i \in [1, n]$, there are atoms of the same predicate P, $P(\vec{y})$ in $body(q)$ and $P(\vec{w})$ in $body(p)$, such that $\vec{x}[i]$ and $\vec{z}[i]$ appear in the same position in $P(\vec{y})$ and $P(\vec{w})$ respectively. For a query $q(\vec{x})$, whenever there is a substitution σ such that $\sigma(\vec{x}) = \vec{b}$, we denote the query obtained by replacing $\vec{x}$ with $\vec{b}$ in q, by $q(\vec{b})$ or $\sigma(q(\vec{x}))$.

Consider schema: `{Patient(pid,name),HasDoctor(pid,did)}`. Query `SELECT name FROM Patient, HasDoctor WHERE Petient.pid=HasDoctor.pid AND HasDoctor.did = ‘723’` corresponds to the rule-notation formula: $q(y) \leftarrow Patient(x, y), HasDoctor(x, 723)$. Given a CQ $q(\vec{x})$ and a database instance D, $q(D)$ denotes the result of evaluating q over D; this is the set of all tuples of constants $\vec{a}$ such that $q(\vec{a})$ holds in D. For every answer tuple $\vec{a}$ of $q(\vec{x})$ over D, there is a homomorphism h of $body(q)$ into D (sometimes we write of q into D) such that $h(\vec{x}) = \vec{a}$.

For two conjunctive queries, $q_1(\vec{t_1}), q_2(\vec{t_2})$, we say that q_2 is *contained* in q_1, denoted by $q_2 \sqsubseteq q_1$, iff for all databases D, $q_2(D) \subseteq q_1(D)$ (strict query containment $\sqsubset$ is defined in the obvious way). For all q_1, q_2, $q_2 \sqsubseteq q_1$ iff there is a containment mapping from q_1 to q_2 [21]. A *containment mapping* from q_1 to q_2 is a homomorphism h:$vars(q_1) \rightarrow terms(q_2)$ such that: (1) for all atoms $\alpha \in body(q_1)$, it holds that $h(\alpha) \in body(q_2)$, and (2) $h(head(q_1)) = head(q_2)$ (modulo the answer relation names of q_1, q_2).

For technical reasons, we mix our rule-notation for CQs with notation from relational algebra and in particular the projection operator. Given a relation R with size $|R| = n$, the standard definition of the projection of R on $A = i_1, i_2, ..., i_m$, a list of attribute positions of R ($i_j \in [1, n]$), is $\pi_A(R) = \{\langle a_1, ..., a_m\rangle \mid \text{there is a tuple } \langle b_1, ..., b_n\rangle$ in R with $a_1 = b_{i_1}, ..., a_m = b_{i_m}\}$. We also apply projection on single tuples: given a tuple $\vec{t} = \langle t_1, ..., t_n\rangle$, and $A = i_1, ..., i_m$ a list of positions of $\vec{t}$ (possibly with repetitions), $\pi_A(\vec{t})$ is the tuple $\langle a_1, ..., a_m\rangle$ such that $a_1 = t_{i_1}, ..., a_m = t_{i_m}$. Given a CQ $q(\vec{t})$, $\pi_A(q(\vec{t}))$ denotes a new query with the same body and head $q(\pi_A(\vec{t}))$.

Lastly, we make use of some logic-programming notions [34]. Let two substitutions $\theta = \{u_1 \rightarrow s_1, \ldots, u_m \rightarrow s_m\}$ and $\sigma = \{v_1 \rightarrow t_1, \ldots, v_n \rightarrow t_n\}$. The *composition* $\sigma \circ \theta$ is obtained from the substitution $\{u_1 \rightarrow \sigma(s_1), \ldots, u_m \rightarrow \sigma(s_m), v_1 \rightarrow t_1, \ldots, v_n \rightarrow t_n\}$ after deleting all those elements $u_i \rightarrow \sigma(s_i)$ for which $u_i = \sigma(s_i)$,

and also deleting all those $v_j \rightarrow t_j$ for which $v_j \in \{u_1, \ldots, u_m\}$. Given a set of atoms S, a *unifier* is a substitution σ such that $\sigma(S)$ is a singleton (i.e., all atoms "merge" into becoming the same one, after substitution). A *most general unifier* or mgu for a set of atoms S is a unifier σ such that any other unifier ρ can be obtained by the composition of the mgu σ with a substitution θ, i.e., $\rho = \theta \circ \sigma$. For example, for $S = \{P(x, y, z), P(v, w, w)\}$ a most general unifier is $\sigma = \{x \rightarrow v, y \rightarrow z, w \rightarrow z)\}$ since $\sigma(P(x, y, z)) = \sigma(P(v, w, w)) = P(v, z, z)$, and all other unifiers must be more "specific", e.g., unifier $\rho = \{x \rightarrow z, y \rightarrow z, w \rightarrow z)\}$, for which $\rho(S) = \{P(z, z, z)\}$ can be obtained by composing the mgu with substitution $\{v \rightarrow z\}$.

3 SEMANTICS OF CONSENT

Within relational databases, it is natural to imagine a set of sharing defaults where a user is specifying what not to reveal. We focus on negative statements of consent called *consent constraints*, formalized as CQs with "negative" semantics, i.e, whose answers are not consented to be revealed. Table 1 contains some consent statements users may make over their health data, and the associated constraints. The third row corresponds to the example of Figure 1.

We discuss our desired semantics starting from the most related data privacy approach that considers negative/secret queries against an adversary, a setting commonly referred to as perfect privacy [36].

3.1 From Adversarial to Collaborative Privacy

Perfect privacy considers whether a CQ view V (or a set of views) exposes answers, known as critical tuples, to a secret CQ query q. A tuple $\vec{t}$ is *critical* for a query q, if there exists a database instance I that contains $\vec{t}$, such that the answer of the query changes if $\vec{t}$ is removed from I; that is, $\vec{t}$ is critical for q if $q(I \setminus \{\vec{t}\}) \neq q(I)$ [36]. The set of all critical tuples for a query q is denoted as $crit(q)$.

Definition 3.1. [36] A CQ q is *secure* with respect to a CQ view V, denoted $q|V$, if $crit(q) \cap crit(V) = \emptyset$.

Our consent constraints can be seen, at a first glance, as secret queries. However, there are three main properties that our non-adversarial setting aims to offer differently to perfect privacy:

Avoid collusion explicitly. Perfect privacy aims to implicitly avoid future collusion when answering a query: a secret query in [36] is secure with respect to different individual views in isolation iff it is secure against any possible combination of these views. On the other hand, for collaborative privacy, we will avoid collusion by having all service provider queries and their combinations willingly go through the framework and not maliciously explored outside. As we discuss below this is beneficial in multiple ways.

Provide fine-grained consent statements. Perfect privacy is coarse-grained: the critical tuple semantics characterizes all the tuples, and their values, in the image of a secret query's homomorphism as critical/sensitive. In essence, no part of a tuple or a query answer is revealed even if only a small subset of tuples/cell values are essentially private. In fact, there is no distinction between secret boolean queries and secret queries with free variables (they characterize the same tuples as critical). On the other hand, our explicit treatment of collusions, allows us to develop much more fine-grained and flexible semantics. We aim to annotate particular combinations of attributes as private; projections in our constraints make a difference since they characterize particular cells within a query image which are not consented to be used/shared.

For example, in Table 1, constraints N_1 - N_4 essentially collapse to the same (N_2) using the perfect privacy semantics, as they all reveal information (in an information-theoretical sense [36]) about the tuples of Patient '1312'; these tuples are all critical. With our modeling, these are different constraints. Including attributes in the head of constraints denotes that only queries that ask for these (or a superset of these) attributes are violating the constraints. Boolean constraints, e.g, N_2, are the most strict in our framework since they disallow the sharing of any attributes (any attribute set is a superset of the empty set). In that sense, our boolean constraints are the most similar to secret queries. Section 3.5 provides a formal connection of our semantics to perfect privacy using boolean queries.

Offer partial consent-abiding query answers. Perfect privacy is a total "accept/reject" query answering approach. That is, if a secret query and a view share a critical tuple the entire query is deemed insecure and rejected. Instead, we will allow partial query answers: by removing only the non-consented tuples from the result of a query, we still return the most information allowed.

Note that [36] also studies the probability of disclosing a secret. This is not applicable to our setting. Private values in our context are not in fact "secret"; the service provider does have these values but is willing to use them only in the intended way, so the probability of learning a secret is irrelevant.

Faithful to the above requirements, we subsequently define the semantics of consent constraints that allow us to detect consent violations and implement consent-abiding query answering. In principle, consent-abiding query answering is relatively simple: a consent constraint describes values that are preferred not to be returned and so we should remove those tuples from the query answers; there are technical details however that we need to consider.

As a first step, we need to identify the "overlapping" part in the answers of two CQs (one being a consent constraint), and we do this in the next subsection by using data annotations (labelling every data tuple as in [17, 29, 46]). While in Section 4 we develop an algorithm to implement our semantics without relying on annotations, using annotations for our semantics allows for a more tangible interpretation of what our constraints mean since they describe particular tuples; as such, a front-end implementation could even support a visually-aided way of denoting the constraints similar to Figure 1. There is ongoing research [7, 32] as well as real and emerging systems [23, 42] that help transform user policies into CQs. In this paper we focus on the foundational and algorithmic aspects of consent abiding-query answering, rather than the interface.

3.2 Annotated Relations & Overlapping Queries

To pinpoint particular values in our data, we exploit the provenance [29] of a tuple, assuming each tuple in the database annotated with a unique identifier or label (as in [46]), via an *annotation function* λ (see Table 2). We will annotate the answer tuples of a query with the labels by creating a different annotated tuple for every different query homomorphism in the data. Moreover, towards being even more fine-grained, we aim to annotate each individual term in a result tuple with exactly the tuple identifiers it "came from"; this is different to provenance semirings that annotate the

Table 1: User Consent and its mapping to negative consent constraints.

User Consent	Negative Constraint
Patient 1312 does not want to share her phone number (5th attribute of the `Patients` relation)	$N_1(x_5) \leftarrow \texttt{Patients}(1312, x_2, x_3, x_4, x_5, x_6, x_7)$
Patient 1312 does not want to share her any combination of her attributes (N_2). N_2 actually subsumes N_3 and N_4 which dissalow Name and Disease respectively	$N_2() \leftarrow \texttt{Patients}(1312, x_2, x_3, x_4, x_5, x_6, x_7)$ $N_3(x_2) \leftarrow \texttt{Patients}(1312, x_2, x_3, x_4, x_5, x_6, x_7)$ $N_4(x_7) \leftarrow \texttt{Patients}(1312, x_2, x_3, x_4, x_5, x_6, x_7)$
Patients 4872 and 2321 do not want to share the association of their Birthdate together with their Disease	$N_5(x_4, x_7) \leftarrow \texttt{Patients}(4872, x_2, x_3, x_4, x_5, x_6, x_7)$ $N_6(x_4, x_7) \leftarrow \texttt{Patients}(2321, x_2, x_3, x_4, x_5, x_6, x_7)$
Do not share patient IDs when their disease is being cross checked against the `Insurance` table	$N_7(x_1) \leftarrow \texttt{Patients}(x_1, x_2, x_3, x_4, x_5, x_6, x_7) \wedge \texttt{Insurance}(x_7, x_8)$

Table 2: An annotated database I with 4 relations, showing the identifier annotations on the left of each tuple, and example queries/constraints with their annotated answers.

A

a_1 :	1

B

b_1 :	1	1	2	2	3
b_2 :	1	1	1	1	1
b_3 :	1	4	2	2	3
b_4 :	1	4	4	5	6

C

c_1 :	1	5	5	6
c_2 :	1	5	6	6
c_3 :	1	5	5	5

D

d_1 :	1	5
d_2 :	2	7
d_3 :	2	8

$q_1() \leftarrow B(1, 1, 2, 2, 3)$ | $N_0() \leftarrow B(x, y, z, z, v), B(x, x, x, y, y)$

$q_1^\lambda(I) = \{<>^{b_1}\}$ | $N_0^\lambda(I) = \{<>^{b_1, b_2}, <>^{b_2}\}$

$q_2(x_1, 3) \leftarrow A(x_1), B(x_1, x_2, x_3, x_3, 3), D(x_3, x_4)$

$q_2^\lambda(I) = \{< 1^{a_1, b_1}, 3^{b_1} >, < 1^{a_1, b_3}, 3^{b_3} >\}$

$q_3(x_1, x_4) \leftarrow B(x_1, x_2, x_3, x_3, x_4), C(x_1, x_5, x_5, x_6)$

$q_3^\lambda(I) = \{<1^{b_1, c_1}, 3^{b_1}>, <1^{b_2, c_1}, 1^{b_2}>, <1^{b_3, c_1}, 3^{b_3}>, <1^{b_1, c_3}, 3^{b_1}>, <1^{b_2, c_3}, 1^{b_2}>, <1^{b_3, c_3}, 3^{b_3}>\}$

$q_5(x_1) \leftarrow A(x_1), B(x_1, x_2, x_3, x_3, x_4), C(x_1, x_5, x_5, x_6)$

$q_5^\lambda(I) = \{ <1^{a_1, b_1, c_1}>, <1^{a_1, b_2, c_1}>, <1^{a_1, b_3, c_1}>, <1^{a_1, b_1, c_3}>, <1^{a_1, b_2, c_3}>, <1^{a_1, b_3, c_3}> \}$

$q_6(z_1) \leftarrow B(z_1, z_2, z_2, z_3, z_4), C(z_1, z_5, z_6, z_6), D(z_1, z_6)$

$q_6^\lambda(I) = \{< 1^{b_2, c_3} >, < 1^{b_4, c_3} >\}$

entire result tuple (for boolean queries where the result tuple is empty we do annotate it in its entirety).

Consider for example query q_2 and database I in Table 2. This query has the following four different homomorphisms on I:

	homomorphism	annotation
h_1	$\{x_1 \rightarrow 1, x_2 \rightarrow 1, x_3 \rightarrow 2, x_4 \rightarrow 7\}$	$\langle 1^{\{a_1, b_1\}}, 3^{\{b_1\}} \rangle$
h_2	$\{x_1 \rightarrow 1, x_2 \rightarrow 1, x_3 \rightarrow 2, x_4 \rightarrow 8\}$	$\langle 1^{\{a_1, b_1\}}, 3^{\{b_1\}} \rangle$
h_3	$\{x_1 \rightarrow 1, x_2 \rightarrow 4, x_3 \rightarrow 2, x_4 \rightarrow 7\}$	$\langle 1^{\{a_1, b_3\}}, 3^{\{b_3\}} \rangle$
h_4	$\{x_1 \rightarrow 1, x_2 \rightarrow 4, x_3 \rightarrow 2, x_4 \rightarrow 8\}$	$\langle 1^{\{a_1, b_3\}}, 3^{\{b_3\}} \rangle$

For each homomorphism, we annotate a value in the result tuple with the identifiers of the tuples (in the image of the homomorphism) that this value appears in. For q_2, this gives rise to two different annotated answer tuples. Note that different homomorphisms do not always give different annotated tuples since the images of atoms that do not contain head terms (such as D in q_2) do not participate in the annotations of the result tuples.

Note that in provenance semirings [29], we annotate the tuples of a query by constructing a monomial for each homomorphism and "summing up" all monomials to a polynomial. In effect, according to [29] the polynomial annotation for the result $\langle 1, 3 \rangle$ of q_2 would be $a_1 b_1 d_2 + a_1 b_1 d_3 + a_1 b_3 d_2 + a_1 b_3 d_3$. For presentation purposes, and because we want to ignore d_2, d_3 labels that are not projected, we choose to "break" this down to its different monomials.

For boolean queries we will use all image identifiers to label the entire (empty) answer tuple. Shown in Table 2 the boolean query q_1 has a single annotated (empty) answer tuple on I, while constraint N_0 will have two annotated answer tuples: tuple $<>^{b_1, b_2}$ for the homomorphism $\{x, y \rightarrow 1,\ z \rightarrow 2, v \rightarrow 3\}$ that maps the two different atoms of N_0 to b_1 and b_2 respectively, and $<>^{b_2}$ for the homomorphism $\{x, y, z, v \rightarrow 1\}$ that maps both atoms of N to b_2.

Definition 3.2. For all CQs $q(x_1, x_2, ..., x_n)$, instances D, homomorphisms h from q to D, with $h(\langle x_1, x_2, ..., x_n \rangle) = \vec{t} = \langle t_1, t_2, ..., t_n \rangle$, an *annotated answer* tuple of q over D via h, is $\langle t_1^{L_1}, t_2^{L_2}, ..., t_n^{L_n} \rangle$, denoted also $\vec{t}^{\vec{L}}$ with $\vec{L} = \langle L_1, L_2, ..., L_n \rangle$, where each L_i is the set of labels $L_i = \{\lambda(h(\alpha)) |\ \alpha \in atoms(body(q))$ s.t. $x_i \in terms(\alpha)\}$. If q is boolean we annotate an entire (empty) result tuple with $L = \{\lambda(h(\alpha)) \mid \alpha \in atoms(body(q))\}$, and we denote it $\vec{t}^L$.

The *annotated answer* of q over D, denoted $q^\lambda(D)$, is the set of all annotated answer tuples of q over D. Table 2 shows a number of example queries with their annotated answers. We will generally write $\vec{\tau}^L$ for an annotated answer tuple, to either mean that L is a set of identifiers (for boolean queries) or vector of sets (for non-boolean). Given annotated answer tuple $\vec{t} = \vec{\tau}^L$ we define $base(\vec{t}) = \vec{\tau}$. For Γ a set of annotated tuples, $base(\Gamma) = \{base(\vec{t}) | \vec{t} \in \Gamma\}$.

Next, we use annotated tuples to define the overlap between two queries (or a query and a constraint). Intuitively, our semantics will want to eliminate all base tuples from a query answer for which all corresponding annotated answer tuples are violating (common to) some consent constraint; or dually, return those tuples which can be obtained by at least one legitimate way with respect to annotations. Given two annotated answer tuples with the same base, $\vec{t_1} = \vec{\tau}^{L_1}$ and $\vec{t_2} = \vec{\tau}^{L_2}$, their *annotations intersect*, denoted $\vec{t_1} \sqcap \vec{t_2}$, if $L_1 \cap L_2 \neq \emptyset$. When L_1, L_2 are vectors $L_1 = \langle L_{11}, L_{12}, ..., L_{1n} \rangle$ and $L_2 = \langle L_{21}, L_{22}, ..., L_{2n} \rangle$, then $L_1 \cap L_2 \neq \emptyset$ means $L_{1i} \cap L_{2i} \neq \emptyset$ for all $i \in [1, n]$. Two queries are overlapping if they have some annotation intersected tuples *on some database*, per the next definition.

Definition 3.3. For all CQs, q and p of the same result schema, we say that q and p *overlap*, if there exists a database D, a $\vec{t_1} \in q^\lambda(D)$ and a $\vec{t_2} \in p^\lambda(D)$ s.t. $\vec{t_1} \sqcap \vec{t_2}$.

For example, queries q_2 and q_3 from Table 2 overlap on I. Indeed, one annotated answer tuple of q_2, obtained by homomorphism h_2, is $\vec{t}_2^{\vec{L}_2} = \langle 1^{a_1,b_1}, 3^{b_1} \rangle$ and one of q_3 is $\vec{t}_3^{\vec{L}_3} = \langle 1^{b_1,c_1}, 3^{b_1} \rangle$ (let h_{q_3} the homomorphism constructing this tuple). The fact that $\vec{L_2} \cap \vec{L_3} \neq \emptyset$ (thus, $\vec{t}_2 \sqcap \vec{t}_3$) means that each result value in $\vec{t}_2^{\vec{L}_2}$ and $\vec{t}_2^{\vec{L}_3}$ exists in the same actual tuple (here b_1), common to the image of h_2 and h_{q_3}. In fact queries q_2 and q_3 overlap for both annotated answer tuples of q_2; if q_2 was an input query and q_3 a negative consent constraint we would have to remove all annotated answer tuples of q_2 on this particular database in order to execute it in a consent abiding way.

Before we fully define consent-abiding query answering, we further enhance our semantics to capture cases where a query and a constraint differ in their answer relation schemas but we would still like to consider them overlapping.

3.3 Query-consent Overlap and Violation

Our definition of overlapping queries relies on annotated answer tuples having the same base and schema. However, we might want to consider a query and a constraint overlapping even if the answer relations (the heads of the rules) have trivially different schemas.

First, notice that a resulting query relation might have a different order on its attributes than the constraint answer relation, but we might still want to consider these overlapping. Consider for example, $q_7(x_7, x_2) \leftarrow \texttt{Patients}(x_1, x_2, x_3, x_4, x_5, x_6, x_7)$ which asks for $\langle Disease, Birthdate \rangle$ pairs of patients. Constraints N_5 and N_6 of Table 1 disapprove the sharing of $\langle Birthdate, Disease \rangle$ pairs for two particular patients. Even though the head attributes are the same, their order is different and hence a base tuple in the result of q_7 can not be the same as one returned by constraints N_5 or N_6. Nevertheless, our natural interpretation of N_5 and N_6 is that users prefer not to associate birthdates with diseases, independently of the actual order of appearance of these values in an answer tuple.

Second, equally important is to abide to user consent in the face of a query that asks for "more" attributes than those corresponding to the constraint head. For example, for query $q_8(x_7, x_1, x_2, x_3) \leftarrow \texttt{Patients}(x_1, x_2, x_3, x_4, x_5, x_6, x_7)$ we believe it is natural to remove tuples with non-consented values for *Birthdate* and *Disease* in the attributes for x_2 and x_7 (per constraints N_5 and N_6); even if these tuples are "larger" and contain more values, they still violate intended consent. The requirements of reordering or dropping attributes of the query can be supported by the projection operator; essentially we want to look for overlaps between any projection $\pi_A(q(\vec{x}))$ of a query q and a constraint N.

Our last observation for establishing our consent semantics is the following. Consider constraint N_1 from Table 1 and a query asking for the IDs of patients whose phone number is 23293456 (that happens to be the phone number of patient 1312): $q_9(x_1) \leftarrow \texttt{Patients}(x_1, x_2, x_3, x_4, 23293456, x_6, x_7)$. With our definitions up to now, q_9 and N_1 cannot be overlapping since they do not even project on the same attribute of `Patients`. However, patient 1312, in Figure 1, does not want her phone number revealed, and if we return her ID in the answer of q_9 we are violating her consent. The observation here is that although the query does not explicitly ask for the *PhoneNo* attribute through a head variable, it grounds this attribute to a constant and in a sense it is still "asking" for it. To address this, we place all constants appearing in the query, in its head as well. Given a conjunctive query $q(\vec{x})$, with $\vec{x} = \langle x_1, x_2, ..., x_n \rangle$, let $\langle c_1, c_2, ..., c_k \rangle$ be the tuple of all constants $c_i \in (consts(body(q)) \setminus \vec{x})$ in a lexicographic order (although the order does not matter). By $\hat{q}$ we denote the *constant-extended version* of q, that is, the conjunctive query with same body as q, and head $\hat{q}(x_1, x_2, ..., x_n, c_1, c_2, ..., c_k)$.

To take all three observations into account, given a query $q(\vec{x})$ we define its *support set* of queries $\hat{\Pi}_q$, as the set of queries which are projections on $\hat{q}(\vec{x})$, i.e., $\hat{\Pi}_q = \{\pi_A(\hat{q}(\vec{x})) \mid A$ is a non-empty list of integers (positions) $\langle i_1, ..., i_m \rangle$, with $i_i \in [1, |\vec{x}|]\}$. In practice, we can limit m to the maximum arity of any constraint head (there is no use to create support queries with heads larger than those of the constraints since these will necessarily have different schemas).

Definition 3.4. For all CQs q, for all CQ consent constraints q_N, we say that q *violates* q_N, if there is a support query $q_\pi \in \hat{\Pi}_q$ such that q_π and q_N overlap.

As an example, query q_9 with the user's phone number violates N_1 from Table 1, since the support query $\pi_1(\hat{q}_9(x_1, 23293456)) \leftarrow \texttt{Patients}(x_1,x_2,x_3,x_4, 23293456,x_6,x_7)$ is contained in N_1, and thus trivially $\pi_1(\hat{q}_9)$ and N_1 overlap. Note that support query projections are performed only on the input query and not the constraint. Per the semantics of Table 1, N_5 is not violated by a query asking for only one of the two attributes in its head, rather N_5 is only violated by queries asking (or setting) at least both attributes. Note that support queries are not necessary for boolean constraints, since just the boolean version of the query is enough to detect violations.

3.4 Consent-abiding Query Answering

Once we detect a violation of a support query q_π and a constraint q_N, we should remove from the query those annotated answer tuples that intersect with the answer of q_N, in order to answer the query in a consent abiding way. To compare an annotated tuple $\vec{t}$ from the query to a tuple in the answer of q_N, we need to use the projection operator on the single tuple $\vec{t}$ as well, in order to reorder or drop some of its values, as defined in Section 2. When we use projection on an annotated answer tuple, the annotations of projected terms are maintained.

Definition 3.5. Given a CQ q, a set of CQ constraints $\mathcal{N}$, and a database D, the annotated consent-abiding answer of q w.r.t. $\mathcal{N}$ is $q \setminus_D \mathcal{N} = \{\vec{t} \mid \vec{t} \in q^\lambda(D)$ and there is no support query $q_\pi = \pi_A(\hat{q}) \in \Pi_q$, constraint $q_N \in \mathcal{N}$ and $\vec{t_N} \in q_N^\lambda(D)$ such that $\pi_A(\vec{t}) \sqcap \vec{t_N}\}$.

After removing all consent-violating annotated tuples we drop annotations to get the final answer tuples of our query. Thus, intuitively, base tuples make it to the consent-abiding answer if they can be obtained by at least one non-consent-violating way. The *consent-abiding query answer* of q with respect to $\mathcal{N}$ over D, denoted by $q^\neg(\mathcal{N}, D)$, is $q^\neg(\mathcal{N}, D) = base(q \setminus_D \mathcal{N})$.

Towards a practical implementation of our semantics note that if a query and a constraint overlap they do so on all databases where both queries are answerable and their homomorphic images "intersect" on some database atoms. In fact, two query atoms that map to the same database tuple are unifiable (see section 2), and

by computing unifications between query and constraint atoms we will present a rewriting algorithm that produces a query that can be directly executed using SQL and returns $q^{\neg}(\mathcal{N}, D)$. Intuitively, unifications can give us the intersections of the queries' annotated tuples on a "schema level". This has multiple benefits: It allows us to develop a data-independent approach; our consent-abiding rewriting is the same for all database instances. At the same time we do not need to annotate the data nor do a brute-force comparison of the annotation vectors for all pairs of resulting tuples; unified parts of the query and the constraint will directly map to intersected tuples. Moreover, by bookkeeping the way constraint head variables unify with query terms, in the next section, we automatically obtain all support queries as results of these unifications.

3.5 Privacy guarantees

As discussed, our framework has been designed with collaborative privacy in mind, and not to protect against an adversary. Nevertheless we provide a formal connection to traditional data privacy and in particular perfect privacy [36]. The following Theorem guarantees that using our framework with boolean constraints, a query that does not violate a constraint per our definition means always that the constraint is secure with respect to the query per [36] (all theorem and lemma proofs can be found in our online appendix at https://github.com/georgeKon/enabling-personal-consent).

Theorem 3.6. *For all CQs q and boolean CQs q_N, q if q does not violate q_N then $q_N|q$.*

Note that our boolean constraint semantics is more strict than secret queries; there are corner cases where we might call a violation where perfect privacy is not violated. Due to space limitation we do not give an example but point the interested reader to [36], page 7, example immediately following Prop 3.9. Admittedly as the authors of [36] point out this concerns some rather unnatural cases, and in practice and most cases our semantics actually coincides (certainly for all examples of this paper and of [36]).

4 A PRACTICAL SOLUTION

Our objective in this section is to produce a rewriting of the query and the constraints that can be directly executed to obtain the consent-abiding answer on any (non-annotated) database. If we attempt to do this by simply taking the difference of a constraint q_N from a query q we might remove tuples which are not annotation-intersected. Thus, intuitively, we aim to produce a new constraint q_m, by unifying q and q_N appropriately, such that we can safely remove all answers of q_m; due to the way q_m is constructed all its homomorphisms will be "touching" tuples that the query does.

4.1 Query Unifications and SQL Rewritings

In order to unify a set of query atoms to atoms of another query or constraint, we define a variant of unification for sets of atoms (reminiscent of piece unification [8]). Our unifiers describe all atoms of a query that are unified with some atoms in the constraint.

Definition 4.1. For non-empty sets of atoms $S_1 = \{a_1, a_2, ..., a_m\}$ and $S_2 = \{b_1, b_2, ..., b_n\}$, a piece unifier of S_1 over S_2 is a substitution $\theta^{S_1}_{S_2}$ s.t. for all $a_i \in S_1$ there is a $b_j \in S_2$ with $\theta^{S_1}_{S_2}(a_i) = \theta^{S_1}_{S_2}(b_j)$; we call all such (a_i, b_j) pairs *the unifiable pairs* of $\theta^{S_1}_{S_2}$.

For example let $S_1 = \{B(x_1, x_2, x_3, x_3, x_4), C(x_1, x_5, x_5, x_6)\}$ and $S_2 = \{B(z_1, z_2, z_2, z_3, z_4), C(z_1, z_5, z_6, z_6), D(z_1, z_6)\}$. Notice that S_1 is a subset of the body of q_5 from Table 2 and S_2 is the body of q_6. One piece unifier is ${\theta_1}^{S_1}_{S_2} = \{ x_1 \to z_1, x_2 \to z_2, x_3 \to z_2, x_4 \to z_4, x_5 \to z_5, x_6 \to z_5,\ z_3 \to z_2,\ z_6 \to z_5 \}$. Applying the unification, ${\theta_1}^{S_1}_{S_2}(S_2) = \{B(z_1, z_2, z_2, z_2, z_4), C(z_1, z_5, z_5, z_5),\ D(z_1, z_5)\}$, and indeed for all a_i in S_1, $\theta^{S_1}_{S_2}(a_i)$ is in this set. Note that this unified piece has a homomorphism to the annotated tuples in Table 2 that are common to q_5 and q_6 (where q_6 can be negative constraint). To construct a proper unfied CQ out of this unifier we should also unify the heads of the the query and the constraint.

Definition 4.2. For all CQs $q(\vec{x})$, $q_N(\vec{z})$, and non-empty $S \subseteq body(q)$, a query unifier of q over q_N for S, is a piece unifier of S over $body(q_N)$, $\theta^S_{body(q_N)}$, such that $\theta^S_{body(q_N)}(\vec{x}) = \theta^S_{body(q_N)}(\vec{z})$.

Given a query unifier θ of $q(\vec{x})$ over $q_N(\vec{z})$, the *query unification* for θ, denoted $\theta[q, q_N]$, is the conjunctive query with head $q(\theta(\vec{x}))$ and body $\theta(body(q)) \cup \theta(body(q_N))$. Using ${\theta_1}^{S_1}_{S_2}$ from previously as θ, the query unification $\theta[q_5, q_6]$, of q_5 over q_6 is:
$q_{BC}(z_1) \leftarrow A(z_1), B(z_1, z_2, z_2, z_2, z_4), C(z_1, z_5, z_5, z_5), D(z_1, z_5)$. There are other query unifiers for q_5 over $q6$, that could potentially use a different subset of $body(q_5)$, as long as these still unify the distinguished variables of both queries, e.g., for $S_3 = \{B(x_1, x_2, x_3, x_3, x_4)\}$ or $S_4 = \{C(x_1, x_5, x_5, x_6)\}$. For example ${\theta_2}^{S_3}_{body(q_6)} = \{x_1 \to z_1, x_2 \to z_2, x_3 \to z_2, x_4 \to z_4, z_3 \to z_2\}$ unfies only the B atoms and keeps both C atoms of the queries, producing query unification $q_B(z_1) \leftarrow A(z_1), B(z_1, z_2, z_2, z_2, z_4), C(z_1, x_5, x_5, x_6), C(z_1, z_5, z_6, z_6), D(z_1, z_6)$, while unifier ${\theta_3}^{S_4}_{body(q_6)} = \{x_1 \to z_1, x_5 \to z_5, x_6 \to z_5, z_6 \to z_5\}$ unfies only the C atoms producing $q_C(z_1) \leftarrow A(z_1), B(z_1, x_2, x_3, x_3, x_4), B(z_1, z_2, z_2, z_3, z_4), C(z_1, z_5, z_5, z_5), D(z_1, z_5)$.

Intuitively, all query unifications provide only consent-violating annotated answer tuples. For every distinguished variable of the constraint, a query unification contains a unified atom with this variable, and anywhere this atom maps on a database the annotation labels will be "propagated" to the constrtaint result value. At the same time, the query also maps to this atom by mapping one of its own head terms and thus inheriting the same database annotations on this term. For example, the annotated answer tuples that q_B gives would contain a superset of the annotations that q_5 gets on the resulting value for z_1 (since q_B contains more atoms). Similarly, q_C will propagate the annotations, for any tuple of table C it touches, to both q_5 and q_6. Lastly, note that q_{BC} might only give a subset of the annotated answer tuples that either q_B or q_C give as it is more restrictive, essentially contained in both q_B and q_C. In this example q_B and q_C are enough to capture (intersect with) all intersected answer tuples of q_5 with Q_6. Indeed, we will not need all query unifications but just the most general ones, i.e., the ones that come out of a minimal set of unifications between the query and constraint (which still unify all constraint head terms).

Definition 4.3. For all CQs $q(\vec{x})$, $q_N(\vec{z})$, and non-empty $S \subseteq body(q)$, a most general query unifier of q over q_N for S, is a query unifier $\theta^S_{body(q_N)}$, s.t. there is no non-empty $S' \subset S$ with a query unifier $\theta^{S'}_{body(q_N)}$ for which $\theta^S_{body(q_N)}(\vec{x}) = \theta^{S'}_{body(q_N)}(\vec{x})$.

Let $MGQU(q, q')$ the set of all most general query unifications of q over q'. We can now implement the annotated consent abiding answer of a query and a set of constraints, as in Def. 3.5, by considering queries in $MGQU$ rather than the constraints.

Theorem 4.4. *Given a CQ q, set of constraints $\mathcal{N}$, and database D, $q \setminus_D \mathcal{N} = \{\vec{t} \mid \vec{t} \in q^{\lambda}(D)$ and there is no support query $q_{\pi} = \pi_A(\hat{q}) \in \hat{\Pi}_q$, constraint $q_N \in \mathcal{N}$, $q_m \in MGQU(q_{\pi}, q_N)$ and $\vec{t_m} \in q_m^{\lambda}(D)$ such that $\pi_A(\vec{t}) \sqcap \vec{t_m}\}$.*

Given a support query and a constraint, Theorem 4.4 dictates that it is only tuples of a query unification that need to be removed. In practice, as explained in Section 3.4 (and as we will do in Section 4.2), by relaxing the requirement to unify all query head terms we essentially obtain directly the corresponding projections of a support query; it has the head constructed from those query terms that unified with the constraint distinguished variables. In the examples above, even if q_5 had additional free variables to z_1, it is only z_1 that is needed to violate q_6 (as a constraint). Thus, every different appropriate unification of the query and the constraint produces a different q_m in Theorem 4.4, and for every q_m we will create a query rewriting; the union of these rewritings gives the consent-abiding query answer, without relying on annotations, by removing those tuples that exactly "satisfy" unifications q_m. In the next subsection, we present a bottom up, dynamic-programming style of algorithm to obtain an mgu.

To see how we construct our rewritings, consider query $q(x) \leftarrow A(x, y)$ and a constraint $q_N(z) \leftarrow A(z, z)$; the most general query unification is $q'(x) \leftarrow A(x, x)$, equating x and y in q (support query is q itself). Let a database with two tuples $D = \{A(1, 0), A(1, 1)\}$. If we were, however, to execute $q(D) \setminus q'(D)$ the result would be empty since the value 1 coming from tuple $A(1, 0)$ would also get removed, but per our semantics it should be retained. To achieve this without annotations we have to execute a query informally looking like $q(x) \leftarrow [A(x, y) \setminus (A(x, x), x = y)]$; by this we mean to iterate over every tuple of $A(x, y)$ and remove those which agree with $(A(x, x), x = y)$, i.e., agree on all unified attributes, pinpointing exactly the tuples to remove. We can express such queries in SQL.

Lemma 1. *Let a CQ $q(\vec{x}) \leftarrow P_1(\vec{x_1}), ..., P_n(\vec{x_n})$, a CQ constraint q_N, a support query $q_{\pi}(\vec{y}) = \pi_A(\hat{q})$, and a $q_m \in MGQU(q_{\pi}, q_N)$ with $q_m(\vec{z}) \leftarrow R_1(\vec{z_1}), ..., R_m(\vec{z_m})$. The following SQL query returns all tuples $\vec{t} \in q(D)$ for which there are no $\vec{t_q} \in q^{\lambda}(D)$ with $base(\vec{t_q}) = \vec{t}$, and $\vec{t_m} \in q_m^{\lambda}(D)$, such that $\pi_A(\vec{t_q}) \sqcap \vec{t_m}$:*

```
SELECT x⃗ FROM P_1, ..., P_n
WHERE J_q AND NOT EXISTS(
        SELECT z⃗ FROM R_1, ..., R_m
        WHERE J_m AND J_A AND J_θ);
```

where

- *J_q is a conjunction of equalities of the form $P_i.x = P_l.x$ for all the joins/repeated terms in atoms P_i, P_j in q,*
- *J_m is the corresponding conjunction of equalities, $R_i.z = R_j.z$, for the joins/repeated terms in q_m,*
- *J_A is the conjunction of equalities $\bigwedge_{i=1}^{i=|\vec{y}|} \vec{y}[i] = \vec{z}[i]$ (equating the projection/re-ordering of terms in $\pi_A(\hat{q})$ and q_m),*
- *J_{θ} adds the unification equalities between q_{π} and q_m, i.e., for $\theta[q_{\pi}, q_N] = q_m$ and for all unifiable pairs of θ, $(\alpha, \delta) \in body(q) \times body(q_N)$, for all terms $u \in terms(\alpha)$ we add $u = \theta(u)$ in J_{θ}.*

As an example consider again q_5 against constraint q_6 and their most general query unification q_B (using q_5 itself as support query). To remove consent violating tuples of q_B from q_5 we execute:

```
SELECT A.at1 FROM A, B, C
WHERE A.at1=B.at1=C.at1 AND B.at3=B.at4 AND C.at2=C.at3
AND NOT EXISTS(
SELECT A2.at1 FROM A as A2,B as B2,C as C2,C as C3,D
WHERE  A2.at1=B2.at1=C2.at1=C3.at1=D.at1  AND  B2.at2 =
B2.at3 = B2.at4 AND C2.at2=C.2at3 AND C3.at3=C3.at4=D.at2
AND B.at1=B2.at1 AND B.at2=B.at3=B2.at2 AND B.at5=B2.at5)
```

In the face of multiple constraints and MGQUs, Lemma 1 generalizes to construct $q^{\neg}(\mathcal{N}, D)$ by simply conjuncting more nested `NOT EXISTS` queries to the `WHERE` clause of the outer query. Alternatively, we could `UNION` more nested queries inside the `NOT EXISTS`. Notice that there is not a unique way to express difference with union in SQL; in fact, in Section 5.2 we run different comparisons with EXCEPT, LEFT OUTER JOIN and NOT EXISTS. Through experimentation, we decide to use NOT EXISTS; however, finding the optimal serialization is an orthogonal rich research problem [43]. Independently of the actual serialization of Lemma 1 in the face of multiple constraints the query produced is the *consent-abiding rewriting* of q with respect to $\mathcal{N}$, and we can represent it as a union of CQs (UCQ) with safe negation.

4.2 Algorithm

In this section, we present the core algorithm of our framework to find the most general query unifications of support queries. This happens by unifying the original query's body with the constraint such that all constraint distinguished variables are unified with either distinguished query variables or query constants (thus identifying the necessary head terms of a support query). The output of this algorithm is a set of query unifications q_m each paired with the necessary equalities J_A and J_{θ} as required by Lemma 1; these pairs are then translated to SQL as in the previous subsection.

Generally, given an input query q and a constraint q_N we will start by pairwise unifying, initially taking the mgu (see Sect. 2) of single atoms in the query and the constraint, while making sure that we unify head terms in the constraint with either a head term of the query or a constant. By starting from pairwise unifications of single atoms, we guarantee that our unifiers will be most general, i.e., contain a minimal number of query atoms (or unifiable pairs). If by simple pairwise unifications we cover all head terms of the constraint (if the constraint is boolean, one atomic unifier is enough) we have essentially managed to find a most general query unifier over a support query of q. If by using pairwise mgu's we fail to unify all head terms of the constraint it might be because two, or more, query atoms need to be unified with a constraint atom. To unify larger sets of atoms in a minimal and efficient way, if a set of atoms has been unified and has not covered all the constraint head terms, this unifier is still "open" and we will try to further unify it with another query atom. Whenever some sets of atoms unify successfully we shall not use them in further unifications, as unnecessary unifications will spoil the most-generality property.

The structure that keeps our substitutions/unifications is a set of *equivalences classes* named `ECset` in our pseudocode. Each equivalence class in this set contains terms that have been unified with

one of them being a representative term, so each equivalence class essentially is a substitution that maps all the terms it contains to the representative (obviously if there is constant in an equivalence class this should be the representative). Substitutions that are used together in a unifier are kept in a set of `ECset` objects.

Algorithm 1: findMGQUs

Input: A Query q and a set of constraints $\mathcal{N}$
Output: Union of pairs $\langle MGQU(q_\pi, q_N), J_A, J_\theta \rangle$ where $q_\pi, q_N, J_A, J_\theta$ are as in Lemma 1

1: `Result` $\leftarrow \emptyset$
2: **for all** constraints $q_N \in \mathcal{N}$ **do**
3: `ApplicableECsets` $\leftarrow \emptyset$
4: `OpenAtomicECsets` $\leftarrow \emptyset$
5: **for all** atoms α in q **do**
6: **for all** atoms δ in q_N **do**
7: `ECset` $\leftarrow$ `unify`(α, δ)
8: **if** `unify`(α, δ) was not successful **then**
9: **continue** line 6
10: **if** `coversHead`(`ECset`, q_N) = `true` **then**
11: `ApplicableECsets` $\leftarrow$ `ApplicableECsets` $\cup$ `ECset`
12: **else**
13: `OpenAtomicECsets` $\leftarrow$ `OpenAtomicECsets` $\cup$ `ECset`
14: $n \leftarrow 2$ //first pick pairs, then triples, etc
15: **while** $n <$ |`OpenAtomicECsets`| **do**
16: **for all** sets `T` = {`ECset`$_1$, ..., `ECset`$_n$} where `ECset`$_i \in$ `OpenAtomicECsets` and for $i \neq j$, `ECset`$_i \neq$ `ECset`$_j$ **do**
17: **if** no subset of T has been marked applicable **then**
18: `ECset` $\leftarrow \emptyset$
19: **for** $i \in \{1, ..., n\}$ **do**
20: `ECset` $\leftarrow$ `unionECSets`(`ECset`,`ECset`$_i$)
21: **if** union unsuccessful **then**
22: **goto** line 16
23: **if** `coversHead`(`ECset`,q_N) **then**
24: add `ECset` to `ApplicableECsets`
25: mark `T` as applicable
26: $n \leftarrow n + 1$
27: **for all** `ECset` $\in$ `ApplicableECsets` **do**
28: `mgqu-query` $\leftarrow$ `apply`(`ECset`, q_N) //replace terms with EC representative
29: `Equalities` $\leftarrow \emptyset$
30: **for all** `EC` $\in$ `ECset` **do**
31: `Equalities` $\leftarrow$ `Equalities` $\cup$ transitive closure of `EC`
32: `Result` $\leftarrow$ `Result` $\cup$ $\langle$`mgqu-query`, `Equalites`$\rangle$
33: **return** `Result`

Algorithm 1 starts by initializing the applicable unifiers, i.e., those that are already most general, and the open ones, i.e., those that should be combined further (lines 3-4). Then, it tries to unify all pairs of query-constraint atoms (lines 5-13), and for the successful unifications it checks whether each one of them is a most general query unifier, i.e., whether it has covered all head variables of the constraint (line 10); those are called applicable and stored aside for later generation of the most general query unifications. Those that are not applicable are still open (line 13). Note that, each one of the applicable unifiers corresponds to a different query unification of a support query over the constraint, since most probably it is using different head terms and constants of the original query. The pairwise unifications that are still open are combined gradually in larger groups to each other (lines 14-25). Note that, to make sure we end up with a most general unifier, we do not try combinations of $n+1$ open unifiers before we finish examining combinations of n. We first chose two ECSets (line 14) to see if they are compatible and they can be unioned (line 20). This unioning essentially consists of comparing all equivalence classes in the two sets and merging them if they contain common values; or fail if they contain common values but their representatives are different constants. At the end of this process we apply the most general query unifiers to get their query unifications, and we compute equalities between terms that we will use as J_A and J_θ (lines 27-32).

In practice, our implementation is more careful than the abstract algorithm presented here, e.g., in line 31 our implementation computes only a required part of the transitive closure. Moreover, notice that in line 28 we obtain the most general query unification by only applying the unifier to the constraint. We do not have to include the extra joined atoms of the input query apart from the unified atoms. Our end objective is to remove tuples from the input query and, since the input query contains these joins, any tuples that we are going to remove have to satisfy such joins in the first place. This is referred to as *serialization optimization* in our experiments.

Every unification Alg. 1 finds creates an element in the union of our rewriting. The latter can be exponentially large but each element is NP-complete to compute in constraint size, as we prove below. Moreover, to evaluate each of the elements in the union has the same complexity as evaluation of CQs with safe negation (in NP in query size). Thus, the whole end-to-end problem remains NP-complete in combined complexity. Our complexity results are reminiscent of Ontology Based Data Access [19], where a CQ is rewritten over ontological constraints into a (possibly exponential) UCQ. Similar to these results and since our rewriting is a UCQ with safe negation we know that our data complexity (fixing the query and constraints) is in AC0.

Theorem 4.5. *Given a CQ q, a constraint q_N, a database D, and a tuple t, deciding if $t \in q^{\neg}(\{q_N\}, D)$ is NP-complete.*

5 EVALUATION

The algorithms described in Section 4, were implemented in Java and executed over a PostgreSQL database using default settings (code, data and experiments can be found in [1]). All experiments were run on a 2.30GHz processor, with 32GB of memory, and a total of 512GB of disk space. We have used two datasets for our experiments. Initially, we used the TPC-H [47] dataset; in addition to being a widely used benchmark in databases, it also provides realistic scenarios for our framework such as customers and their personal order data. We scaled TPC-H with default distribution using scale factors 0.1, 1, 7, 33, 66, and 100. For reference, this results in 15K, 150K, 1 million, 5 million, 10 million and 15 million tuples in the Customer relation alone. We manually extracted the conjunctive part of the TPC-H queries (queries 14 and 22 are meaningless as CQs and discarded). Our second, real, dataset is MIMIC-III [3], a large, freely-available database comprising of de-identified health-related data associated with over forty thousand patients who stayed in critical care units of the Beth Israel Deaconess Medical Center

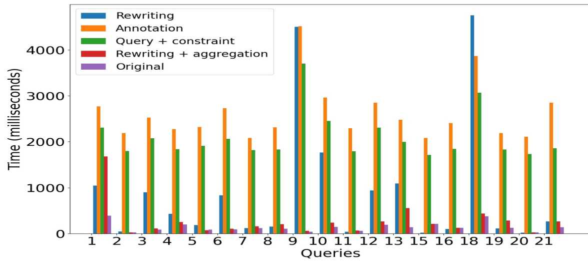

Figure 2: TPC-H queries against constraints that opt-out 10% of the join values. The graph shows our rewriting- vs the annotation-based approach, the original query times also with the execution of all constraints, and the re-application of aggregation operators after the rewriting process.

between 2001 and 2012. We hand-crafted a number of queries and automatically generated constraints inspired by the policies of [48].

We have four sets of main experiments. First, we compare against our annotation-based semantics and we also show the impact of re-application of aggregations on the rewritten queries. Second, we compare against the most relative opt-in/out approach from data privacy. Third, we explore the overhead of supporting fine-grained consent in query answering (which often is minor), and then we do scalability experiments where we scale the number and size of constraints, using both TPC-H and MIMIC-III datasets.

5.1 Annotation semantics and aggregation

First, we compare our rewriting-based approach with our implementation of the annotation-based semantics. To implement this semantics we wrote scripts that extend the database schema with a label column and the data tuples with a unique id. We then implemented Def. 3.5 to obtain the consent abiding answer of the query. We run several experiments with the TPC-H queries, using the OJ set of constraints explained in Section 5.3. We varied the data size and the number of constraints and we compared the rewriting-based approach with the annotation-based. Fig. 2 shows a small TPC-H scale (0.1) using 500 constraints.

The rewriting-based semantics clearly performs faster, even by an order of magnitude, in all but two queries. Nevertheless, we find that in larger scales the difference between the two approaches becomes smaller and further investigation is needed into settings where the annotation-based implementation might be preferable. At the same time, the figure shows the original query execution's time, and as well the sum of the times it takes to execute all constraints plus the query in isolation. We see that executing the rewriting in all but the two slow aforementioned rewritings (Q9, Q18) is faster than just executing the query and the constraints in isolation. This, intuitively, shows that our rewriting approach very often does not induce more overhead than what the constraints encode; on the contrary, it might actually factor out execution cost of constraints that are not overlapping with the query. The last point that Figure 2 makes is about inserting aggregation back in our re-writings. As discussed, we rewrite the CQ part of our original query; thereafter we have implemented a re-insertion of the aggregation back into the rewriting. Note that our scripts do a "best-effort" approach in re-inserting the aggregations; many of the TPC-H queries contain aggregations in nested queries or interacting with other features which are stripped out of the CQ version. As such, re-inserting aggregations might end up in queries with different semantics. In fact, we believe that focusing on aggregations should be an independent piece of feature research on its own. Our experiments do show however the impact of aggregations to some extend and we see that, in almost half the queries the rewriting plus the aggregation is faster than the rewriting itself. This happens in particular when the rewriting produces an output which is order of magnitudes larger than the result set of the aggregated version. Note that at this point aggregation is supported only on the input queries and not the negative constraints; for the latter further research is needed.

5.2 Comparison to Opt-in/out Approaches

Our approach allows us to capture more traditional opt-in/out approaches as well. Here, we compare against the Hippocratic work [33] where a user opts-in/out of sharing an attribute of a table.

Experiments in [33] use opt-in/-out choices at varying degrees of selectivity over a single table. These options are encoded either within the data as extra columns in the same data table (-IN) or externally in a separate table (-EX). In our approach we can easily encode these options as negative constraints. We reimplemented this work (HIPPO) using the Customer table of the TPC-H dataset which has 8 attributes. For the solution HIPPO-EX, we create a "Choice Table" with 8 attributes distributing values that can take 1 (opt-in) or 0 (opt-out) as follows. Over the 8 attributes for which a user can express choices, we vary the privacy selectivity to encompass: 1%, 10%, 50%, 90%, 100% opt-ins. Within [33], a basic query selecting all attributes is executed; based on the consent stored in the choice cells, data cells are nulled out, and if the key becomes null the entire tuple is removed from the answer. In our recreation of HIPPO, we execute the same "`SELECT *`" query over Customer with either internally or externally represented choices. Additionally, we create a set of negative constraints that reflect information from the choices. These are serialized via EXCEPT (E), LEFT OUTER JOIN (LOJ), NOT EXISTS (NE), as discussed in Section 4.1. For these use the optimization serialization discussed in Sec. 4.2, but we also create a version of NOT EXISTS that does not use this optimization (NEU). We also run the unmodified query (UM).

Queries are evaluated on a warmed-up database, and we take the average of 11 executions. Fig. 3 contains times for all these queries that honor consent both via HIPPO and our approach, for the TPCH scale 100. As expected, enforcing the privacy choices is slower than the unmodified query (UM) for all implementations. Our implementation of NOT EXISTS with internal choices (NE-IN) outperforms the unoptimized NOT EXISTS with internal choices (NEU-IN), indicating that the optimization we discuss at the end of Sec. 4.2 is worthwhile. Our NOT EXIST external choices (NE-EX) outperforms both LEFT OUTER JOIN external choices (LOJ-EX) and EXCEPT external choices (E-EX). Based on this we use NOT EXISTS with optimization as our serialization method for the rest of our experiments. Moreover, we can focus on comparing HIPPO-IN against our NE-IN, and HIPPO-EX against our NE-EX. Our NE-IN is always on par or slightly faster than HIPPO-IN, while our NE-EX

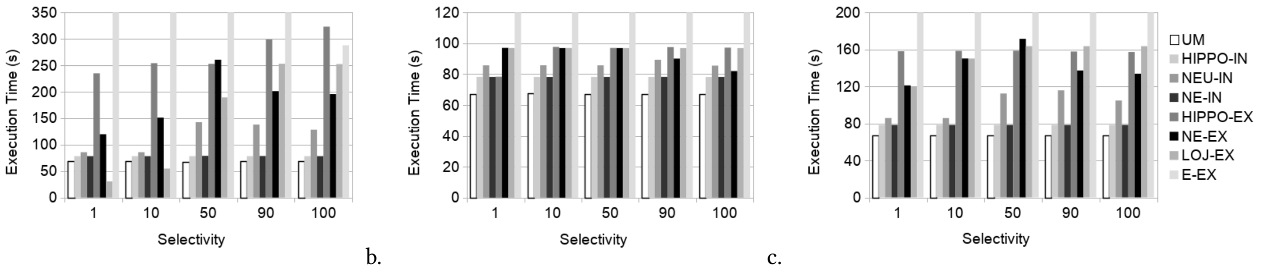

Figure 3: Comparing the methods used in the Hippocratic Databases and different possible serialization methods for consent abiding queries over TPC-H with a scale factor of 100 (15 million rows), with constraints retaining 1%, 10%, 50%, 90% and 100% of the unmodified query results. a. AllChoices contains 8 projected attributes in the Query and 8 in the constraint head; b. KeyChoices contains 8 projected attributes in the Query and just the single customer key in the constraint head; c. SomeChoices projects on 4 non-key attributes in the Query and has the same attribute in the constraint head. implementations are shown for each choice configuration in the same order as shown top to bottom in the legend for ease of comparison.

is faster than HIPPO-EX in 66% of the cases and mostly on par in the rest. Overall, our performance is comparable or faster than the hippocratic work, while being substantially more expressive.

A particular kind of additional useful expressivity that [33] does not support has to do with joins. As a conventional data privacy approach, [33] is more strict and operates by “nulling-out” attributes or entire tuples that are opted-out as soon as it scans a database table. Thus, the Hippocratic approach will not be able to answer query $q(x) \longleftarrow A(x, y), B(y, z)$ if y is private - even though it is not returned. We offer a more expressive language, where one could choose to disallow the sharing of join. In the query above we would simply return all values for x from tuples that do not join with B. Simple opt-in/opt-out choices can be captured by atomic constraints and joins can be seen as a direct extension of these. E.g., the query q(tel) ⟵ Nurse(tel, name, building), NightShift(building, “17/04/21”) will return all telephone numbers (as long as they are not private) of nurses in buildings that have a nightshift on a specific date, even though all other attributes of the query could be private.

To evaluate this we created a set of constraints on attributes which are joined in the TPC-H queries. For each query we randomly chose a different joined attribute and made it private (set it to opt-out) in approximately 10% of its values in the data. We run all queries with joins (so we left out 1, 4, 6 and 15) on a database with scale 1 and we measured that our approach gives from 10% to over 40% more answer tuples in this setting (graph omitted due to space).

5.3 Complex Constraints

In order to investigate the impact of complex constraints on performance, we have generated a set of four constraints for each query in the TPC-H dataset. Constraint 1 and 2 for a given query are built by starting with the initial query and adding additional atoms. The addition of these atoms is designed to remove ~50% and ~ 5% of the unmodified query’s tuples respectively. For instance, Query 10’s Constraint 1 has Lineitem’s L_linestatus attribute fixed to a value of ’F’, which is the value that 50% of the rows in Lineitem take; thus it will remove half of the query results. Constraint 2 for every query is designed to remove 4% of possible tuples. Constraints 3 and 4 are built by starting with Constraint 1 and 2 and removing atoms (such as Nation) that do not essentially remove query tuples (all queries and constraints can be found on our Github page [1]).

Fig. 4 shows the time to rewrite a query to be consent abiding, grouped by constraint type (e.g. Constraints 1-4). Every constraint was applied individually to its related query. The results have been averaged, per constraint type for a given number of atoms. All rewritings are within 3 milliseconds. The rewriting time appears to grow exponentially with the number of atoms in the constraint (as the problem is NP-complete), but it is mostly insignificant compared to the actual query execution time as shown in Fig. 5, indicating that consent enforcement does not induce a large overhead.

Figure 5 compares the slowdown of query execution, as calculated by {consent-abiding query execution time / unmodified query execution time}, versus the percentage of query results returned as calculated by { consent abiding # rows / unmodified query # rows}. Four query-constraint pairs (q_{id},C_{id}) are not shown in Figure 5: (20,3) (11,3), (15,4), (20,4) with slowdown multipliers of 18, 32, 141 and 642 respectively. For (20,3) and (20,4), the unmodified query returns 2861 rows, while the constraints “touch” 21million rows and 586million rows respectively, which would not be expected for a “personal” constraint; data about a particular person will often be very small compared to the entire database that might contain data from thousands of people. This of course can be false in some domains (e.g., genomic databases). For (11,3), the constraint itself when executed as a standalone query is so slow that it timed out after 1 hour; we would expect poor performance when used as a constraint. Finally, for (15,4), the constraint requires a join across 4 relations, even though the unmodified query runs over only 1 relation, which again seems unlikely for a ’personal’ constraint.

As Figure 5 shows, the bulk of the queries have little (1-2x slowdown) impact on execution time, and some are even faster than the unmodified queries (when, e.g., the query returns a large number of tuples while the rewriting a small). Only 4 queries (2,4,13,16) slow down as more tuples are removed, and the slowdown seems to be linear. From these results, it appears that the number of tuples removed is not a key factorInstead, the number of tuples "touched" by the constraint seems to be crucial. This is encouraging for a system designed to honor personal consent precision statements.

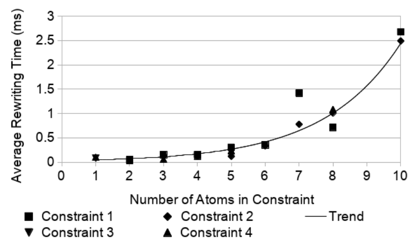

Figure 4: Effect of the number of atoms within the constraint on the time it takes to rewrite a query to be consent abiding.

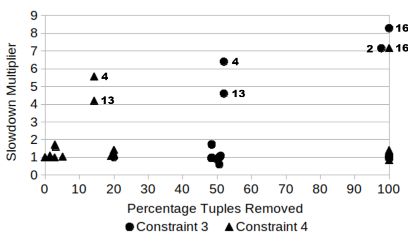

Figure 5: Slowdown of query vs percentage of tuples removed to honor consent for TPC-H scaled by 7.

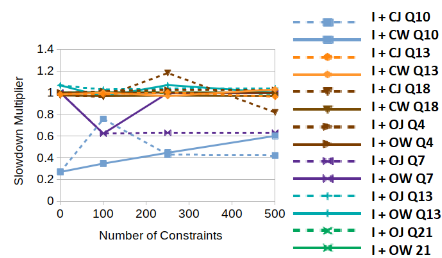

Figure 6: Effect of Institutional Policy and Personal Consent constraints on query time.

5.4 Scaling Constraints and Supporting Policies

In this section we explore scalability of our system with a large number of individual fine-grained consent statements by generating four sets (CW, CJ, OW, OJ) of 5000 independent constraints. The first set of constraints, *CW* (customer without joins) are atomic queries that: (1) contain a *Customer* atom with a random fixed *C_Custkey*, and (2) a random number of attributes are projected. For *CJ*, we randomly join each *CW* constraint with either *Orders* or *Nation* while fixing some of the values within *Orders* and *Nation*. The third and fourth sets, *OW* and *OJ*, were generated similarly, but using *Orders* instead of *Customer* as the base seed atom. For *OJ* the base constraints are joined to either *Lineitem* or *Customer*. This allowed us to create many relevant constraints to a query (having non-trivial rewritings) which however do not remove much, consistent with our intention of them being personal constraints. For example, for TPC-H query Q10 rewritten using the constraints from *CW*, 1650 (33%) of the 5000 constraints are applicable, and result in the removal of .003% of the unmodified query's 10million tuples. For scalability testing, we rewrite the query using different numbers up to 5000 constraints. Fig. 7 shows the results of this experiment, averaging over each query. There are more *OW* and *OJ* data points than *CW* and *CJ* because the CQ parts of only TPC-H queries 10, 13, and 18 refer to *Customer*, while there are many TPC-H queries for *Order*. Figure 7 shows the performance when a large number of personal constraints are applied at the same time. Notice in Fig. 7a the execution time grows linearly. Smilarly, Fig. 7b shows that as we encounter more unifications, we have a linear slowdown. Fig. 7c shows the time to rewrite queries to become consent-abiding; this time grows linearly with more constraints, however it is still just a few milliseconds. Lastly, the sets of constraints that include joins (*CJ*, *OJ*) exhibit a greater slowdown than the sets without joins, confirming that simpler constraints are faster to enforce.

Finally, we look at the interactions of institutional policies and personal consent. We encode three institutional policies that would mimic HIPPA [2] requirements on health data: (i) do not associate name, ID for patients in category X; (ii) do not associate name, phone number when country=X; (iii) do not share addresses once patients have completed treatment. In TPCH we implement (i) by not sharing keys and names of customers in a particular building, (ii) as is, and (iii) not sharing addresses of customers with finished transactions. These policies were added to the individual constraints described above. Fig. 6 shows the slowdown of 7 queries in this setting. The institutional constraints do not seem to affect or, in cases, even speed up queries by significantly reducing result tuples.

5.5 Real dataset

Our last set of experiments was performed using the MIMIC-III dataset [3] that contains real but de-identified critical care unit data. This dataset comprises of 27 tables with millions of tuples for a total database size of 47GB, where one of the tables, `ChartEvents`, is orders of magnitude larger than the rest, occupying 33GB on disk. MIMIC-III has been used in a similar setting before in [48] where the authors study the problem of policy enforcement on query metadata (e.g., do not answer a query if the query issuer has done too many queries in the last day). Inspired by [48] we created a set of 7 queries such as "select patient id and gender from patients that have their heart rate monitored", or "select all instrument measurements for a specific patient", or "select admission information for patients who are in ICU and the type of service they are receiving". The entire list of our queries can be found on our github. To create constraints for our experiments we used variations of the queries to create patterns of constraints that we instantiate with hundreds of different constants from the database, e.g., patient ids. This way we created 1000 particular constraint instances per query. The results of our experiment can be seen in Fig.8. Query Q4 is a particularly slower query that is using `ChartEvents`; while most queries can be answered in under a few seconds for hundreds or a thousand constraints Q4 takes up to 20sec (Fig.8(a)). Nevertheless, since the original query is very slow to begin with, the slowdown for this query in the face of the constraints is relatively ok: 20 times slower for 500 constraints, and around 50 times slower for 1000 constraints. This slowdown (of a few tens of times) is common for all queries (Fig.8(a) and (b)), except Q6 and Q7 that were particularly designed to overlap with all thousand constraints in more than one ways (each query-constraint pair can have multiple unifications). Thus Q6 and Q7 in Fig.8(b) become linearly slower with the addition of constraints, by hundreds of times. Still, these queries can be rewritten and executed in a few seconds even with 1000 constraints (Fig.8(a)). Our experiments with this dataset show that in realistic scenarios and with real data, consent-abiding query answering can become more challenging, inducing overhead for certain queries which however seems linear. Our results are promising and show that consent can be enforced in seconds also in this setting.

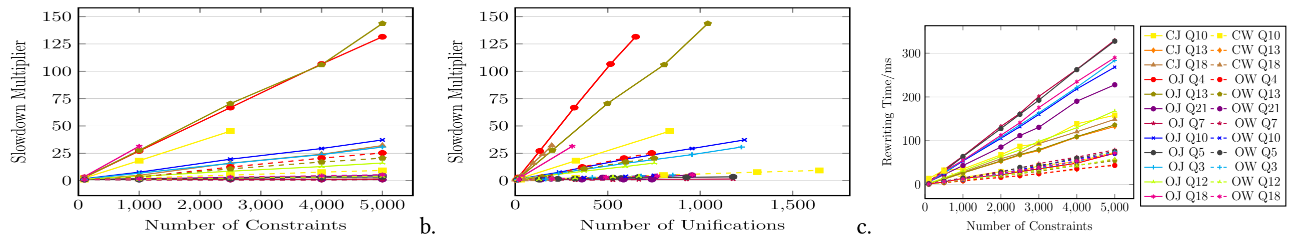

Figure 7: a. Number of constraints impact on query execution time. b. The slowdown per number of unifications (number of UNION elements in our rewriting). c. The rewriting time to create consent-abiding queries across each size of constraint set.

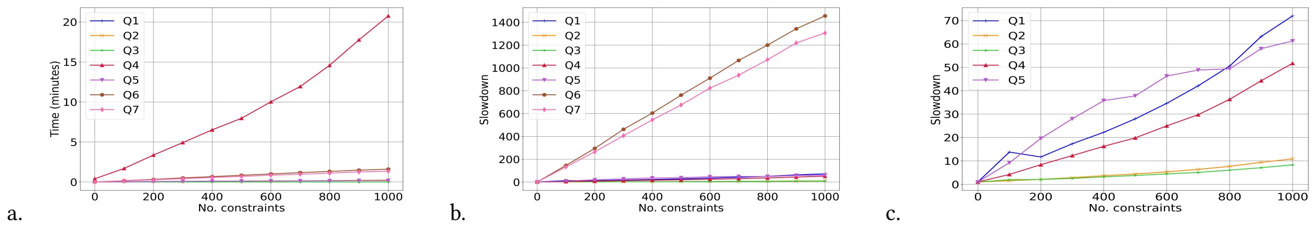

Figure 8: a. Consent-abiding execution time vs number of constraints. b. Slowdown per number of constraints. c. The 5 queries from (b) with the smallest slowdown.

6 RELATED WORK

Research in data privacy can be broadly classified into two categories. First, we have statistical approaches [25, 35, 45] that focus on protecting the identity of individuals when releasing aggregate statistics or performing machine learning tasks, by inserting noise to data or masking/supressing the data. Second, we have logic-based approaches, where usually data is not distorted but rather a declarative layer written in logic, often using logical views [11, 16, 22, 28, 40, 41], is used to filter queries that might compromise privacy. This and relevant problems such as inference control [12, 26], or controlled query evaluation [13, 27], study when sensitive information can be disclosed from non sensitive data. Often, authorisation views [20, 37, 51] extend classical access control approaches [6, 18, 44, 49, 50] by defining a logical security layer and assuming a secret query/policy [9–11, 22, 36, 39]; the problem then is to design safe views, or decide if views expose secret answers via inference. Our approach is much closer to the logic-based approaches that protect a secret query's answer (for us a negative constraint). However, instead of limiting access to data in order to hide the constraint's answer, we explicitly describe this undesired use in a formal contract released to the service-provider. To the best of our knowledge, ours is the first approach to address this problem from a collaborative privacy perspective. Datalawyer [48], seems the only other approach in Databases that is motivated by supporting top-down data privacy policies, without aiming to prevent access to information. In its current version, it is using data-dependent semantics while our approach is data-independent. It also follows an "accept or reject the query" policy, in contrast to our approach that gives back the maximal consent-abiding part of the answer. Hippocratic Databases [33], which we compare against, were inspired by implementing P3P-like policies [24] within an RDBMS, and assume an access control perspective. We envision our approach also being useful in data auditing scenarios [5, 38].

7 CONCLUSION

Society's interest is shifting towards underlying systems that appropriately manage and facilitate fine-grained consent. We present the first, foundational work to look at how to manage, and honor, fine-grained, personal consent and contracts of data usage in a collaborative privacy setting. We implement a solution in an open source DBMS, and show that we can cover and outperform previous works such as [33] on information disclosure while having much larger expressive power. Our expressive consent can be managed with small overhead to queries, even in the face of thousands of constraints. Conjunctive queries are already employed as a privacy policy language in several areas [10, 23, 36]. Many future directions for this research open up, some of which we are already working on: bringing this language closer to the end-user [32], investigate the benefits of our alternative annotation-based implementation, investigate interaction with reasoning constraints, precisely formalize aggregation, and finally look into how we can support a service provider to enforce consent across multiple queries, in essence,"propagate" consent by inducing more consent constraints relative to a query answer for later use; we envisage the latter happening with a query-answering-using-views technique or similar [30, 31].

ACKNOWLEDGMENTS

George Konstantinidis was supported by the Alan Turing Institute through a Fellowship and an Enhancement Project. Adriane Chapman was partially supported by EPSRC (EP/SO28366/1). We deeply thank Paolo Pareti and Muhammed Qaid for helping with some of the experiments.

REFERENCES

[1] https://github.com/georgekon/enabling-personal-consent, 2021.
[2] Health insurance portability and accountability act of 1996. *Public law, United States of America*, 104:191, 1996.
[3] Medical Information Mart for Intensive Care III. *https://mimic.mit.edu/docs/iii/*, 2018.
[4] S. Abiteboul, R. Hull, and V. Vianu. *Foundations of Databases*. Addison-Wesley, 1995.
[5] R. Agrawal, R. Bayardo, C. Faloutsos, J. Kiernan, R. Rantzau, and R. Srikant. Auditing compliance with a hippocratic database. In *Proceedings of the Thirtieth International Conference on Very Large Data Bases - Volume 30*, VLDB '04, pages 516–527, 2004.
[6] F. Arjumand, G. Yumna, S. M. Awais, and A. A. Ghafoor. Towards attribute-centric access control: an abac versus rbac argument. *Security and Communication Networks*, 9(16):3152–3166, 2016.
[7] T. Baarslag, A. T. Alan, R. Gomer, M. Alam, C. Perera, E. H. Gerding, and m. schraefel. An automated negotiation agent for permission management. In *Proceedings of the 16th Conference on Autonomous Agents and MultiAgent Systems*, AAMAS '17, pages 380–390, 2017.
[8] J.-F. Baget, M. Leclère, M.-L. Mugnier, and E. Salvat. Extending decidable cases for rules with existential variables. In *Twenty-First International Joint Conference on Artificial Intelligence*, 2009.
[9] M. Benedikt, P. Bourhis, L. Jachiet, and M. Thomazo. Reasoning about disclosure in data integration in the presence of source constraints. In *Proceedings of the Twenty-Eighth International Joint Conference on Artificial Intelligence, IJCAI 2019, Macao, China, August 10-16, 2019*, pages 1551–1557, 2019.
[10] M. Benedikt, B. C. Grau, and E. V. Kostylev. Logical foundations of information disclosure in ontology-based data integration. *Artificial Intelligence*, 262:52–95, 2018.
[11] L. Bertossi and L. Li. Achieving data privacy through secrecy views and null-based virtual updates. *IEEE Transactions on Knowledge and Data Engineering*, 25(5):987–1000, 2012.
[12] J. Biskup and P. A. Bonatti. Lying versus refusal for known potential secrets. *Data & Knowledge Engineering*, 38(2):199–222, 2001.
[13] J. Biskup and P. A. Bonatti. Controlled query evaluation for known policies by combining lying and refusal. *Annals of Mathematics and Artificial Intelligence*, 40(1-2):37–62, 2004.
[14] J. Biskup, M. Bring, and M. Bulinski. Inference control of open relational queries under closed-world semantics based on theorem proving. *Information Systems*, 70, 2016.
[15] J. Biskup, R. Menzel, and J. Zarouali. Controlled management of confidentiality-preserving relational interactions. In G. Livraga, V. Torra, A. Aldini, F. Martinelli, and N. Suri, editors, *Data Privacy Management and Security Assurance*, 2016.
[16] P. A. Bonatti and L. Sauro. A confidentiality model for ontologies. In *International Semantic Web Conference*, pages 17–32. Springer, 2013.
[17] P. Buneman, S. Khanna, and W.-C. Tan. On propagation of deletions and annotations through views. In *Proceedings of the Twenty-first ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems*, PODS '02, pages 150–158, 2002.
[18] J.-W. Byun, E. Bertino, and N. Li. Purpose based access control of complex data for privacy protection. In *Proceedings of the Tenth ACM Symposium on Access Control Models and Technologies*, SACMAT '05, pages 102–110, 2005.
[19] D. Calvanese, G. De Giacomo, D. Lembo, M. Lenzerini, A. Poggi, M. Rodriguez-Muro, and R. Rosati. Ontologies and databases: The dl-lite approach. In *Reasoning Web International Summer School*, pages 255–356. Springer, 2009.
[20] D. Calvanese, G. De Giacomo, M. Lenzerini, and R. Rosati. View-based query answering in description logics: Semantics and complexity. *Journal of Computer and System Sciences*, 78(1):26–46, 2012.
[21] A. K. Chandra and P. M. Merlin. Optimal implementation of conjunctive queries in relational data bases. In *Proceedings of the 9th ACM Symposium on Theory of Computing (STOC)*, pages 77–90, Boulder, Colorado, 1977.
[22] R. Chirkova and T. Yu. Exact detection of information leakage: Decidability and complexity. In *Transactions on Large-Scale Data-and Knowledge-Centered Systems XXXII*, pages 1–23. Springer, 2017.
[23] G. Cima, D. Lembo, L. Marconi, R. Rosati, and D. F. Savo. Controlled query evaluation in ontology-based data access. In *International Semantic Web Conference*, pages 128–146. Springer, 2020.
[24] W. W. W. Consortium et al. The platform for privacy preferences 1.0 (p3p1. 0) specification. *World Wide Web Consortium*, 2002.
[25] C. Dwork. Differential privacy: A survey of results. In M. Agrawal, D. Du, Z. Duan, and A. Li, editors, *Theory and Applications of Models of Computation*, pages 1–19. Springer Berlin Heidelberg, 2008.
[26] C. Farkas and S. Jajodia. The inference problem: a survey. *ACM SIGKDD Explorations Newsletter*, 4(2):6–11, 2002.
[27] B. C. Grau, E. Kharlamov, E. V. Kostylev, and D. Zheleznyakov. Controlled query evaluation for datalog and owl 2 profile ontologies. In *Twenty-Fourth International Joint Conference on Artificial Intelligence*, 2015.
[28] B. C. Grau and E. V. Kostylev. Logical foundations of privacy-preserving publishing of linked data. In *Thirtieth AAAI Conference on Artificial Intelligence*, 2016.
[29] T. J. Green, G. Karvounarakis, and V. Tannen. Provenance semirings. In *Proceedings of the Twenty-sixth ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems*, PODS '07, pages 31–40, 2007.
[30] G. Konstantinidis and J. L. Ambite. Scalable query rewriting: a graph-based approach. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 97–108, Athens, Greece, 2011.
[31] G. Konstantinidis and J. L. Ambite. Optimizing the chase: Scalable data integration under constraints. *Proceedings of the VLDB Endowment*, 7(14):1869–1880, 2014.
[32] G. Konstantinidis, A. Chapman, M. J. Weal, A. Alzubaidi, L. M. Ballard, and A. M. Lucassen. The need for machine-processable agreements in health data management. *Algorithms*, 13(4):87, 2020.
[33] K. LeFevre, R. Agrawal, V. Ercegovac, R. Ramakrishnan, Y. Xu, and D. DeWitt. Limiting disclosure in hippocratic databases. In *Proceedings of the Thirtieth International Conference on Very Large Data Bases - Volume 30*, VLDB '04, pages 108–119, 2004.
[34] J. W. Lloyd. *Foundations of logic programming*. Springer Science & Business Media, 2012.
[35] A. Machanavajjhala, M. Venkitasubramaniam, D. Kifer, and J. Gehrke. l-diversity: Privacy beyond k-anonymity. In *ICDE*, 2006.
[36] G. Miklau and D. Suciu. A formal analysis of information disclosure in data exchange. In *Proceedings of the 2004 ACM SIGMOD International Conference on Management of Data*, SIGMOD '04, pages 575–586, 2004.
[37] A. Motro. An access authorization model for relational databases based on algebraic manipulation of view definitions. In *[1989] Proceedings. Fifth International Conference on Data Engineering*, pages 339–347, 1989.
[38] R. Motwani, S. U. Nabar, and D. Thomas. Auditing sql queries. In *ICDE*, pages 287–296, 2008.
[39] A. Nash, L. Segoufin, and V. Vianu. Views and queries: Determinacy and rewriting. *ACM Trans. Database Syst.*, 35(3):21:1–21:41, 2010.
[40] S. Rizvi, A. Mendelzon, S. Sudarshan, and P. Roy. Extending query rewriting techniques for fine-grained access control. In *Proceedings of the 2004 ACM SIGMOD International Conference on Management of Data*, SIGMOD '04, pages 551–562, 2004.
[41] A. Rosenthal and E. Sciore. View security a the basis for data warehouse security. In *DMDW*, 2000.
[42] Rosenthal et al. Kairon consents. *http://kaironconsents.sourceforge.net/*, 2012.
[43] Y. Sagiv and M. Yannakakis. Equivalences among relational expressions with the union and difference operators. In *Journal of the ACM*, volume 27, pages 633–651, 10 1980.
[44] M. I. Sarfraz, M. Nabeel, J. Cao, and E. Bertino. Dbmask: Fine-grained access control on encrypted relational databases. In *Proceedings of the 5th ACM Conference on Data and Application Security and Privacy*, CODASPY '15, pages 1–11, 2015.
[45] L. Sweeney. k-anonymity: A model for protecting privacy. *International Journal of Uncertainty, Fuzziness and Knowledge-Based Systems*, 10(05):557–570, 2002.
[46] W.-C. Tan. Containment of relational queries with annotation propagation. In G. Lausen and D. Suciu, editors, *Database Programming Languages*, pages 37–53, Berlin, Heidelberg, 2004. Springer Berlin Heidelberg.
[47] Transaction Processing Performance Council. TPC Benchmark H. *http://www.tpc.org/tpch/*, 2018.
[48] P. Upadhyaya, M. Balazinska, and D. Suciu. Automatic enforcement of data use policies with datalawyer. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, SIGMOD '15, pages 213–225, 2015.
[49] Q. Wang, T. Yu, N. Li, J. Lobo, E. Bertino, K. Irwin, and J.-W. Byun. On the correctness criteria of fine-grained access control in relational databases. In *Proceedings of the 33rd International Conference on Very Large Data Bases*, VLDB '07, pages 555–566, 2007.
[50] C. Xu, J. Xu, H. Hu, and M. H. Au. When query authentication meets fine-grained access control: A zero-knowledge approach. In *Proceedings of the 2018 International Conference on Management of Data*, SIGMOD '18, pages 147–162, 2018.
[51] Z. Zhang and A. O. Mendelzon. Authorization views and conditional query containment. In *International Conference on Database Theory*, pages 259–273. Springer, 2005.