Project Report:
GDPR-Compliance in Web Applications by Construction

Kinan Dak Albab

Abstract

The General Data Protection Regulation (GDPR) guarantees several rights to data subjects using a web application, including rights to access, deletion, data portability, and restriction of processing. Web applications can only guarantee GDPR-compliance by providing their users with functionality that allows them to exercise these rights. Implementing such functionality is tedious and error-prone. Additionally, this functionality may come at a performance cost.

This report proposes a novel backend system for web applications, that guarantees GDPR-compliance by construction. The backend system is organized at the level of a user shard, which is a physical unit of data that stores exactly a single user's data. We demonstrate how this re-organization provides efficient functionality for accessing and deleting a user's data, and discuss how the performance penalty it induces on common application queries can be minimized.

1 Introduction

Ensuring that a web application (or any piece of software) is GDPR compliant is a tedious task: GDPR compliance touches all aspects of the application. GDPR poses specific requirements on the storage of data, as well as how it is processed. Furthermore, it requires that applications provide users with specific functionality, such as a mean to retrieve all of that users data.

The interplay between GDPR compliance and modern web applications design principles, as expressed in frameworks such as Django, provides opportunities for interesting ways to achieve GDPR compliance by construction. While web applications design focus on modularity and separation of concerns (e.g. model-view-controller architecture), ad-hoc GDPR compliance may result in sprinkling the logic for such compliance throughout these modules, making it hard to maintain and design such applications, and to validate whether that logic is indeed sufficient for compliance.

Furthermore, it is clear that GDPR-compliance affects performance in traditionally-designed systems and applications. A traditional web application can implement GDPR-compliance functionality (such as right to access or deletion) as it would any other functionality. However, such implementation will incur significant overheads, since acquiring or deleting all the data of a user in a traditional setup involves many expensive joins and filters. Furthermore, the core functionality of the web application may also be affected, as it has to be modified in various ways to ensure processing a user's data is done with proper consent and permissions. On the other hand, minimally modifying existing database and backend systems to provide compliance is demonstrated to introduce a significant overhead to host applications [2].

We believe efficient compliance-by-construction can only be achieved with a fundamental re-design of databases and backend systems. We describe a novel design that provides such compliance out of the box, and implement a prototype backend system achieving this design. Our system provides a familiar SQL-interface to web applications for schema creation, data insertion, and data processing.

Host applications interact with the backend system via traditional SQL statements, written against the original schema that the host application specified. This original schema is used as a contract between our system and the host application. Internally, the system stores and organizes data in a radically different way: instead of storing all the data in a single database with the original schema, the system maintains a collection of physical shards (i.e. mini databases), that store portions of the overall data per user. A shard contains an automatically deduced subset of tables from the original schema, which are populated only with rows that correspond to the user of the shard.

This sharded design allows our system to provide efficient mechanisms for data access and deletion, both implemented as constant time operators independent from the number of the users, the size of the database, or complexity of the schema. Retrieving the data of a particular user corresponds to simply dumping that user's shard, and deletion requires only deleting that shard. Both functionalities can be carried out without the need for any joins or filtering.

2 User-Sharded Database Design

Our backend system stores and manages data internally sharded by user. Instead, of a single monolithic database, our system manages several mini-databases, with each such single database dedicated to a single user, and storing the entirety of that user's data.

This sharding is performed automatically. The host application specifies a logical unsharded SQL schema, which is analyzed by our system, and transformed into a user-sharded schema. All subsequent insertions, deletions, and queries made by the host application are automatically transformed into equivalent statements compatible with the underlying user-sharded organization.

Our system maintains an in-memory cache of user to shard mapping, which allows our system to identify the all existing user shards and the users they correspond to. This cached mapping is automatically restored whenever the system is restarted, and is updated whenever it is affected by an executed update or delete statement. This cache is used to efficiently identify which shard a transformed statement should be executed against.

Overall, the design consists of the following components:

Schema Rewriter: The schema rewriter analyzes the logical unsharded schema of a table, one at a time, and determines what its user-sharded schema looks like. The main task of the rewriter is to determine whether row in a table represents a user of the application, data associated with some user(s), or data unrelated to any user.

The notion of a user is a semantic property of the host application. A table of users in one application may look significantly different than one from another application. Furthermore, within a single application, there may exist several types of users, each stored in a different table. For example, an application for course assignment submissions includes both "students" and "instructors", each commonly stored in a separate table.

Our system can only discover this semantic notion effectively with inputs from the host application. We extend SQL with a "PII" annotation, and rely on application developers to annotate relevant columns in user-describing tables with that annotation. Our system considers tables with such annotations to be user tables.

Our schema rewriter judges whether a table contains user related data by analyzing foreign keys out of that table. If a table contains a direct foreign key to a user defining table, or a foreign key to a different user-related table, it is considered to be user-related. The second case implicitly defines a relationship between rows in this table and users, via transitive foreign keys.

Data is commonly shared or related to different users, such as direct messages between users of a social media application, or transactions in a banking application. Tables containing such data include multiple foreign keys, each relating the data with one of these users. Our schema rewriter needs to make a judgment regarding which user shard such data ultimately resides in.

Although shards are an implementation detail internal to our system, they actually codify deeper semantic relationships between data and users. In particular, a shard is meant to include exactly all the data belonging to its user, such that retrieving this shard is equivalent to retrieving all that user's data, and deleting that shard is equivalent to removing that user from the application completely.

In common cases, our schema rewriter can automatically deduce these semantic relationship, and make sharding judgment based on them. However, these relationships are ambigious for data shared between various users. The exact semantic relationships in such a case are only known to the host application developer. Thus, we require that developers annotate the schema of such shared data with an "Owner" annotation. Our rewriter throws a runtime error whenever an ambigious relationship is encountered that was not resolved by the developer using this annotation.

Our rewriter considers tables annotated with a single Owner annotation to contain data owned exclusively by the annotated user, even if it is related to multiple users via foreign keys. Alternatively, having multiple owner annotations codifies having data whose ownership is shared among all annotated users.

When a table gets sharded via our rewriter, no physical database operations are exeucted. Instead, the sharded schema and judgments are saved within our system's state. When data for a new user is inserted, our system uses this stored schema to create a new shard with the relevant sharded tables for that user.

Insert Statements Rewriter: SQL insert statement specify a sequence of rows to insert into a given table. Our insert statements rewriter can identify whether that table is sharded or not, by looking it up in the system's state.

For an unsharded table, the insertion is unmodified, and is executed within the default shard that includes user-unaffliated data. However, if the table represents a user, the insertion also triggers a shard creation for that user, which is initially empty.

On the other hand, inserting data into a sharded table requires rewriting the statement. First, the owners of the data are identified according to the state produced by the schema rewriter. Second, the values specified in the insert statement are inspected to find the concrete value corresponding to each owner. Finally, for every user owner, the data is inserted into that user's shard.

Queries Rewriter: Host applications provide queries written against the unsharded original schema. Therefore, a single query may yield data from various users, and thus may span several shards. For example, querying a table without any filters returns all the data in that logical table, which includes the data stored at every user-shard for that table.

The same may hold for queries filtered on some set of columns. For example, querying for all homework submissions that occured after a certain timestamp may span up to all student shards, since any given student may have submitted a homework after that timestamp. On the other hand, filtering by user id restricts the relevant shards by definition: querying for all submission made by a given user need only involve that user's single shard, since a shard by definition stores all the data of its user.

Our queries rewriter determines whether the tables being queried are sharded or not, and then determines which specific shards to query, by analyzing the select statement's WHERE clause and the state of our system. This overall design works for complex select statements, including ones that join over multiple tables, even if they belong to different shards. However, it is clear that this is not efficient in general, since a select query may result in a similar query executed at every shard. We discuss how this can be mitigated later in this report.

Delete Statements Rewriter: Deletes are handled somewhat similarly to queries. An SQL delete statement may specify a condition specifying which data to delete via a where clause. These conditions may allow our system to choose a subset of shards to execute against. This analysis can be done by looking at the delete statement's WHERE clause as well as the state of our system.

One important difference between deletes and queries is in cascading: depending on schema, deleting a certain row may entail deleting other rows that refer to it. Cascading deletes within a single shard is easy, and we rely on the underlying database system to provide it for schemas that are configured correctly.

Additionally, our design supports efficient cascade-deletion of all of a user's data. Our system identifies delete statements that delete a user, since the system is aware of which tables represent different kinds of users. When such a statement is executed, our system determines the ID of the user being deleted, and uses it to delete the shard of that user, by removing it from the file system, since a shard is an independent mini-database. Notice that this operation requires no joins or filters, and instead is a constant time operation that only de-links an inode from the file system.

Data owned by several users is only deleted when all its owners are deleted. Such data is duplicated along several shards, one for each owning user. When either one of the owners is deleted, the corresponding shard is deleted along with a single duplicate of the data. However, other duplicates remain, and only get removed when all owners are deleted.

Notice that the duplicate data may contain "dangling" values referencing deleted users (or there data). Cross-shard cascading is currently unsupported, we discuss plans for supporting it towards the end of the report.

3 Implementation

Our implementation follows the design outlined in the above section, and is available here. The implementation contains instructions for building and running the system, as well as example SQL workloads.

Our implementation makes two simplifications to the previously described design:

1. We do not support tables with implicit / transitive relationships to users. While our schema rewriter is perfectly capable of handling such tables, inserting to such tables requires the system to identify the owning user(s) by traversing the transitive relationships until the relevant user table is reached, which requires the system to maintain additional caches, make several queries to the underlying shards, or require host applications to scope inserts by users.

   All of these solutions have their own benefits and disadvantages, and a combination of them is likely worthwhile implementing. Neither of them pose any significant design challenges, and are just a matter of implementation.

2. We support a subset of SQL. Specifically, we only support a subset of possible conditionals and expressions in WHERE statements. We make these restrictions in order to simplify our code and the time required to implement it. We detail out the exact restrictions in our source code, which throws runtime errors whenever any of these restrictions is unmet. You can view these restrictions here.

4 Evaluation

4.1 Expressivity

We wrote three different simplified schemas for a social media, doctor-patient medical communication, and assignment submission application. As well as associated simplified insert, query, and delete statements. We have used these applications to test the expressivity of our "PII" and "Owner" abstractions. Specifically, whether we can automatically deduce ownership without annotations in simple scenarios, and whether the "Owner" abstraction is sufficient to encode ownership in more sophisticated scenarios where data is shared between various users. The schemas and SQL operations can be found here.

Our use cases demonstrate that these abstractions have significant expressive power: they can encode scenarios where data is owned by various users, such as a direct message sent over a social media application between two friends, as well as data that is owned exclusively by one user, even though it is related to several ones.

Consider a social media application such as Facebook. Users have the ability to exchange private messages among themselves. A message sent or received by any particular user is in part owned by that user: the user gets to see that message at will, regardless of any policies imposed by other users. When a user requests a copy of all their data from the service, it is reasonable to expect such a message to be part of this data. Furthermore, if either users delete their account, the other user expects that they maintain access to the message history with the deleted user.

In such a scenario, the data is clearly "equally" owned by both users. Using our abstractions, this can be achieved by annotating both the send and receiver as Owners of the data. This results in the expected deletion semantics, as well as expected behavior when either user requests access to their data, and causes our system to duplicate message data in both the sender and receiver shards.

On the other hand, there exists scenario where data is owned by a single user, even if it is related or even created by a different user. For example, imagine a medical chat applications where doctors and patients communicate to describe symptoms and receive diagnosis. Clearly, a patient expects that the service provides them with a complete transcript of all their incoming and outgoing communication with any doctor, whenever that patient requests a copy of their data. Furthermore, the patient is in control of the policies associated with all such communications, including messages containing diagnosis sent by a doctor to that patient. The doctor should not be able to deny the patient access to such a message for example, or unilaterally delete their diagnosis.

Furthermore, a doctor that unsubscribes from this application and request that all their data is removed should not expect current or previous diagnosis they sent to patients to be deleted as well, since patients clearly have a right to maintain such diagnosis as part of their stored medical history. While patients have the right to expect all data around them, particularly their sensitive medical history, is delete upon request.

In this case, the ownership is clearly for the patient, although the particular message also refers to the doctor that sent or received it. This is expressible using our system by applying the Owner annotation to the patient, but not the doctor. This maintains the foreign keys to both patient and doctor tables, but correctly encodes the deletion and access semantics to be in favor of the patient only.

Finally, for data that relates only to a single user, such as a homework submission table, which relates to a student and assignment, but not instructor or other students, our implementation is able to automatically deduce ownership and apply correct deletion and access semantics.

4.2 Performance

In its current state, it is difficult to perform detailed performance experiments on our system. Primarily because our design is only finalized for the right to access and delete functionality. In particular, common host application queries are not meant to be executed using the current rewriting design. Instead, the complete design relies on continously updated materalized views that essentially provide cache-like lookup for these common queries. The current query rewriting is only meant to be used for testing the correctness of sharding internally, and for tinker and debugging by host application developers, but not in production.

For right of access and right of deletion, it is clear to see that our design achieves them with efficient and constant time operations. Especially when compared to traditional databases, which require joins and filters to be performed for either functionalities, relative to the complexity of the schema, over the entire data in the relevant tables in the databases.

5 Future Plan

In addition to augmenting the implementation to handle previously restricted SQL expressions and transitive relationships, we plan on extending the system in three major directions:

Materialized Views We plan on hooking up our system to a data flow system, which transforms common predefined SQL queries given by the host application to a tree of data flow operators that compute materialized views that represent the output of these queries.

Whenever data is inserted via our system, the new data is additionally fed into the data flow system at the appropriate input operators. The data flow system carries out executing these operators, feeding their outputs to the corresponding next level of operators, until the materialized views have been updated.

When the host application executes a concrete instantiation of one of these pre-defined queries, our system looks up the output in the relevant materialized views, according to the concrete input values used in these queries. Thus avoiding the performance cost of executing and aggregating queries over many shards.

Developer-Augmented Deletion Semantics Our ownership primitive, whether explicitly annotated or implicitly deduced, provides our system with some guidance as to how cascading a user deletion must be performed, e.g. whether to delete data if all or either of the users it relates to are deleted. However, our simple formula is unlikely to support all scenarios, such as esoteric ones where data is owned by various users, but is deleted when some subset of them are deleted.

Furthermore, our user deletion leaves behind dangling values referencing the deleted user in other shards, since our system does not know how such values ought to be handled. This is a special case of a larger issue: cross-shard cascading.

We plan to extend SQL with additional delete-oriented annotations that allow developers to specify desired cross-shard cascading functionality. This extension is both critical for integrity as well as privacy, and previous work, such as

DELF [1], demonstrated that it can have good performance when architected correctly.

User Policy In addition to storing all of a user's data within the corresponding shard, we want to store user-provided policies governing that data in the shard as well. These policies dictate how user data (at different granularity) can be processed, especially which data flows the user's data is allowed to be fed into.

Furthermore, we are curious about investigating how such policies can be exported to host applications on query, and how they can be used to restrict processing encoding within host application (and not just data flows within our system), using techniques such as information flow control.

References

[1] Katriel Cohn-Gordon, Georgios Damaskinos, Divino Neto, Joshi Cordova, Benoît Reitz, Benjamin Strahs, Daniel Obenshain, Paul Pearce, and Ioannis Papagiannis. {DELF}: Safeguarding deletion correctness in online social networks. In *29th {USENIX} Security Symposium ({USENIX} Security 20)*, 2020.

[2] Supreeth Shastri, Vinay Banakar, Melissa Wasserman, Arun Kumar, and Vijay Chidambaram. Understanding and benchmarking the impact of gdpr on database systems. *Proc. VLDB Endow.*, 13(7):1064–1077, March 2020.