Retrofitting GDPR Compliance onto Legacy Databases

Archita Agarwal[*][†]
Denison University
agarwala@denison.edu

Marilyn George[*]
Brown University
marilyn_george@brown.edu

Aaron Jeyaraj
Brown University
aaron_jeyaraj@brown.edu

Malte Schwarzkopf
Brown University
malte@cs.brown.edu

ABSTRACT

New privacy laws like the European Union's General Data Protection Regulation (GDPR) require database administrators (DBAs) to identify all information related to an individual on request, *e.g.*, to return or delete it. This requires time-consuming manual labor today, particularly for legacy schemas and applications.

In this paper, we investigate what it takes to provide mostly-automated tools that assist DBAs in GDPR-compliant data extraction for legacy databases. We find that a combination of techniques is needed to realize a tool that works for the databases of real-world applications, such as web applications, which may violate strict normal forms or encode data relationships in bespoke ways. Our tool, GDPRizer, relies on foreign keys, query logs that identify implied relationships, data-driven methods, and coarse-grained annotations provided by the DBA to extract an individual's data.

In a case study with three popular web applications, GDPRizer achieves 100% precision and 96–100% recall. GDPRizer saves work compared to hand-written queries, and while manual verification of its outputs is required, GDPRizer simplifies privacy compliance.

PVLDB Reference Format:
Archita Agarwal, Marilyn George, Aaron Jeyaraj, Malte Schwarzkopf. Retrofitting GDPR Compliance onto Legacy Databases. PVLDB, 15(4): 958–970, 2022.
doi:10.14778/3503585.3503603

PVLDB Artifact Availability:
The source code, data, and/or other artifacts have been made available at https://github.com/architaagarwal/GDPRizer.

1 INTRODUCTION

Many new privacy laws contain provisions that allow individuals whose data is stored and processed to request a copy of their data. For example, the European Union's General Data Protection Regulation (GDPR) [19] includes rights of access, erasure, and data portability, all of which require accurate identification of all data related to an individual "data subject" (a natural person). Other laws contain similar provisions, such as the "Right to Know" in the California Consumer Privacy Act (CCPA) [14] or in Virginia's Consumer Data Protection Act (VCDPA) [44, §59.1-573]. To comply with these *data access requests*, a database administrator (DBA) must be able to identify all data related to a particular individual.

Identifying all data related to an individual in an existing legacy database can be daunting. Legacy databases have schemas designed without future data access requests in mind, and consequently lack the necessary secondary indexes or metadata to look up data by the associated individual [41]. Moreover, what data should actually be returned often involves application-specific policy choices. As a result, DBAs and developers must manually design and write queries that identify and extract the relevant data. In practice, this may require several iterations to follow indirect dependencies across tables, extract further related data identified by a foreign key, and post-process it appropriately. Consider the classic TPC-H benchmark: its `supplier` and `customer` tables identify individuals, who are linked to orders, parts, and addresses stored in other tables. To satisfy a data access request on behalf of a customer, the DBA must (at least) query the tables connected to `customer` via either direct foreign keys, such as `orders`, or via indirect ones, such as `lineitem`. These queries require more than a simple transitive closure over foreign keys. Querying *all* tables connected via foreign keys might return more data than required (*e.g.*, revealing the personal details of a supplier to a customer); data might need post-processing to remove internal or private details; or data might be missing, as many applications have imperfect foreign key specifications in their schema. These challenges show up in the databases of real-world web applications, such as Lobsters [26] and HotCRP [24].

In this work, we investigate whether an RDBMS or external tool can help DBAs and developers *retrofit* data access request compliance onto legacy databases. The goal of our tool, GDPRizer, is to generate a set of queries that extract or delete an individual's information in accordance with data access requests. To achieve this, we had to establish what inputs GDPRizer requires to generate queries that extract complete and accurate information. GDPRizer must be practical for real applications' databases, whose schemas have evolved over time and are often messy.

Our work shows that high accuracy and complete data extraction hinges upon solving two challenges. First, GDPRizer must identify how data is related across tables, and ensure that a data access request returns rows from all relevant tables. Missing a relationship between tables results in missing rows in the output, which can make the data access request fall short of legal compliance. Yet, the dependencies can be non-obvious, as an application might *e.g.*, encode a relationship using particular attribute values. For example, HotCRP indicates co-authorship on a paper via an entry in the `PaperConflict` table, with the `conflictType` column set to a special numeric constant indicating a "co-authorship" conflict type. GDPRizer should—with suitable inputs—understand this sort of dependency. Second, GDPRizer must avoid extracting too much data.

Proceedings of the VLDB Endowment, Vol. 15, No. 4 ISSN 2150-8097.
doi:10.14778/3503585.3503603

[*]Both authors contributed equally.
[†]Part of the work completed while at Brown University.

Even though a table may store or reference data associated with an individual, returning that data might overreach. For example, an author's data access request in HotCRP should return reviews for their papers, but the `Review` table's rows also contain the identity of the reviewer. To preserve reviewer anonymity, the rows returned to an author must have the reviewer ID erased.

GDPRizer relies on two key ideas to solve these challenges: a *relationship graph analysis* helps identify implicit dependencies across tables, and *schema-oriented customizations* limit the data extracted based on coarse-grained schema annotations provided by the DBA. Depending on the application, GDPRizer uses up to five types of input: (*i*) explicit foreign keys, if present; (*ii*) a log of runtime queries the application executes, which helps infer relationships between the columns; (*iii*) foreign keys detected through data-driven methods; (*iv*) schema annotations that specify connections between tables that are connected by implicit data or those whose column relationships cannot be inferred from queries or foreign keys; and (*v*) schema annotations that specify what connections across tables to prune, and how to filter the extracted data. GDPRizer uses these inputs to traverse the database, extracting the information required to satisfy a data access request. It also provides warnings to the DBA if the extracted data might be incomplete.

We implemented a prototype of GDPRizer and evaluated it with a synthetic schema (TPC-H) as well as three real applications: Lobsters [26], a Reddit-style news aggregator application that declares some foreign keys in its schema; HotCRP [24], a conference paper review application without any explicit foreign keys; and WordPress [8], a popular blogging platform with a non-traditional, performance-optimized schema. Our experiments show that GDPRizer achieves 62–100% precision (fraction of extracted records that are correct) and 66–100% recall (fraction of records extracted) without manual input for these applications. Manual customizations increase this to 100% precision and 96–100% recall.

In summary, we make the following key contributions:

(1) We investigate what satisfying data access requests over legacy schemas entails, and what information beyond existing RDBMS abstractions (such as foreign keys) is needed.
(2) We describe an algorithm to traverse a database and extract the information needed to satisfy a data access request.
(3) We present GDPRizer, a tool that implements this algorithm and interactively guides a developer or DBA in generating the queries for data access requests.
(4) We evaluate a GDPRizer prototype, demonstrating high accuracy on three real web applications' databases, and compare GDPRizer to bespoke GDPR compliance plugins for the WordPress blogging platform.

GDPRizer's automation is fundamentally limited by the fact that legacy databases' schemas may fail to reflect key application semantics, and that data-driven methods for foreign key discovery produce imperfect recall and require manual verification. However, our work shows that it is possible to much reduce the manual labor required to satisfy data access requests.

2 BACKGROUND AND RELATED WORK

High-profile data breaches and an increasing interest in consumer privacy regulation have led to a glut of new privacy laws in recent years. The European Union's GDPR and California's CCPA are particularly well known due to their comprehensiveness and reach—both laws effectively apply globally since it is difficult to establish if a user is in the E.U. or in California—but many similar regulations now exist. China [43], India [22], Brazil [31], Thailand [20], and various U.S. states [44] have passed privacy laws that give new rights to consumers to receive information about and control how their data is processed. Failure to comply with these laws can lead to reputational damage, revenue loss, and substantial fines [9].

Data Access Requests. Most of these privacy laws grant individuals the right to request a copy of their data from those who store or process it, and to ask for its removal. In the GDPR, for example, these rights of a "data subject" (a natural person) are codified in Articles 15 ("Right to Access") and 17 ("Right to Erasure"). Other laws contain similar provisions, such as the CCPA's and VCDPA's "Right to Know". We refer to the power granted by these provisions as a *data access request*. A data access request requires the party controlling the data (a "data controller" in GDPR lingo) to identify all information they hold about the requester. Satisfying the request requires care, as the data returned must not violate the privacy of other individuals [19, Art. 15, §4], so post-processing of the data identified is typically required.

Web Applications. Privacy laws have a broader scope than just web applications, but their impact is particularly serious for organizations that operate web services. Web applications often use a database backend coupled with stateless frontend logic. Popular frameworks like Ruby on Rails [5] and Django [1] use a relational database for storage by default. Yet, legacy web applications' schemas were developed without attention to data access requests, and the relational storage paradigm's strength—organizing records by type in tables—fundamentally mixes different individuals' data. This makes the task of identifying the data associated with an individual complicated and application-specific. Working out what information a data access request needs to return requires application developers or DBAs to navigate application-specific table and column names, as well as underspecified or implicit relationships that indicate records' association with individual users. This requires substantial manual, error-prone labor for many applications.

One might be tempted to believe that abstractions for relating entities across tables (such as foreign keys) could help. While this is true in theory—a data access request is, essentially, a recursive traversal of related entities from a starting entity in a table—real-world application schemas frequently fail to conform to third normal form (3NF) or lack the required keys. For example, we studied nine open-source web applications, ranging from chat plugins to social networks, blogging and conference review platforms [11, 15, 24, 26, 30, 33, 35, 37, 40] and found that only two of them specify foreign keys in their schema.

Data-driven functional dependency (FD) detection. To address the problem of missing foreign keys, database researchers developed techniques to detect the presence of such functional dependencies from the actual data. Generally, these techniques first identify an *inclusion dependency* between two columns: *i.e.*, all values in one column (the candidate source) are contained in the set of values in the other column (the candidate destination) [13, 29, 34]. A set of heuristics based on typical properties of foreign keys—*e.g.*, a broad range of covered values, evenly distributed values, and

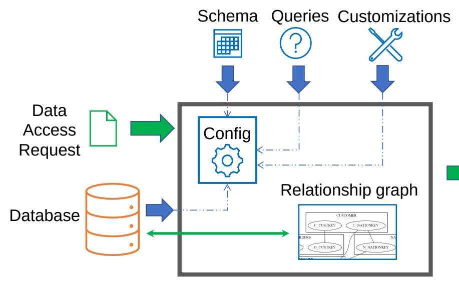


Figure 1: GDPRizer overview: given a data request, GDPRizer queries the database and post-processes the retrieved data according to the configuration.

similar column names—help filter the inclusion dependencies down to candidate foreign keys [16, 36, 45]. Systems for data cleaning and exploration have successfully applied these ideas [10, 18]. Similarly, these techniques could help detect the data required to satisfy data access requests, subject to a manual check and possible redaction of private information. Lopes et al. showed that join queries can help refine inclusion dependencies [27]; we investigate this and other approaches to combining queries and database contents.

Compliance plugins. For some popular frameworks, application-specific, third-party privacy compliance plugins are available. For example, the WordPress blogging platform's plugin registry lists dozens of GDPR plugins related to cookie consent [42] or GDPR compliance [17, 32]. However, such plugins can have serious deficiencies (as we will show in §7.5), but the DBA must blindly trust their correctness. For custom web applications or less popular frameworks, no such plugins are available.

Other approaches to compliance. Some researchers have proposed entirely new database systems [25, 38] or storage hardware [23] to achieve privacy compliance. While helpful for future deployments, these systems do not help legacy databases comply with data access requests. Other research has studied the performance costs of adding metadata structures (*e.g.*, secondary indexes) to existing databases to help satisfy data access [39, 41]. These techniques come with high overhead and without any automation. Odlaw [28] helps retrofit data access requests to legacy databases by building a graph of foreign key dependencies across tables and providing a graphical interface for DBAs to identify relevant data for a data subject. However, Odlaw assumes that the database schema contains explicit foreign keys, which many real applications lack.

3 GDPRIZER OVERVIEW

We present GDPRizer, a tool that retrofits compliance with data access requests onto legacy databases. GDPRizer explores a trade-off between fully manual, application-specific scripts that must be written with great care and human effort, and automated—but likely imperfect—general-purpose solutions. Our goal is to investigate the degree of automation that a tool can provide for data access requests over legacy databases, while minimizing any manual inputs. At a very high level, GDPRizer uses the database schema, database contents, and a query log from the application to extract semantic relationships between columns in the database. It then uses these relationships and manual customizations to generate a configuration that helps the tool satisfy data access requests for an individual by querying the database (Figure 1).

3.1 Automated Relationship Detection

A well-formed database schema in 3NF will indicate semantic relationships between tables via foreign keys: a foreign key indicates that the source table references objects in the destination table. This information is crucial to serve a data access request. Such a request starts with a data subject ID (DS ID), which typically corresponds to a row in some table—*e.g.*, `customer` or `supplier` in TPC-H, since both customers and suppliers are data subjects under laws like the GDPR. A foreign key into the table that contains data subjects indicates that records in another table are associated with the data subjects. A transitive foreign key (often) indicates the same about an object that is two or more steps away from a data subject table. When present, GDPRizer therefore uses foreign keys to detect data related to a data subject. However, practical application database schemas often lack FKs. Any real-world compliance solution therefore has to tackle challenges such as lack of referential integrity and implicit or conditional relationships between columns.

If the database schema lacks sufficient information about foreign keys, GDPRizer uses alternate sources of information to identify relationships between tables. Two key sources are the application queries and the relationships encoded in the database contents.

The idea behind using application queries is that a runtime join between two tables often implies a foreign key relationship between these tables, particularly if the destination column is a table's primary key column. For example, TPC-H joins `customer.c_custkey` with `orders.o_custkey` for a query that summarizes a customer's order information, matching the foreign key constraint between the two columns. While joins on non-foreign key columns are possible, they are fairly rare in practical web applications. Keeping this in mind, GDPRizer uses application query logs to supplement the relationship information provided by explicit foreign keys in the schema. Query logs are easy for DBAs to obtain (*e.g.*, by sampling some fraction of runtime queries, or by enabling query logging).

Database contents themselves can also empirically indicate the presence of FKs. If the database contents are available, GDPRizer runs techniques from the literature on functional dependency detection to first identify inclusion dependencies and then filter them down to likely foreign key dependencies via standard heuristics [36].

Relationship Graph. GDPRizer uses the foreign keys (if present), the joins in the query log, and the candidate foreign keys identified using database contents to generate a *relationship graph*. The relationship graph is a raw, unprocessed set of known relationships between columns across tables. Each column in the database is represented by a vertex in this graph. The relationship between a pair of columns—actual or candidate foreign key constraints, or joins—is represented by an edge between the columns. A table is represented by multiple, grouped vertices in the relationship graph. Figure 2 shows GDPRizer's relationship graph for HotCRP, and highlights which edges occur as joins, which are candidate foreign keys identified via data-driven methods, and which are both.

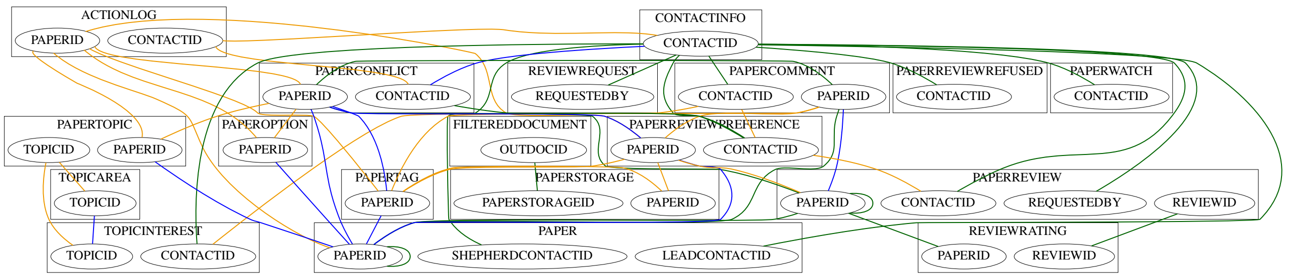


Figure 2: Relationship graph of HotCRP [24]. Each box represents one table and its corresponding columns. HotCRP's schema lacks explicit foreign keys, so edges show candidate foreign keys inferred from the data (orange), from joins (green), or from both (blue).

Data Extraction. When GDPRizer receives a data access request, it traverses the relationship graph to extract data relevant to the data subject. The traversal begins at the key column that contains the data subject's primary identifier. GDPRizer then proceeds to extract records directly or indirectly connected to this key column. In other words, GDPRizer explores the transitive closure of all the connected tables in the graph to extract the relevant data.

3.2 Manual Customizations

However, GDPRizer (and likely any general-purpose tool) needs additional input to identify application-specific semantic structure in the database. When the relationship graph fails to express some aspects of the application semantics, GDPRizer allows for a domain expert, such as the application developer or DBA, to intervene and customize either the relationship graph itself, or the data output after traversal. We aim to minimize this intervention in GDPRizer, but our case studies show that a modicum of manual input is often required. GDPRizer's customizations fall into four categories:

(1) Edge pruning. The relationship graph for an application sometimes contains relationships that are irrelevant to a data access request. This can happen because a foreign key connects internal application data rather than user data, or because the application joins columns that are semantically unrelated, or because similar data in two columns suggest a foreign key where none exists. For GDPRizer to ignore these irrelevant relationships, the relationship graph needs customizing. GDPRizer supports edge pruning annotations, which allow the DBA to indicate that all edges (*i.e.*, relationships) incident on a particular column should be ignored.

(2) Adding missing edges. Even after using information from the schema, runtime queries, and database contents, the relationship graph may still be missing relationships. For example, the relationship graph for WordPress has no edge between the table with user information and the table that holds comments, even though it is clear from the application semantics that they contain related columns. When tables are disconnected in the relationship graph, GDPRizer prompts the DBA to manually "connect" the disjoint components. GDPRizer uses heuristics based on data types and column content to suggest edges that might have semantic significance. The DBA considers the list of suggestions and adds the missing relationships.

(3) Data-dependent and conditional relationships. A more involved customization is necessary if the application has implicit or conditional relationships that cannot be expressed in terms of existing columns. This happens *e.g.*, if a second column indicates the semantic meaning of a foreign key. For example, paper conflict types in HotCRP represent both co-authorship and conflicts of interest, which have different semantics for data access requests. The DBA can provide an input that transforms the data such that the relationship is direct and explicit. In particular, GDPRizer supports creating views that contain rows from a source table only if a predicate over the rows holds true. These views become part of the graph, replacing other tables and edges in data extraction.

(4) Output filtering. Once GDPRizer has completed the relationship graph traversal and queried the database, it may be necessary to filter the resulting records to remove personal information of other individuals (*e.g.*, reviewer details in HotCRP) or unrelated data (*e.g.*, internal supplier information in TPC-H). The DBA specifies columns to filter from the output by annotating the schema, and GDPRizer removes or rewrites these columns.

GDPRizer only needs to be configured once and the manual customizations are one-off for a given database. Once a relationship graph exists, GDPRizer saves the customizations as a configuration for all future data access requests on the same schema, essentially creating an application-specific GDPR compliance tool with less effort than would have been necessary to write the queries manually.

4 RELATIONSHIP GRAPH

When GDPRizer receives a data access request, it identifies all data relevant to the individual making the request (the "data subject"). GDPRizer assumes that the data subject is uniquely described by a row in a *primary table*. This is common: many applications have a users table with their users' details, or represent individuals as rows in tables associated with their role (*e.g.*, TPC-H's `customer` and `supplier`, or Lobsters's invited users in `invitations` and registered users in `users`). Other entities in the database refer to these primary table rows, establishing a *relationship*. For example, TPC-H has rows in the `order` table refer to customers by their unique key in the `customer` table (a foreign key constraint). These related rows might also be relevant to the customer, so GDPRizer must identify and use the relationship between `customer.c_custkey` and `order.o_custkey` to eventually extract the data. GDPRizer represents these relationships as edges in the relationship graph. The relationship graph combines relationships specified explicitly in the database schema with inferred relationships determined from application queries or values in the database.

Foreign keys. The most reliable source of relationships in a database is the database schema. If the schema is well-formed and in 3NF, it contains all the foreign key constraints, *i.e.*, all by-key relationships between columns across tables. In TPC-H, for example, there exists a foreign-key constraint between `customer.c_custkey` and `order.o_custkey`. These constraints can be conceptualized as a graph whose vertices are columns, and whose edges are foreign-key constraints. We refer to this graph as the *schema-based relationship graph*, $\mathtt{R}^{\mathtt{S}}$. However, many real-world database schemas fail to conform to strict 3NF, so $\mathtt{R}^{\mathtt{S}}$ alone is insufficient. GDPRizer must infer relationship information that the database schema lacks.

Queries. Another approach is to use the application semantics expressed in runtime queries to infer relationships within the database. For example, if the application joins two columns in the database at runtime, GDPRizer can infer that the columns are likely related, as the application assumes that they share data values. GDPRizer therefore processes a query log of the application, provided by the developer or DBA, to identify columns that queries join at runtime. Such a log is easy to obtain, *e.g.*, by enabling query logging in the database, or by instrumenting the application's DB access code. (Note that the log need not be complete—a sample is often sufficient.) These inferred relationships can also be represented as a graph. This is the *query-based relationship graph*, $\mathtt{R}^{\mathtt{Q}}$.

Patterns in the data. Data-driven methods can also help infer relationships between columns. Specifically, GDPRizer leverages techniques for foreign key discovery from a database's inclusion dependencies. Columns A and B form an inclusion dependency if all values in A are contained in B, *i.e.*, $Vals(A) \subseteq Vals(B)$. Intuitively, this will be the case for a well-formed foreign key relationship $A \rightarrow B$. GDPRizer creates the *data-driven relationship graph*, $\mathtt{R}^{\mathtt{D}}$, by first determining all candidate dependencies (all pairs of columns with the same datatype in the schema), and then filtering them down to inclusion dependencies by comparing the column values. For the remaining pairs, GDPRizer applies four heuristics to reduce the inclusion-dependent columns to likely foreign key columns:

(1) the Out of Range heuristic [36], which requires calculating the ratio of values in B that are outside of $[\min(A), \max(A)]$, keeping the pairs whose ratio is below a threshold;
(2) the Coverage heuristic [36], which calculates the ratio of values in B that are contained in A to the total unique values in B, keeping pairs that exceed a threshold;
(3) a Wilcoxon test to determine if the distribution of values in A is approximately a random sample of values in B , retaining columns for which the test passes; and
(4) a variation of the Jaro-Winkler similarity test to determine the similarity in the column names of A and B [16, 36], keeping columns with sufficient similarity.

We chose these heuristics because they were the most effective out of ten heuristics Rostin et al. studied [36]. Each candidate that passes is then added as an edge in $\mathtt{R}^{\mathtt{D}}$.

Combining the graphs. GDPRizer can combine the different relationship graphs to improve the accuracy of its data extraction. It always makes sense to use $\mathtt{R}^{\mathtt{S}}$, as it contains the most reliable information. We denote a union of $\mathtt{R}^{\mathtt{S}}$ with another graph, such as $\mathtt{R}^{\mathtt{Q}}$, as $\mathtt{R}^{\mathtt{S,Q}}$. When a query log is available, GDPRizer may use $\mathtt{R}^{\mathtt{Q}}$; and when the database contents are available and the database size feasibly allows analysis, GDPRizer may use $\mathtt{R}^{\mathtt{D}}$. Some of the edges in these graphs are distinct and some overlap (Figure 2). While merging the graphs is feasible ($\mathtt{R}^{\mathtt{Q}} \cup \mathtt{R}^{\mathtt{D}}$), such a union in practice usually results in a large relationship graph of poor quality. Instead, considering only edges that show up in queries *and* whose data suggests a foreign key may be an effective strategy to remove redundant edges. We refer to this combined graph of edges that pass both heuristics as $\mathtt{R}^{\mathtt{Q}} \cap \mathtt{R}^{\mathtt{D}}$. In the rest of this paper, we consider $\mathtt{R}^{\mathtt{Q}}$, $\mathtt{R}^{\mathtt{D}}$, and $\mathtt{R}^{\mathtt{Q}} \cap \mathtt{R}^{\mathtt{D}}$; when explicit foreign keys are available, we augment these with $\mathtt{R}^{\mathtt{S}}$.

5 GRAPH TRAVERSAL

Given a relationship graph, GDPRizer uses it to retrieve a data subject's records from the database. This requires GDPRizer to traverse the relationship graph, starting with a row in the primary table, and to generate meaningful queries as the traversal proceeeds.

A naïve graph traversal, which traverses all the edges, might extract too much or too little data:

(1) if several paths from the primary table to another table exist, each of them could lead to a different set of extracted rows, which might be too much data; and
(2) since there are (usually) no edges between the columns of the same table, the graph consists of many disconnected components, as shown by the colors in Figure 3. Any edge-based traversal that begins in one component will fail to extract data from the unreachable components.

GDPRizer addresses these challenges with heuristics based on *proximity* and *implied relationships*, as explained in the following.

To avoid overextraction and duplicate data, GDPRizer only visits each column once. When multiple paths to a column are available, GDPRizer picks the shortest one. Prioritizing in this way makes sense because, intuitively, columns that are "closer" to the starting column, *i.e.*, the primary key of the primary table, are more relevant to the data subject. Therefore, GDPRizer traverses the graph in a breadth-first manner rather than depth-first from the starting column. To address the second challenge, GDPRizer traverses the graph via two types of relationships (Figure 3):

- relationship edges, based on foreign keys, application joins, or database contents, such as the edge between columns A and B; and
- *implied relationships* between the columns of the same table, *e.g.*, columns B and C.

GDPRizer uses the relationship edges for data extraction, and the implied relationships to connect the components of the graph. Figure 3 shows a high-level overview of the graph traversal.

Relationship edges. Whenever possible, GDPRizer follows the graph's relationship edges. For example, in Figure 3, column A is the primary key column of the primary table and therefore the graph traversal's starting column. It follows that the columns accessible by a relationship edge from A, like B and E, are directly related to the data subject since they were either joined in the query set, or have a foreign key defined on them, or their data suggested a foreign key. For columns not directly linked to the primary key of the primary table, GDPRizer uses their distance from the starting column to decide which paths to explore. We call a column's distance from the starting column its *proximity*. Computing the proximity helps GDPRizer traverse columns nearer the starting column before ones that are further away, and ensures that it prefers shorter paths over longer ones. The proximity also naturally imposes a direction on the traversal of the edge between any two columns—the traversal proceeds from the column closer to the starting column to the column that is further away. GDPRizer uses a standard breadth-first traversal (BFT) from the starting column to compute the proximity of columns in that component of the graph. Starting the BFT from A, the proximity of columns B and E is 1, the proximity of F is 2 and that of G is 3. After exploring these columns, there are no more relationship edges to traverse, and the BFT stops.

Implied relationships and inferred proximity. At this point, GDPRizer has visited only one component of the graph. To continue the traversal through the remaining disconnected components, GDPRizer chooses a *secondary* starting column for each component, and computes the distance from this secondary starting column. GDPRizer chooses the secondary starting column using another proximity-based heuristic. Having already computed proximities for the component that contains the starting column, GDPRizer now considers the untraversed *siblings* of the traversed columns. (Two columns are siblings if they belong to the same table.) This set of untraversed siblings must be part of disconnected components, as they would have been traversed already if they were part of the starting column component. GDPRizer uses implied relationships between siblings to infer the proximity of an untraversed sibling column: the proximity of an untraversed column is the minimum proximity over its siblings + 1. In Figure 3, columns C, D, J are siblings of B, E, G, with proximities 2, 2, 4 respectively. Setting this proximity is equivalent to traversing an (implied) relationship edge from the sibling with minimum proximity. After this augmentation, GDPRizer continues with the proximity rule used for the first component, and picks the column with the minimum proximity as the secondary starting column. GDPRizer then repeats a BFT using the relationship edges in that component. The secondary starting columns in the figure are then C, D, J and the respective BFTs are indicated on the graph. This process of using relationship edges and implied relationships alternately continues until GDPRizer has traversed all the columns in the graph, or no more viable sibling columns exist.

Data extraction. GDPRizer's data extraction proceeds alongside the graph traversal. GDPRizer starts by issuing a query for the all records associated with the data subject identifier (DS ID) in the primary table, and then associating the value of DS ID with the starting column. In subsequent steps, for each relationship edge between columns A and B, traversed $A \longrightarrow B$, column A already has some associated value from the previous step of the traversal. GDPRizer issues a SQL query for all the records with this value in column B, as the relationship edge requires the values of A and B to be identical. This process repeats for all relationship edges. When GDPRizer uses an implied relationship, say from B to C, it already knows the value for B (which was reached via the relationship edge from A to B). GDPRizer queries the records with that value in B and obtains the matching values in sibling column C. These associated values then initiate the traversal (and thus, data extraction) in that component of the graph. Finally, GDPRizer combines the SQL queries' results, optionally redacting them as described in §6.2.

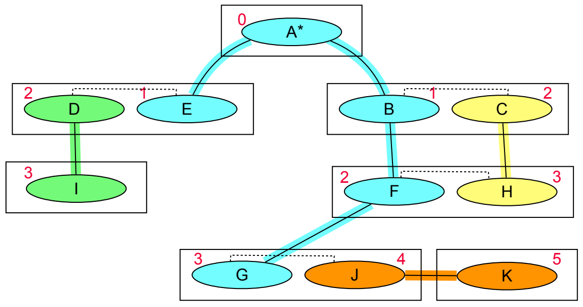


Figure 3: An example graph traversal: A **is the primary column of the primary table (*), and each column is annotated with its proximity. We show the first BFT in blue and subsequent BFTs using implied relationships in green, orange and yellow.**

6 CUSTOMIZING THE GRAPH TRAVERSAL

In practice, application databases' structure may have semantic properties that the relationship graph fails to capture. GDPRizer offers manual customization options to modify the relationship graph and the extracted data. These options are semi-automated: the developer or DBA adds customizations either in response to prompts from GDPRizer (*e.g.*, if there are disconnected components of the relationship graph that GDPRizer's traversal cannot reach) or after inspecting the data returned by GDPRizer's extraction.

6.1 Graph customization

GDPRizer supports three graph customizations: (*i*) edge removal or pruning; (*ii*) edge addition; and (*iii*) vertex addition.

Edge Pruning. When a database schema contains columns that GDPRizer should avoid using to extract data, a DBA can annotate the columns to avoid traversal (and further data extraction) via these columns. In HotCRP, for instance, a paper in the `Paper` table is linked to conflicted individuals' records via a relationship to `PaperConflict`, which in turn has a relationship with `ContactInfo`. GDPRizer should not extract information about the conflicted individuals. To avoid this, the DBA might prune the `contactId` column in the `PaperConflict` table, removing all edges incident on it.

Edge Addition. Edge addition becomes necessary when the relationship graph lacks edges to some tables in the schema. This happens if: (*i*) in R^S, no explicit foreign keys were specified between the tables, and, (*ii*) in R^Q, the tables were never joined by application queries, or (*iii*) in R^D, no data dependency was discovered between the tables. In such cases, GDPRizer prompts the DBA to "connect" these tables to the rest of the relationship graph. The prompt provides a list of plausible edges based on matching column datatypes, primary key constraints, and inclusion dependencies in the data.

Vertex Addition. Vertex addition is the most complex customization GDPRizer supports. It is required when the database contains conditional or implicit relationships. These relationships are computed programmatically, rather than being expressed as simple foreign keys or joins. One example of this is how HotCRP represents the co-author relationship on a paper. A co-author is specified using a row in the `PaperConflict` table, with the `conflictType` column set to a specific bitfield value. Based on this relationship, GDPRizer must extract the data for papers that a user has co-authored. Rows with other conflict types (*e.g.*, institutional, advisor-advisee) also have relationships with `Paper` and `ContactInfo`, but GDPRizer must not extract their data.

To support this, GDPRizer allows a DBA to add a *virtual column* to a table, effectively defining a view. The virtual column transforms the implicit or conditional constraint into an explicit column, *i.e.*, a vertex in the relationship graph. The virtual column is derived from a source column, and GDPRizer copies all edges of the source column to the virtual column. In HotCRP, the DBA provides GDPRizer with the query to create a view of the `Paper` table. The view contains one row for each co-author user ID, and bases the co-author ID column on the source column `leadContactId`. Hence, GDPRizer copies all the relationships of `leadContactId` to the virtual co-authors column. The graph traversal then uses this view in place of the `Paper` table.

6.2 Output customization

GDPRizer must also take care to avoid returning internal information or the information of other data subjects as part of the data it extracts for a data access request. For this purpose, and to reduce unnecessary output, GDPRizer supports post-processing of the data extracted during the graph traversal.

Filtering. After the data is extracted, GDPRizer allows the DBA to filter out any unnecessary columns from the output using filtering annotations. This is expressed as a list of columns to drop or rewrite in the output. In order to reduce manual input, GDPRizer automates filtering for one specific case: a *mapping table*, which is a table that consists entirely of relationship columns (*i.e.*, all columns are foreign keys). For example, in HotCRP, the `PaperTopic` table maps paper IDs to their topic IDs, but contains no other information. Since GDPRizer will return records from both the `Paper` and the `TopicArea` tables, it is unnecessary to return the mapping table records as well, and GDPRizer drops these tables from the output.

Roles. Finally, GDPRizer supports *roles*, which allow applying different customizations by data subject type. In TPC-H, for instance, `customer` rows or `supplier` rows can represent data subjects who can issue data access requests, but GDPRizer should return different information depending on whether the request originated with a customer or a supplier. The traversal must account for these roles and avoid extracting more data than is necessary for each role. The customers' primary table is `customer` and the suppliers' primary table is `supplier`. GDPRizer allows the DBA to specify separate roles, each associated with a different primary table and a set of per-role customizations, which specify a custom traversal for each role. Since roles are application-specific, GDPRizer requires manual input to specify them and their customizations.

7 EVALUATION

We prototyped GDPRizer in approximately 1,700 lines of Python. We evaluate it with a synthetic benchmark (TPC-H) and with three real web applications (Lobsters [26], HotCRP [24], and WordPress [8]). Our evaluation seeks to answer these questions:

(1) Does GDPRizer correctly identify the data to return from a data access request, and how does the method of relationship graph generation impact results? (§7.2)
(2) How many manual customizations do applications require to achieve perfect results? (§7.3)
(3) What impact do specific manual customizations have on GDPRizer's results, and when are they required? (§7.4)
(4) How does GDPRizer compare to the third-party GDPR compliance plugins available for some applications? (§7.5)

Our implementation of GDPRizer uses moz-sql-parser [3] to parse SQL queries. Since this parser can only handle some subset of SQL-92 queries, GDPRizer skips over the queries that moz-sql-parser cannot handle. This only affects a small number of queries. To detect candidate foreign keys from data, we implemented inclusion dependency detection and standard heuristics to detect foreign keys (F2, F8, and a modified F6 from Rostin et al. [36]; and a Wilcoxon test [45]). These heuristics rely heavily on thresholds. We determined GDPRizer's thresholds through experiments, selecting thresholds beyond which the number of edges passing the heuristic were stable. For out-of-range (F8), coverage (F2), the Wilcoxon test, and column name matching (modified F6), we chose 0.2, 0.8, 0.7 and 1.0 respectively. We use the same thresholds for all applications.

Accuracy measurements. We measure GDPRizer's accuracy for four applications: TPC-H, HotCRP, Lobsters, and Wordpress. None of these applications currently have native support for data access requests. Hence, they lack a *ground truth* on the data that should be returned for each data subject. We used our knowledge of the applications to determine the data that we believe a data access request should return. We studied the application's schema and for each table in its database, we manually wrote a set of "ground truth" queries. For Wordpress, which has publicly-available GDPR plugins, we compare our results to the data extracted by these plugins.

We then compare the rows that GDPRizer extracts with the rows included in these ground truths. To measure GDPRizer's accuracy, we compute *precision*, *recall* and *F1 score* relative to the ground truth. We denote precision by P, recall by R and F1 score by $F1$ and use their standard definitions:

$$P = \frac{\mathsf{tp}}{\mathsf{tp} + \mathsf{fp}}, \qquad R = \frac{\mathsf{tp}}{\mathsf{tp} + \mathsf{fn}}, \qquad F1 = \frac{2 * P * R}{P + R},$$

where tp, fp, fn, respectively are the number of true positives, false positives and false negatives in GDPRizer's results. We report the averages of per-table accuracy results, which in turn are averaged over individual users' data access requests. Table-level analysis helps us measure the performance of GDPRizer on different parts of the database and uncover any specific shortcomings.

Inflated per-table averages. Before we discuss our results, we consider a problem that could skew the precision and recall metrics when a majority of data subjects have no data in certain tables.

Suppose that GDPRizer's extraction should avoid querying a table T for any data subject with a particular role, *e.g.*, the `customer` table for suppliers. If GDPRizer queries T for data subjects with no data in T, the database will return no data, and hence GDPRizer will appear to have 100% precision. For such data subjects, GDPRizer did the right thing—extracting no data—but for the wrong reasons.

Specifically, in this example, GDPRizer extracts no data because T had no matching records, not because GDPRizer did not query it. Averaging over many such data subjects, this yields an inflated averaged precision value. A similar issue can occur with recall. To ensure our results meaningfully report GDPRizer's correctness, we exclude the data subjects with no data in T from the table's results.

7.1 Applications

We evaluate our prototype with the TPC-H benchmark and three applications: HotCRP, Lobsters and Wordpress. While TPC-H serves as a sanity check, the three applications evaluate GDPRizer's real-world performance. In this section, we describe the setup of each application: how we populate the application database, how we collect its queries, and how we establish the ground truth.

TPC-H. The TPC-H benchmark models events between suppliers and customers in a warehousing system [6]. We generated 100MB of data, with 150 customers and 10 suppliers. TPC-H's 22 SQL queries contain 62 joins, and the schema has 10 foreign-key constraints [7, Fig. 2]. GDPRizer can use the foreign-key relationships, but also successfully extracts them from the queries.

We run experiments for the customer and supplier roles. For a customer, the `customer` table is the primary table and its primary key, `c_custkey`, the primary column. The ground truth contains queries for each table except the `supplier` and `partsupp` tables, because `supplier` contains the suppliers' private information (*e.g.*, account balance), while `partsupp` contains sales information for a supplier (*e.g.*, supply cost). For suppliers, `supplier` is the primary table and its primary key, `s_suppkey`, the primary column. We exclude data from the `customer`, `orders`, and `lineitem` tables from the ground truth, since customer details and order processing details of the warehouse do not concern suppliers.

Lobsters. Lobsters is a link aggregator page, similar to HackerNews [2] or Reddit [4]. Its database has 25 tables which describe user posts, comments, votes, moderations, and so on. Lobsters comes with a sample dataset for 44 users, which we used to populate the database. We created three additional users and logged queries generated during interactions with the application. We attempted to exercise all possible actions and collected 3,960 queries. We extracted 41 edges from foreign-key constraints in the Lobsters schema and 2 additional edges from joins in the queries. For Lobsters, the `users` table is the primary table and its `id` column the primary column. We included 23 queries in the ground truth, covering 17 tables. We excluded eight tables that contain Lobsters's Ruby-on-Rails metadata rather than user data (*e.g.*, `ar_internal_metadata`, `keystores`, and `schema_migrations`).

HotCRP. HotCRP is a conference peer review application [24] and its database has 24 tables. We use an anonymized HotCRP dataset from a real conference, which contains 1,273 authors and 507 papers. We also use a sample of 251 queries. Since the schema lacks foreign-key constraints, GDPRizer uses the queries and data to build the relationship graph (Figure 2). We set `ContactInfo` as the primary table and its primary key, `contactId`, as the primary column. For the ground truth, we wrote 17 queries that extract data from 12 tables and exclude data from 12 others. Tables excluded are either application management tables such as `Settings` and `MailLog`, or mapping tables such as `PaperTopic`.

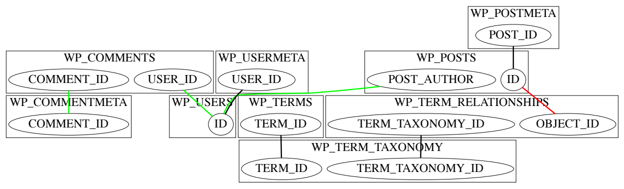


Figure 4: Without customizations, Wordpress's $R^{S,Q}$ has four disconnected components; we manually added the green edges and pruned the red edges. R^D has three edges (not shown, §7.2).

Table 1: Relationship graph statistics.

	# edges in R^S	# edges in R^Q	# edges in R^D	# edges in $R^{S,Q}$	# edges in $R^Q \cap R^D$
TPC-H	10	10	10	10	10
Lobsters	41	17	25	43	3
HotCRP	0	30	31	30	10
WP (base)	0	5	3	5	1
WP (+plugins)	0	12	120	12	2

Wordpress. WordPress is a popular blogging platform and content management system [8]. A key feature of WordPress is its plugin architecture, which allows users to add functionality to their WordPress installation. For example, the WooCommerce [12] plugin adds e-commerce functionality, allowing users to host online shops. The base WordPress database has 12 tables which describe users, comments, posts, terms, and other metadata. The WooCommerce plugin adds 27 new tables to support online shops. We investigate how GDPRizer performs on the base installation of WordPress and how it adapts to the updated database when WooCommerce is added. We generated sample data for 46 users using another WordPress plugin, FakerPress [21]. For WooCommerce, we manually generated sample data. As with Lobsters, we logged the queries generated during interactions with the WordPress site and attempted to replicate all possible actions. Overall, we collected 9,301 queries.

As the WordPress schema does not specify any explicit foreign keys, the relationship graph includes edges from queries (Figure 4) and database contents. We treat `wp_users` as the primary table and its `id` column as the primary column. For base WordPress, we included six queries over six tables in the ground truth. We excluded tables that did not contain data directly related to users (there were six such tables: `wp_links`, `wp_terms`, `wp_termmeta`, `wp_options`, `wp_term_relationships`, `wp_term_taxonomy`.) For the WooCommerce plugin's 27 new tables, we included nine new queries in our ground truth, covering nine of the new tables.

7.2 High-level Accuracy Results

We study the accuracy of GDPRizer's results for each application, based on different relationship graphs, with and without manual customization. Each application contains many data subjects and tables, and we report the the averages of per-table accuracy results, which in turn are averaged over the data subjects. A good result

Table 2: High-level results for GDPRizer by application: without any manual input GDPRizer achieves 62–73% F1 score with R^Q and 48–70% both with R^D and $R^Q \cap R^D$; with manual input, it achieves 100% F1 score, except for HotCRP. Values reported here averaged over per-table values, which in turn are averages over individual data subjects. Lobsters uses $R^{S,Q}$ and $R^{S,D}$.

	Pre-customization							Post-customization	
	Using join queries (R^Q)			Using database contents (R^D)			$R^Q \cap R^D$	R^Q	R^D
	Precision (avg.)	Recall (avg.)	F1 (avg.)	Prec. (avg.)	Rec. (avg.)	F1 (avg.)	F1 (avg.)	F1 (avg.)	F1 (avg.)
TPC-H (customer)	0.68	1	0.7	0.68	1	0.7	0.7	1	1
TPC-H (supplier)	0.62	1	0.62	0.62	1	0.62	0.62	1	1
Lobsters (+ R^S)	0.70	0.99	0.73	1	0.48	0.48	0.48	1	1
HotCRP	0.76	0.76	0.64	0.62	0.88	0.58	0.63	0.96	0.93
WP (base)	1	0.67	0.67	1	0.58	0.58	0.58	1	1
WP (w/ plugins)	1	0.66	0.66	1	0.64	0.64	0.64	1	1

for GDPRizer would show high F1 score both with and without manual customization.

Table 1 shows the numbers of edges in each type of relationship graph. Except for the synthetic TPC-H benchmark, all applications' R^Q and R^D contain edges not otherwise discovered. Only Lobsters has explicit foreign keys (*i.e.*, R^S is not empty), of which R^Q captures 15 and R^D captures 4 edges. But at least 26 foreign keys are missing in Lobsters's R^Q and R^D, which illustrates that it is helpful for GDPRizer to use explicit foreign keys when available. R^D in WordPress with plugins is large (120 edges) as GDPRizer's heuristics suggest many false positive foreign keys between WooCommerce's customer IDs and WordPress user ID columns. Finally, $R^Q \cap R^D$ is generally small.

Table 2 measures the correctness of the data GDPRizer returns for the different relationship graphs. Without any manual customizations, GDPRizer achieves at least 62% F1 score across applications with R^Q and 48% with R^D. Individual precision or recall metrics reach 100% for some applications, but no application sees both perfect precision and recall. Since privacy compliance is all-or-nothing, this would be insufficient in a practical setting. The final columns in Figure 2 illustrate that, with inputs from the DBA, GDPRizer achieves perfect F1 score for all applications except HotCRP, whose score is 96% with R^Q and 93% with R^D. With R^Q, this imperfect score happens because there are two paths into HotCRP's `TopicArea` table. GDPRizer ignores the longer path, and hence under-extracts from `TopicArea`. However, `TopicArea` only contains public information (the paper topic categories), so a DBA could plausibly annotate the table to indicate that GDPRizer should always return it in its entirety. With R^D, the same problem persists with the `TopicArea` table; moreover, GDPRizer uses the `ActionLog` table to extract papers written by a data subject, but over-extracts for data subjects who are PC members, reducing the precision (and F1 score).

These results show that R^Q achieves slightly better results than R^D, suggesting that GDPRizer prefer R^Q when both are available. However, if no query log is available, R^D is an adequate substitute. Finally, using $R^Q \cap R^D$ provides no benefit over using R^D directly.

7.3 Manual Customizations Needed

While minimal input from the DBA is desirable, our experiments show that some manual input is usually necessary. We now investigate the inputs required for our applications. Given the relationship graph and suggestions from GDPRizer, we believe that any DBA with some background knowledge of the database should be able to apply these customizations. Table 3 summarizes the customizations for each application and relationship graph type.

Across applications, all the relationship graphs require some amount of manual customization. The specifics vary between relationship graphs, however: the combined relationship graph, $R^Q \cap R^D$, consistently requires the largest number of customizations. This makes sense, as the combined graph has only a few edges in general (see Table 1). Comparing R^Q and R^D produces a more nuanced picture. In HotCRP, R^Q requires more annotations (31) than R^D (29), while in Lobsters, $R^{S,D}$ requires more annotations (26) than $R^{S,Q}$ (16). Moreover, R^D consistently requires a larger number of *edge addition* customizations, which are more challenging for a DBA than column pruning annotations. This suggests that, in general, using R^Q is preferable (if it is available). R^D is still useful, but potentially adds overhead as GDPRizer must do work proportional to the size of the database to build R^D, and it requires more sophisticated input from the DBA, which suggests it is best used when no other information is available. Next, we give examples of the customizations required by different applications.

Columns added. Adding "virtual" columns helps GDPRizer deal with implied or conditional relationships, but is rarely required. Only HotCRP with R^Q requires this customization. This is because HotCRP stores co-authorship information in the `PaperConflict` table, since papers can have an arbitrary number of co-authors. (This makes sense because each co-author has a conflict of interest with their own paper.) Specifically, the value in the `conflictType` column of the `PaperConflict` determines if the row represents a co-authorship or a different type of conflict.

To handle this in GDPRizer, the DBA defines a virtual column. This adds a new column, named `v_author`, to the `Paper` table, which is a direct foreign key into the `ContactInfo` table (*i.e.*, `v_author` stores the contact IDs of the co-authors of a paper). This results in a new relationship graph vertex, `v_author`, which captures the author relationship and lets GDPRizer traverse the relationship graph in its usual way. This customization is not required with R^D, because that graph provides an auxiliary mapping between `ContactInfo` and `Paper` via the `ActionLog` table.

Edges pruned. Edge pruning is the most common customization, and affects the most tables and columns. Ideally, the reasons for pruning should be easily evident to a DBA. In our example applications, pruning annotations fall into three categories.

Table 3: **Number of manual customizations needed to achieve perfect accuracy. In TPC-H, $\mathtt{R^S}$, $\mathtt{R^Q}$, and $\mathtt{R^D}$ are identical; in Lobsters, GDPRizer uses the available explicit foreign keys, and hence all results include $\mathtt{R^S}$.**

	TPC-H (cust)	TPC-H (supp)	HotCRP			Lobsters			WP (base)			WP (w/ plugins)
	$R^Q/R^S/R^D$	$R^Q/R^S/R^D$	R^Q	R^D	$R^Q \cap R^D$	$R^{S,Q}$	$R^{S,D}$	$R^{S,Q} \cap R^{S,D}$	R^Q	R^D	$R^Q \cap R^D$	R^Q
# cols added	0	0	1	0	1	0	0	0	0	0	0	0
# cols filtered	0	0	18	18	18	0	0	0	0	0	0	0
# edges added	0	0	2	5	11	1	0	2	3	4	5	9
# edge pruning annotations	4	7	10	6	4	15	26	15	1	0	0	3
Total # customizations	**4**	**7**	**31**	**29**	**34**	**16**	**26**	**17**	**4**	**4**	**5**	**12**

Tables without user relevance. GDPRizer should not extract application data that is irrelevant to users. A DBA identifies internal tables and annotates all columns in these tables with incident edges for pruning. For example, in Lobsters, three columns in management tables need annotating; in HotCRP, we annotate four columns; and in WordPress, we prune one column in the base setup and another column that represents global product permissions with the WooCommerce plugin. These prunings are unnecessary if the DBA decides to return data from these tables; in our experiments, we considered data in management tables irrelevant and removed their data from the ground truth, so pruning made sense.

Other data subjects' information. Privacy laws require avoiding to return personal information of other individuals when satisfying a data access request, meaning that such information cannot be returned by GDPRizer. For example, if the data subject is a customer in TPC-H, we prune the edges into the the `supplier` and the `partsupp` tables, whereas for a supplier, we prune edges into the `customer`, `lineitem` and `order` tables (see §6).

Avoiding over-extraction. The third kind of edge pruning requires the DBA to identify individual edges that might extract incorrect data. For example, in Lobsters, we prune the `story_id` column from six tables to stop GDPRizer from retrieving information about stories that a data subject might have acted upon but does not own. For instance, we prune `votes.story_id` to avoid information on stories that a data subject voted on but has not written (although GDPRizer still extracts the vote records themselves). This pruning also occurs in HotCRP. With $\mathtt{R^Q}$, we prune `shepherdContactId` from `Paper` to stop GDPRizer from extracting details of papers which a data subject shepherded but did not write. We prune seven columns of this type in HotCRP with $\mathtt{R^Q}$ and six with $\mathtt{R^D}$. These pruning annotations require a DBA to consider and inspect GDPRizer's output, and to potentially iterate until the output is correct.

Edge additions. Edge additions provide GDPRizer with missing relationships in the relationship graph. This customization is crucial if an application's schema lacks foreign keys and $\mathtt{R^Q}$ and $\mathtt{R^D}$ both provide incomplete information. $\mathtt{R^Q}$ may lack edges because application developers never joined the columns, *e.g.*, because they compute joins in application code. This sometimes happens as developers seek to improve scalability by avoiding joins that require locking multiple tables. $\mathtt{R^D}$ sometimes lacks edges because the data is inconsistent (violating the inclusion dependency requirement) or because the edges fail to pass the heuristics' thresholds. For example, Lobsters's `mod_nodes.moderator_user_id` and `users.id` form an actual foreign key, but the pair fails the coverage threshold as only a small number of users are moderators.

In our applications, between one and nine edges needed adding with $\mathtt{R^Q}$, and four to 14 edges needed adding with $\mathtt{R^D}$. In HotCRP, we add an edge between `ContactInfo.contactId` and `ReviewRating.contactId` to capture the review ratings a user contributed. With $\mathtt{R^Q}$ (but not $\mathtt{R^D}$), we also add an edge between the `topicId` columns in `Paper` and `TopicArea` to capture topics. In Lobsters, $\mathtt{R^D}$ lacks any of the 13 edges on the primary `users.id` column. But these edges are a part of $\mathtt{R^S}$ and therefore of $\mathtt{R^{S,D}}$, so no customizations are needed for these edges. With $\mathtt{R^{S,Q}}$, only the edge between `users.id` and `messages.author_user_id` needs adding, ensuring that private messages a data subject wrote are captured. Lobsters's schema lacks this edge because it avoids having two foreign keys between the same pair of tables (`users` and `messages`), but the relationship nevertheless exists. Wordpress has the least well-connected relationship graphs, likely due to client-side joins ($\mathtt{R^Q}$) and its column naming conventions (in $\mathtt{R^D}$, the name matching heuristic filters six of the nine edges that pass other heuristics). $\mathtt{R^Q}$ for the base Wordpress setup contains four disconnected components (Figure 4) while $\mathtt{R^D}$ has seven disconnected components. To connect the components, we manually add the three edges to $\mathtt{R^Q}$ and four edges to $\mathtt{R^D}$. Edges `users.ID` ↔ `posts.post_author` and `comments.ID` ↔ `commentmeta.comment_id` need adding to both. With the WooCommerce plugin and $\mathtt{R^Q}$, we add six edges in addition to the above. These edges connect disconnected tables that have user IDs to the `wp_users` table. For example, we connect `wp_wc_payment_tokens.user_id` to `wp_users.id` to capture a user's payment tokens. GDPRizer recognizes the disconnected components and helps identify the possible connecting edges based on inclusion dependencies. With the plugin, $\mathtt{R^D}$ includes these edges, but also includes dozens of false-positive edges between WooCommerce's customer IDs and the WordPress user ID columns, which the DBA must remove.

Output filtering. GDPRizer's output filtering removes columns that contain sensitive data. How often this customization is required depends on the application; in the applications we looked at, only HotCRP requires filtering. For example, we filter 18 (out of 42) columns of the `PaperReview` table. These columns contain reviewer-specific information such as the reviewer's user ID, the reviewer's qualifications, and their private comments to the program committee. Relative to the total number of columns in the database (200 in HotCRP), filtering affects only a small number of columns.

7.4 Impact of Customizations

We now evaluate how individual customizations affect GDPRizer when using R^Q. Generally, output filtering and edge pruning improve precision, while vertex and edge additions improve recall. Most applications benefit most from a single type of customization, while the others improve accuracy in smaller, but still important, ways. We present the results as grouped bar charts (Figures 5a–5c). The first bar in a group represents data extraction using the relationship graph only, and each successive bar indicates the impact of an additional customization. Our graphs show the mean of per-table averages over all data subjects.

TPC-H. For the customer role, all tables have 100% recall even without customization, and all but four tables (`part`, `supplier`, `partsupp`, and `lineitem`) have 100% precision. After we prune the edges for tables that are irrelevant to customers, precision increases to 100% (Figure 5a). The supplier role shows similar results, except that pruning edges to three tables with customer-related data increases precision to 100% (Figure 5a). With R^D, the results are the same (cf. Table 2), since R^D and R^Q are identical in TPC-H.

Lobsters. Figure 5b shows the results for Lobsters with R^Q. After pruning, GDPRizer achieves 100% precision on all the tables and 100% recall on all but the `messages` table, for which recall is 92%. This is due to a missing edge between `users.id` and `messages.author_user_id`; once we add the missing edge, GDPRizer's recall improves to 100%. With R^D, pre-customization, GDPRizer achieves only 48% F1 score. This is worse than the pre-customization F1 score of 70% with $R^{S,Q}$. This is because the primary `users.id` column is completely disconnected from the rest of the graph in R^D, and hence GDPRizer extracts nothing from any table except `users`. However, once we combine R^S with R^D and do some pruning, GDPRizer's F1 score improves to 100% (Table 2).

HotCRP. Figure 5c shows how successive customizations affect precision, recall and F1 score for HotCRP with $R^{S,Q}$. Without customizations, precision and recall are both around 76% and the F1 score is 64%. The main reason for the low precision and recall is that GDPRizer fails to identify the papers authored by a data subject, and under-extracts on their information. Fixing recall requires a virtual column representing HotCRP's co-authorship (see §7.3). The reason for the low precision is that GDPRizer traverses the edge between `PaperConflict.paperId` and `Paper.paperId` and erroneously extracts papers that the data subject is conflicted with. As with recall, this problem propagates to other paper-related tables. The pruning and filtering described earlier improve their precision. After these customizations, `ReviewRating` (0%) and `TopicArea` (6%) are the only tables with imperfect recall. Adding a missing edge between the `contactId` columns of `ContactInfo` and `ReviewRating` helps GDPRizer retrieve the review ratings. The recall of `TopicArea` remains imperfect (6%) even after we add the edge between the `topicId` columns of `PaperTopic` and `TopicArea`. The is because the relationship graph of HotCRP has two paths into `TopicArea`, one of which represents topic areas for papers that a data subject submitted, while the other represents areas of review interest of a data subject (meaningful only for program committee members). The former path is longer than the latter and GDPRizer ignores it, missing topic areas for submitted papers that aren't also the data subject's preferred review areas.

With R^D, pre-customization precision, recall and F1 score respectively are 62%, 88%, and 58% (Table 2). The main problem is again that GDPRizer extracts conflicted papers' information via `PaperConflict.paperId`, reducing precision. Low precision (0%) and recall (79%) also occur in `TopicInterest`, as GDPRizer uses edge `PaperComment.contactId` ↔ `TopicInterest.contactId` to extract topic interests of the data subject's papers, even though these may be private PC comments. After pruning edges from `PaperConflict` and `PaperComment`, and adding an edge between the `contactId` columns of `ContactInfo` and `TopicInterest`, precision and recall improve. However, recall of `PaperReview` (38%) and `TopicArea` (6%) still remain imperfect and the final F1 score achieved with R^D is 93%.

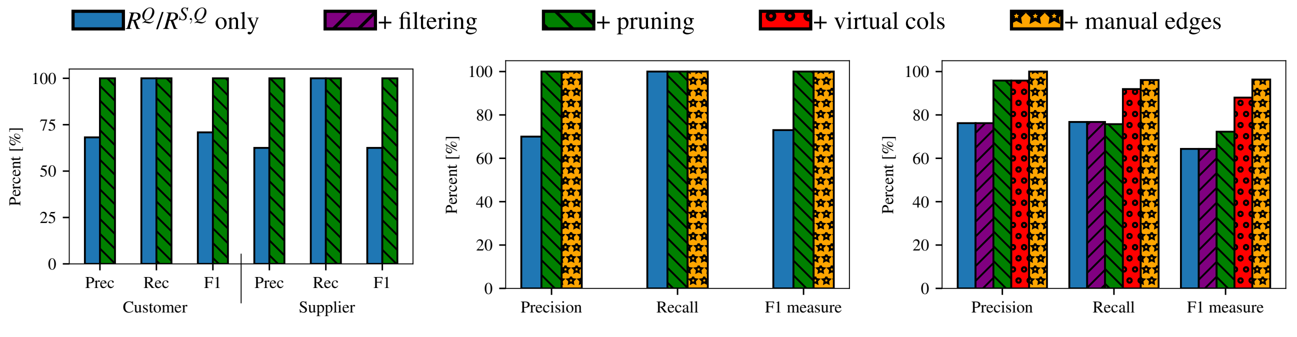
WordPress. Figure 6 shows that for the base installation of WordPress, GDPRizer with R^Q achieves perfect precision for all tables even before customization. However, it only achieves perfect recall on two tables (`wp_users` and `wp_usermeta`). For other tables with user data, such as `wp_comments` and `wp_posts`, recall is zero because `wp_users` and `wp_usermeta` are in a component that is disconnected from the rest of the relationship graph (Figure 4). Manually adding the missing edges to connect the components improves the recall to 100%, however. We see similar results for WordPress with the WooCommerce plugin: the original relationship graph $R^{S,Q}$ has seven disconnected components, but after edge additions, recall improves to 100% for all tables. We see similar results both for the base installation and the WooCommerce plugin when starting with R^D (Table 2). We conclude from this experiment that edge additions are crucial for GDPRizer to support applications with disconnected components in their relationship graphs.

7.5 Comparison with GDPR Compliance plugins

WordPress's extensive collection of third-party plugins includes several plugins which are designed to aid administrators with GDPR compliance. Some of these plugins also support data access requests, and we compare GDPRizer to three existing GDPR plugins: GDPR Compliance and Cookie Consent [42], The GDPR Framework by Data443 [17], and WP GDPR Compliance [32]. These plugins are quite popular: the first two have over 30k installations, and WP GDPR Compliance has 200k+ installations. We assess whether these plugins capture the information specified in our ground truth.

The results for the base installation are in Table 4 and those with the WooCommerce plugin in Table 5. We find that GDPRizer successfully identifies user information from all the tables in the ground truth, while the existing plugins miss some of the tables. For example, all the plugins fail to extract information from `wp_posts`. This may be because the plugins are designed to serve data access requests from users who have interacted with the WordPress site, but do not have accounts on it (*e.g.*, casual commenters). However, our results illustrate that installing such a plugin may be insufficient to achieve true compliance; GDPRizer, working at the level of the database schema, offers a broader set of options to the DBA.

Finally, with the WooCommerce plugin, the plugins once again overlook some tables. This might be due to an oversight on the part of the plugin developers, or due to a different understanding of what information must be returned to users to comply with the



(a) TPC-H. (b) Lobsters. (c) HotCRP.

Figure 5: Precision, recall and F1 score of GDPRizer with successive customizations over R^Q ($R^{S,Q}$ in Lobsters). Edge pruning improves precision the most across applications; in Lobsters, manual edge addition is necessary to reach 100% recall on `messages`, while a virtual column handling co-authorship is essential for recall in HotCRP.

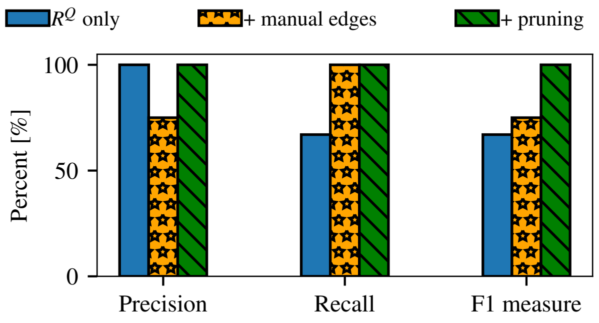


Figure 6: Precision of GDPRizer for the base installation of Wordpress. Manual edge additions improve recall but make precision worse, but the overall F1 score still improves. Further edge pruning is required to achieve 100% F1 score.

Table 4: Comparison of GDPRizer with existing GDPR plugins for WordPress. For a table T, green boxes represent complete extraction of data while red represent no extraction.

	[42]	[17]	[32]	GDPRizer
`wp_user`	✓	🟥	✓	✓
`wp_usermeta`	✓	✓	🟥	✓
`wp_posts`	🟥	🟥	🟥	✓
`wp_postmeta`	🟥	🟥	🟥	✓
`wp_comments`	✓	✓	✓	✓
`wp_commentmeta`	✓	✓	🟥	✓

GDPR. For example, some of the tables included in our ground truth, such as `download_log` and `api_keys`, contain backend information meant for the application rather than end-users, even though this information is tied to a data subject. Under the GDPR, this information (*e.g.*, download events) must nevertheless be returned because it is identifiably associated with a data subject—a nuance that may have escaped the plugin developers, but which puts the plugins' users at risk of violating the GDPR.

Table 5: Comparison of GDPRizer with existing GDPR plugins for WordPress with WooCommerce. For a table T, green boxes represent complete extraction of data, red represents no extraction, and yellow represents partial extraction.

	[42]	[17]	[32]	GDPRizer
`customer_information`	✓	✓	~	✓
`order_information`	✓	✓	~	✓
`order_to_product`	✓	✓	🟥	✓
`order_to_coupon`	🟥	🟥	🟥	✓
`download_log`	🟥	🟥	🟥	✓
`webhooks`	🟥	🟥	🟥	✓
`api_keys`	🟥	🟥	🟥	✓
`download_permissions`	🟥	🟥	🟥	✓
`payment_tokens`	🟥	🟥	🟥	✓

8 CONCLUSIONS

Our goal in building GDPRizer was to understand the trade-offs between automation and manual effort in retrofitting compliance onto applications' legacy databases. We started out with the basic components that we could expect for real-world applications—a schema, a query log, and database contents—and studied general-purpose approaches to compliance using these inputs. However, each application we studied required a small but specific amount of manual customization. Although this is much less effort than a completely manual solution, it still requires human intervention. While further work is needed, we believe that it unlikely that a fully-automated solution exists. Yet, even partially automated solutions will go a long way in making the transition smoother for legacy systems.

ACKNOWLEDGMENTS

We thank Andrew Crotty, Connor Luckett, Kinan Dak Albab, and Ugur Çetintemel for their feedback on drafts of this paper. This work was supported by NSF award CNS-2045170 and by a Google Research Scholar award.

REFERENCES

[1] The Web framework for perfectionists with deadlines | Django. URL https://www.djangoproject.com/. Accessed 13 Dec. 2021.

[2] The Hacker News. URL https://thehackernews.com/. Accessed 13 Dec. 2021.

[3] moz-sql-parser - SQL query parser. URL https://github.com/mozilla/moz-sql-parser. Accessed 13 Dec. 2021.

[4] Reddit. URL https://www.reddit.com/. Accessed 13 Dec. 2021.

[5] Ruby on Rails. URL https://rubyonrails.org/. Accessed 13 Dec. 2021.

[6] The TPC-H decision support benchmark, . URL http://www.tpc.org/tpch/default5.asp. Accessed 13 Dec. 2021.

[7] The TPC-H decision support benchmark documentation, . URL http://www.tpc.org/tpc_documents_current_versions/pdf/tpc-h_v2.17.1.pdf. Accessed 13 Dec. 2021.

[8] WordPress.com: Create a Free Website or Blog. URL https://wordpress.com/. Accessed 13 Dec. 2021.

[9] 18 Biggest GDPR Fines of 2020 & 2021 (So Far) | Updated 2021, May 2021. URL https://www.tessian.com/blog/biggest-gdpr-fines-2020/. Accessed 13 Dec. 2021.

[10] Ziawasch Abedjan, Lukasz Golab, and Felix Naumann. Profiling relational data: A survey. *The VLDB Journal*, 24(4):557–581, August 2015. ISSN 1066-8888. doi: 10.1007/s00778-015-0389-y. URL https://doi.org/10.1007/s00778-015-0389-y.

[11] aermin. ghchat (react version). GitHub. URL https://github.com/aermin/ghChat. Accessed 13 Dec. 2021.

[12] Automattic. WooCommerce. URL https://wordpress.org/plugins/woocommerce/. Accessed 13 Dec. 2021.

[13] Jana Bauckmann, Ulf Leser, Felix Naumann, and Veronique Tietz. Efficiently detecting inclusion dependencies. In *Proceedings of the IEEE 23rd International Conference on Data Engineering (ICDE)*, pages 1448–1450, 2007. doi: 10.1109/ICDE.2007.369032.

[14] California Legislature. The California Consumer Privacy Act of 2018, June 2018. URL https://leginfo.legislature.ca.gov/faces/billTextClient.xhtml?bill_id=201720180AB375. Accessed 13 Dec. 2021.

[15] Adhityaa Chandrasekar. Commento. GitHub. URL https://github.com/adtac/commento. Accessed 13 Dec. 2021.

[16] Zhimin Chen, Vivek Narasayya, and Surajit Chaudhuri. Fast foreign-key detection in microsoft sql server powerpivot for excel. *Proceedings of the VLDB Endowment*, 7(13):1417–1428, August 2014. ISSN 2150-8097. doi: 10.14778/2733004.2733014. URL https://doi.org/10.14778/2733004.2733014.

[17] Data443. The GDPR Framework By Data443. URL https://wordpress.org/plugins/gdpr-framework/. Accessed 13 Dec. 2021.

[18] Dong Deng, Raul Castro Fernandez, Ziawasch Abedjan, Sibo Wang, Michael Stonebraker, Ahmed K Elmagarmid, Samuel Madden, Mourad Ouzzani, and Nan Tang. The data civilizer system. In *Proceedings of the biennial Conference on Innovative Data Systems Research (CIDR)*, 2017. URL http://cidrdb.org/cidr2017/papers/p44-deng-cidr17.pdf.

[19] European Union. Regulation (EU) 2016/679 of the European Parliament and of the Council of 27 April 2016 on the protection of natural persons with regard to the processing of personal data and on the free movement of such data, and repealing Directive 95/46/EC (General Data Protection Regulation). *Official Journal of the European Union*, L119:1–88, May 2016. URL http://eur-lex.europa.eu/legal-content/EN/TXT/?uri=OJ:L:2016:119:TOC.

[20] Thailand Government Gazette. Personal data protection act. URL https://thainetizen.org/wp-content/uploads/2019/11/thailand-personal-data-protection-act-2019-en.pdf. Unofficial English translation. Accessed 13 Dec. 2021.

[21] Gustavo Bordoni. FakerPress. URL https://wordpress.org/plugins/fakerpress/. Accessed 13 Dec. 2021.

[22] PRS Legislative Research India. The personal data protection bill, 2019. URL https://www.prsindia.org/billtrack/personal-data-protection-bill-2019. Accessed 13 Dec. 2021.

[23] Zsolt István, Soujanya Ponnapalli, and Vijay Chidambaram. Software-defined data protection: Low overhead policy compliance at the storage layer is within reach! *Proceedings of the VLDB Endowment*, 14(7):1167–1174, March 2021. ISSN 2150-8097. doi: 10.14778/3450980.3450986. URL https://doi.org/10.14778/3450980.3450986.

[24] Eddie Kohler. Hotcrp conference review software. URL https://github.com/kohler/hotcrp. Accessed 13 Dec. 2021.

[25] Tim Kraska, Michael Stonebraker, Michael Brodie, Sacha Servan-Schreiber, and Daniel Weitzner. Schengendb: A data protection database proposal. In Vijay Gadepally, Timothy Mattson, Michael Stonebraker, Fusheng Wang, Gang Luo, Yanhui Laing, and Alevtina Dubovitskaya, editors, *Heterogeneous Data Management, Polystores, and Analytics for Healthcare*, pages 24–38, 2019. ISBN 978-3-030-33752-0.

[26] Lobsters Developers. Lobsters news aggregator, March 2018. URL https://lobste.rs. Accessed 13 Dec. 2021.

[27] Stéphane Lopes, Jean-Marc Petit, and Farouk Toumani. Discovering interesting inclusion dependencies: Application to logical database tuning. *Inf. Syst.*, 27(1): 1–19, March 2002. ISSN 0306-4379. doi: 10.1016/S0306-4379(01)00027-8. URL https://doi.org/10.1016/S0306-4379(01)00027-8.

[28] Connor Luckett, Andrew Crotty, Alex Galakatos, and Ugur Cetintemel. Odlaw: A tool for retroactive gdpr compliance. URL http://cs.brown.edu/people/acrotty/pubs/918400c709.pdf.

[29] Fabien De Marchi and Jean-Marc Petit. Zigzag: A new algorithm for mining large inclusion dependencies in databases. In *Proceedings of the 3rd IEEE International Conference on Data Mining (ICDM)*, page 27, USA, 2003. IEEE Computer Society. ISBN 0769519784.

[30] Sudharsanan Muralidharan. Socify: open source social network using ruby on rails. GitHub. URL https://github.com/scaffeinate/socify. Accessed 13 Dec. 2021.

[31] Natnal Congress of Brazil. Lei geral de proteção de dados [brazilian general data protection law]. URL https://iapp.org/media/pdf/resource_center/Brazilian_General_Data_Protection_Law.pdf. English translation by Ronaldo Lemos, Daniel Douek, Sofia Lima Franco, Ramon Alberto dos Santos and Natalia Langenegger. Accessed 13 Dec. 2021.

[32] Van Ons. WP GDPR Compliance. URL https://wordpress.org/plugins/wp-gdpr-compliance/. Accessed 13 Dec. 2021.

[33] OpenCart: open source e-commerce platform. Opencart. GitHub. URL https://github.com/opencart/opencart. Accessed 13 Dec. 2021.

[34] Thorsten Papenbrock, Sebastian Kruse, Jorge-Arnulfo Quiané-Ruiz, and Felix Naumann. Divide & conquer-based inclusion dependency discovery. *Proceedings of the VLDB Endowment*, 8(7):774–785, February 2015. ISSN 2150-8097. doi: 10.14778/2752939.2752946. URL https://doi.org/10.14778/2752939.2752946.

[35] PrestaShop SA. Prestashop: open-source e-commerce. GitHub. URL https://github.com/PrestaShop/PrestaShop. Accessed 13 Dec. 2021.

[36] Alexandra Rostin, Oliver Albrecht, Jana Bauckmann, Felix Naumann, and Ulf Leser. A machine learning approach to foreign key discovery. In *Proceedings of the 12th International Workshop on the Web and Databases (WebDB)*, 2009. URL https://hpi.de/fileadmin/user_upload/fachgebiete/naumann/publications/2009/WebDB09_crc.pdf.

[37] schnack! schnack.js. GitHub. URL https://github.com/schn4ck/schnack. Accessed 13 Dec. 2021.

[38] Malte Schwarzkopf, Eddie Kohler, M Frans Kaashoek, and Robert Morris. Gdpr compliance by construction. In *Proceedings of the 2019 Workshop on Heterogeneous Data Management, Polystores, and Analytics for Healthcare (Poly)*, pages 39–53. Springer, August 2019.

[39] Aashaka Shah, Vinay Banakar, Supreeth Shastri, Melissa Wasserman, and Vijay Chidambaram. Analyzing the impact of gdpr on storage systems. In *Proceedings of the 11th USENIX Conference on Hot Topics in Storage and File Systems*, HotStorage'19, page 4, USA, 2019. USENIX Association.

[40] Faiyaz Shaikh. React-instagram-clone-2.0. GitHub. URL https://github.com/yTakkar/React-Instagram-Clone-2.0. Accessed 13 Dec. 2021.

[41] Supreeth Shastri, Vinay Banakar, Melissa Wasserman, Arun Kumar, and Vijay Chidambaram. Understanding and benchmarking the impact of gdpr on database systems. *Proceedings of the VLDB Endowment*, 13(7):1064–1077, March 2020. ISSN 2150-8097. doi: 3384345.3384354. URL https://doi.org/10.14778/3384345.3384354.

[42] StylemixThemes. GDPR Compliance & Cookie Consent. URL https://wordpress.org/plugins/gdpr-compliance-cookie-consent/. Accessed 13 Dec. 2021.

[43] Griffin Thorne. Gdpr meets its match ... in china. China Law Blog, July 2019. URL https://www.chinalawblog.com/2019/07/gdpr-meets-its-match-in-china.html. Accessed 13 Dec. 2021.

[44] Virgina Legislative Information System. 2021 special session, h2307: Consumer data protection act. URL https://lis.virginia.gov/cgi-bin/legp604.exe?212+ful+CHAP0035. Accessed 13 Dec. 2021.

[45] Meihui Zhang, Marios Hadjieleftheriou, Beng Chin Ooi, Cecilia M. Procopiuc, and Divesh Srivastava. On multi-column foreign key discovery. *Proceedings of the VLDB Endowment*, 3(1–2):805–814, September 2010. ISSN 2150-8097. doi: 10.14778/1920841.1920944. URL https://doi.org/10.14778/1920841.1920944.