
# K9db: Privacy-Compliant Storage For Web Applications By Construction

Kinan Dak Albab, Ishan Sharma, Justus Adam, Benjamin Kilimnik, Aaron Jeyaraj, Raj Paul, Artem Agvanian, Leonhard Spiegelberg, and Malte Schwarzkopf,
*Brown University*

https://www.usenix.org/conference/osdi23/presentation/albab

This paper is included in the Proceedings of the 17th USENIX Symposium on Operating Systems Design and Implementation.

July 10–12, 2023 • Boston, MA, USA

978-1-939133-34-2

Open access to the Proceedings of the 17th USENIX Symposium on Operating Systems Design and Implementation is sponsored by


# K9db: Privacy-Compliant Storage For Web Applications By Construction

Kinan Dak Albab Ishan Sharma Justus Adam Benjamin Kilimnik Aaron Jeyaraj
Raj Paul Artem Agvanian Leonhard Spiegelberg Malte Schwarzkopf
*Brown University*

## Abstract

Data privacy laws like the EU's GDPR grant users new rights, such as the right to request access to and deletion of their data. Manual compliance with these requests is error-prone and imposes costly burdens especially on smaller organizations, as non-compliance risks steep fines.

K9db is a new, MySQL-compatible database that complies with privacy laws by construction. The key idea is to make the data ownership and sharing semantics explicit in the storage system. This requires K9db to capture and enforce applications' complex data ownership and sharing semantics, but in exchange simplifies privacy compliance. Using a small set of schema annotations, K9db infers storage organization, generates procedures for data retrieval and deletion, and reports compliance errors if an application risks violating the GDPR.

Our K9db prototype successfully expresses the data sharing semantics of real web applications, and guides developers to getting privacy compliance right. K9db also matches or exceeds the performance of existing storage systems, at the cost of a modest increase in state size.

## 1 Introduction

New privacy laws including the European Union's General Data Protection Regulation (GDPR) [44], the California Consumer Privacy Act (CCPA) [10], and others [7, 17, 20, 56] seek to protect users' rights to their data in web services. Many of these laws provide users with rights to issue subject access requests (SARs), including a *right to access*, which lets users request a copy of their data, and a *right to erasure*, which requires its deletion on request [51, 52]. Many also impose a mandate to store data securely. Compliance with these laws is important, as violations risk severe fines [9, 39–41].

Achieving compliance can be onerous and expensive, however, particularly for small and medium-size organizations. These organizations must write custom queries and track metadata to identify and extract data related to a user, and continuously maintain this infrastructure as services evolve. Even well-intentioned developers sometimes get it wrong: for example, the ownCloud collaboration platform [43], though it claims GDPR compliance [42], retains a user's activity log after account deletion. Retrofitting compliance onto existing systems is tricky, as it still requires manual work [2, 28] and may harm performance [51].

This paper explores an alternative system design that achieves privacy compliance *by construction*. Our key idea is to make data ownership a first-class citizen in the database system itself. K9db, our new database system, tracks sufficient information to know, for each row in the database, what user (or users) have rights to it. This allows K9db to infer correct procedures for data retrieval and deletion, so that the database itself can handle requests under the rights to access or erasure, freeing the application developer from having to write or maintain custom scripts to handle these requests. The ownership information also allows K9db to encrypt data with per-user keys, which helps meet, e.g., the GDPR's "Protection by Design and Default" requirement, which can be satisfied by encrypting at-rest data [36, 44]. Finally, K9db uses ownership information to generate errors if the database schema or operations on database contents risk violating the GDPR.

To realize K9db, we had to address three challenges. First, K9db must understand and model the complex data ownership and sharing semantics of real applications. A user's data may span many tables with transitive relationships, may be shared in complex and data-dependent ways, and may require partial redaction when returned or removed. Second, K9db must maintain and enforce compliance invariants matching these ownership semantics throughout application execution, and correctly respond to user access and deletion requests. Third, K9db should match the performance of today's databases that lack infrastructure for data ownership tracking, and must be both compatible with existing applications and easy for application developers to adopt.

K9db's design addresses these challenges as follows. First, K9db derives a *data ownership graph (DOG)* from a set of coarse-grained, declarative annotations on the database schema. Using a small number of primitives, the DOG models a wide range of complex data sharing relationships found in real-world applications. The DOG is central to K9db's

storage organization, to its handling of users' access and erasure requests, and to K9db's ability to enforce privacy compliance. Second, K9db organizes data storage around data ownership to ensure that applications remain in compliance and handle access and deletion requests correctly by construction, without disrupting regular application operations. Third, K9db is a MySQL-compatible drop-in-replacement for existing databases, and requires few application changes beyond declarative schema annotations for normalized schemas. To accelerate complex queries, K9db provides an integrated, privacy-compliant in-memory cache based on materialized views. By integrating and managing materialized views, K9db provides the benefits of caching to applications, while relieving developers from ensuring compliance of cached data.

K9db structures the actual data storage as a set of user-specific logical "micro-databases" (μDBs), realized over a single physical RocksDB [33] store. Each user's μDB contains the data they own, and is encrypted with a user-specific key. K9db also helps developers use the system correctly by providing compliance-specific functionality not found in other databases. A new `EXPLAIN COMPLIANCE` SQL command gives the developer insight into the DOG and highlights possible schema annotation errors; and K9db supports *compliance transactions* that guard against dynamic compliance problems, such as data without an owner being left behind in the database. K9db provides ACID guarantees similar to those in default MySQL.

K9db provides out-of-the-box compliance for well-intentioned developers who want to comply with privacy laws, and helps developers avoid mistakes. We expect that fines for privacy violations (e.g., the greater than 4% of annual turnover or €25M for GDPR violations) discourage intentional misuse.

In summary, this paper makes the following contributions:

1. The data ownership graph (DOG) for modeling ownership in a database, specified with schema annotations.
2. K9db, a new database that enforces compliance-by-construction based on the DOG and a compliant, ownership-aware storage organization.
3. Mechanisms that, based on the DOG, warn developers if schema annotations are insufficient or if the database becomes non-compliant at runtime.
4. An evaluation of K9db, demonstrating that a database centered around first-class data ownership and compliance-by-construction is practical.

We evaluate K9db with scenarios based on the Lobsters web application [27], the ownCloud document sharing platform [43], and the Shuup e-commerce platform [53]. Our experiments show that K9db can express a wide variety of nuanced data sharing and ownership patterns found in these applications, and that K9db performs on-par with or better than MariaDB and the widely-used MariaDB/memcached stack when serving typical web application workloads.

K9db is open-source at https://github.com/brownsys/K9db.

## 2 Background and Related Work

### 2.1 Privacy Laws

Web services must comply with new privacy and data protection laws [7, 10, 17, 20, 44, 56]. Many of these laws have a comprehensive scope: e.g., the EU's GDPR applies to anyone who offers services to users physically in the EU and touches many aspects of web services [52]. In particular, most laws grant users rights over their data that require services to identify all data related to a user. The GDPR, for example, provides *Subject Access Requests* (SARs) that allow a "*data subject*" (i.e., an end user) to request a copy of their data (Right to Access, Art. 15), to request the deletion of their data (Right to Erasure, Art. 17), and to receive the data in a portable and machine-readable format (Right to Data Portability, Art. 20). Complying with SARs requires the service provider ("data controller" in GDPR terms) to identify the information related to a data subject. As the GDPR has become a model for other privacy laws, many have adopted similar SAR-like requirements. The California Consumer Privacy Act (CCPA), for example, gives consumers a right to request the "specific pieces of personal information [a business] has collected about the consumer" [10, §1789.110] and its deletion [10, §1789.105].

The GDPR and other laws also impose mandates for secure data handling, particularly encryption at rest [44, Arts. 25, 32, 10, §1798.150(a)(1)]. These mandates avoid prescribing particular technologies: e.g., the GDPR only requires that organizations take "appropriate technical measures" to secure personal data [44, Art. 32], giving freedom to meet the requirement in different ways. In practice, encrypting data at rest and deleting encryption keys (referred to as "crypto-shredding"), e.g., to make backups inaccessible, is widely considered a compliant approach [45].

This paper primarily focuses on technical infrastructure to ease compliance with SARs and the requirement for secure storage. Privacy laws also include other provisions that e.g., mandate user consent for processing and regulate data sharing with third parties. Our design is compatible with these requirements, but they are not the focus of this paper.

### 2.2 Complexity of Data Ownership

Compliance with SARs is difficult, both manually and in automated systems, because web services often have complex ownership and data sharing semantics. Identifying data associated with a particular user ("data subject") is challenging. In relational databases, these associations are expressed as foreign keys; but data in many tables link to data subjects transitively via one or more intermediate tables, rather than directly. Multiple data subjects can be associated with the same data (e.g., private messages), and sometimes this association is asymmetric and implies different rights for different data subjects (e.g., a teacher and a student). Finally, many-to-many relationships introduce dynamically changing associations

between data and a variable number of data subjects.

GDPR-like laws afford companies with some flexibility in handling SARs. Applications may keep data associated with the data subject (possibly in some anonymized form) after a deletion request due to legal or contractual obligations (e.g., tax laws) or public interest [44, Art. 17.3]. Data may also be retained depending on the purpose of its processing, including the interests of other users [44, Art 6.1, Art. 17.1(b)]. For example, Facebook's privacy policy specifies that Facebook deletes the comments that a withdrawing data subject made, but not the private messages they sent to a friend, unless that friend also deletes them [16]. Thus, the compliance policy and exact handling of SARs are application and data dependent.

### 2.3 Existing Approaches to Privacy Compliance

Privacy compliance today requires application developers to write custom queries and maintain metadata to identify and track information related to each data subject [51]. The queries are tricky to get right and maintain as the application evolves. To address part of this burden, some large companies built bespoke GDPR metadata stores [13, §1] and dedicated frameworks for data deletion [14]. However, these frameworks only solve part of the problem, and most organizations lack the resources to build such systems themselves. Our work provides compliance within an off-the-shelf database.

Adaptations of existing database systems can go some way towards providing privacy compliance, but can come at a steep performance cost. For example, Shastri et al. found that secondary indexes and strict metadata tracking impose overheads up to 5× [51], leading to proposals to accelerate these operations in hardware [21]. SchengenDB [23] outlines a design that provides GDPR compliance, but relies on extensive metadata and conservative, coarse-grained enforcement, e.g., destroying entire VM clusters when a data subject deletes their account. Our work redesigns the database to make correct privacy compliance a first-class property [49], without sacrificing performance and with moderate overheads.

Other proposals have advocated restructuring web services to enforce users' privacy rights, but face barriers to adoption. W5 [24], Oort [11], Blockstack [3], and Solid [29] decouple data storage from the web application and put data storage under user control. This approach allows for strong guarantees, but requires rewriting web applications, comes with restrictions (e.g., all application logic must run in JavaScript in the browser), and is incompatible with today's advertising-based business model for web services. Data Capsules [58], Riverbed [57], and Zeph [8] let users specify individual privacy policies for their data in web services. Though powerful, custom policies do not solve the problem of identifying all information related to a user; and may limit possible operations (e.g., to those expressible as homomorphic additions). Our work provides by-construction compliance with subject access requests, but with a storage model and database interface that works for existing web applications.

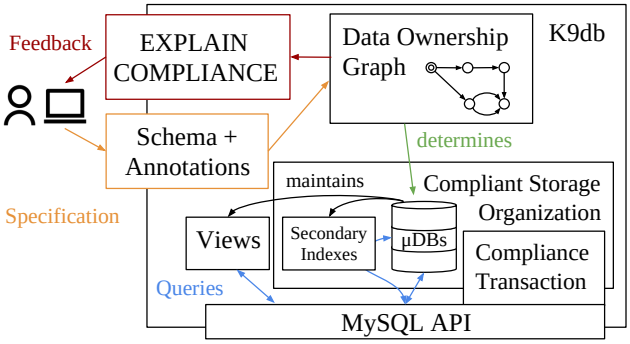



Figure 1: K9db provides privacy-compliant storage based on its data ownership graph, micro-databases (µDBs), and compliance helper mechanisms behind a MySQL interface.

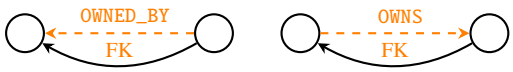
## 3 K9db Overview

K9db is a relational database that makes data ownership an explicit first-class citizen. K9db targets typical web application workloads, which are dominated by reads and point lookup queries [18]. Its design goals are (*i*) to require few changes to application code, (*ii*) to capture and enforce the complex data ownership and sharing semantics of real-world applications, and (*iii*) to provide feedback that helps developers get privacy compliance right.

Figure 1 shows an overview of K9db's components. K9db requires developers to extend their relational schema (i.e., `CREATE TABLE` statements) with a small set of annotations that encode data ownership and sharing semantics. The annotated schema acts as an application-specific compliance policy that specifies how K9db handles SARs. From these annotations, K9db builds its key abstraction, the *data ownership graph (DOG)* (§4). The DOG lets K9db determine, for every row in the database, who owns it and who has rights to it. K9db uses the DOG to satisfy data subjects' SARs, to check that the database remains compliant after the application makes changes, and to warn the developer if their annotated schema and the compliance policy it encodes seem incomplete or contradictory.

Using information from the DOG, K9db organizes its storage in a user-centric way, storing each data subject's data in their own logical "micro-database" (µDB), a shard of the actual database. This design ensures that K9db enforces the developer-provided compliance policy by construction, lets K9db encrypt each data subject's data with a separate cryptographic key, and speeds up compliance-related enforcement and operations (§5). K9db maintains some additional secondary indices compared to a traditional SQL database, which help K9db efficiently resolve which µDBs store particular data. It also maintains materialized views that help simplify and accelerate execution of complex queries, while also providing an integrated, privacy-compliant in-memory cache (§6).

For normalized schemas, K9db requires little to no applica-




(a) Ownership With FK. (b) Ownership Against FK.

Figure 2: K9db's annotations on foreign keys (FKs; orange) indicate the direction of data ownership (black edge) between two tables. Circles are tables.

| Annotation | Example |
|---|---|
| DATA_SUBJECT | CREATE DATA_SUBJECT TABLE users (...) |
| $T_A(x)$ OWNED_BY $T_B(y)$ | stories(author_id) OWNED_BY users(id) |
| $T_A(x)$ OWNS $T_B(y)$ | member(gid) OWNS group(id) |
| $T_A(x)$ ACCESSED_BY $T_B(y)$ | share(share_with) ACCESSED_BY user(id) |
| $T_A(x)$ ACCESSES $T_B(y)$ | taggings(tag_id) ACCESSES tags(id) |
| ON DEL $T_A(x)$ {ANON (...) \| DELETE_ROW} | ON DEL chat(receiver) ANON (receiver) |
| ON GET $T_A(x)$ {ANON (...) \| DELETE_ROW} | ON GET review(paper_id) ANON (reviewer_id) |

Figure 3: K9db's table and column-level annotations. All annotations except DATA_SUBJECT and ANON imply a foreign key from column $x$ in table $T_A$ to column $y$ in $T_B$.

tion code changes, except that developers may need to wrap certain operations in a compliance transaction (§5.5). Developers can use K9db as a drop-in replacement for MySQL.

## 4 Modeling Data Ownership and Sharing

K9db aims to provide correct-by construction compliance with privacy laws, which requires K9db to respond to SARs correctly and enforce several invariants over the data and its storage. Correct compliance has two prongs: (*i*) a compliance policy that is consistent with the privacy law in question, and (*ii*) correct enforcement of this policy when handling both regular application operations and SARs.

The compliance policy is application-specific and depends on the relationships in the underlying data. For a single application, multiple policies may achieve compliance, and laws afford developers some flexibility in choosing a policy that matches their application's semantics (§2.2).

In K9db, developers express their compliance policy using schema annotations, which K9db represents using the *data ownership graph (DOG)*: a directed, acyclic multigraph whose vertices represent database tables, and whose edges represent ownership relationships between rows in the tables.

### 4.1 K9db's Annotations

Developers use schema annotations on foreign keys to communicate their application's data ownership and sharing semantics to K9db. To communicate how the database represents human persons who have rights over data ("data subjects" in GDPR terms), the developer annotates one or more tables with the table-granularity DATA_SUBJECT annotation.

Foreign keys (FKs) relate rows in tables to each other, and often imply ownership—consider e.g., a story pointing to its author. This is the simplest case: a story is owned by the row its FK value points to. K9db provides the OWNED_BY keyword for developers to annotate such FKs (Figure 2a; §4.3 discusses transitive cases). But foreign keys may also point in the *opposite* direction of ownership, as is the case e.g., if a user table has a foreign key to their primary address. For such cases, K9db provides the OWNS annotation (Figure 2b).

In addition to ownership, an application may also have data that is owned by one data subject (who has the right to delete it when removing their account), but share it with others. For example, in the file sharing platform ownCloud [43], users want to share files with others, but when they remove their account have the file be removed for everyone. K9db lets developers express this with the ACCESSED_BY annotation, and its dual for opposite-direction FKs, ACCESSES.

These annotations extend the semantics of foreign keys with compliance semantics, and while every annotation is applied to a foreign key, not every foreign key impacts ownership or needs to be annotated. For example, the foreign key connecting students in a university database with their declared majors carries no ownership information—the students do not own the majors—and should not be annotated.

K9db also provides table-level annotations that allow developers to specify that columns in a table need anonymizing in the context of SARs. This is important because a row may need redacting before returning the row as part of a right-to-access request (ON GET), or because a row may need to be retained in anonymized form (e.g., for tax compliance) after a data subject requests deletion of their data (ON DEL). Each anonymization annotation is associated with an ownership or access foreign key (i.e., an outgoing edge from the table in the DOG). This allows for different anonymization behavior depending on how the data subject who issued a SAR is connected to the data. For example, in the HotCRP conference review system [22], if a data subject who is both a reviewer and an author makes an access request, they should receive an unredacted copy of the reviews they wrote, but redacted, anonymized reviews for the papers they authored.

Figure 3 shows K9db's complete set of schema annotations.

### 4.2 Expressing Developers' Compliance Policies

We demonstrate how developers annotate their schema to express their desired compliance policy using two examples extracted from real applications: stories and messages in Lobsters (Figure 4), and file sharing in ownCloud (Figure 5).

In Lobsters, developers begin by annotating the users table, which records the application's end-users, with DATA_SUBJECT. A user may post several stories, and retains

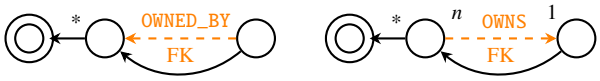
```
CREATE DATA_SUBJECT TABLE users (id INT PRIMARY KEY, ...);
CREATE TABLE stories (
  id INT PRIMARY KEY, title TEXT, ...
  author INT NOT NULL OWNED_BY user(id)
);
CREATE TABLE tags (id INT PRIMARY KEY, tag TEXT, ...);
CREATE TABLE taggings (
  id INT PRIMARY KEY,
  story_id INT NOT NULL OWNED_BY stories(id),
  tag_id INT NOT NULL ACCESSES tag(id)
);
CREATE TABLE messages (
  id INT PRIMARY KEY, body text, ...
  sender INT NOT NULL OWNED_BY user(id),
  receiver INT NOT NULL OWNED_BY user(id),
  ON DEL sender ANON (sender),
  ON DEL receiver ANON (receiver)
);
```

Figure 4: Partial schema for Lobsters. Users own the stories they authored and their associations with tags. Messages are jointly owned by both sender and receiver.

```
CREATE DATA_SUBJECT TABLE user (id INT PRIMARY KEY, ...);
CREATE TABLE group (id INT PRIMARY KEY, title TEXT, ...);
CREATE TABLE member (
  id INT PRIMARY KEY,
  uid INT NOT NULL OWNED_BY user(id),
  gid INT NOT NULL OWNS group(id)
);
CREATE TABLE share (
  id INT PRIMARY KEY, ...
  uid_owner INT NOT NULL OWNED_BY user(id),
  share_with INT ACCESSED_BY user(id),
  share_with_group INT ACCESSED_BY group(id)
);
```

Figure 5: Partial schema for ownCloud file sharing: users own their group membership, which owns the group; files have an owner and are shared with users who have access to them.

sole ownership of them: these stories must be retrieved or deleted when the user issues an SAR. Developers express this by annotating the `author` FK in `stories` with `OWNED_BY`. Lobsters also has a set of tags that represent discussion topics, e.g., `games` and `programming`. Users can assign tags to stories they posted, and have complete ownership of these associations. Developers express this by annotating the `story_id` column in `taggings` with `OWNED_BY`. This makes the story the owner of its taggings, transitively making the data subject who owns the story (i.e., its author) the owner of the associated taggings. But the tags themselves are not related to any data subject. Thus, developers annotate `tag_id` with `ACCESSES` (and not `OWNS`). As a result, a data subject receives a copy of their stories and associated tags when they request access, while disassociating tags from their stories and removing the stories themselves when requesting deletion.




(a) Transitive Ownership. (b) Variable Ownership.

Figure 6: Tables can have transitive ownership relationships (*: zero or more steps of indirection); if an edge follows a one-to-many or many-to-many relationship, it expresses variable ownership. Double circles indicate data subject tables.

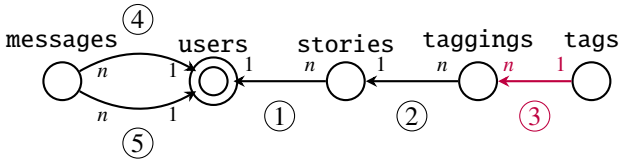



Figure 7: The DOG for stories and messages in Lobsters. Red indicates access-typed edges; 1 and *n* are cardinalities.

Similar to private messages in Facebook [16], messages in Lobsters are only deleted when both sender and receiver request deletion. Thus, developers annotate both `sender` and `receiver` with `OWNED_BY` (i.e., joint-ownership), along with anonymization annotations that instruct K9db to hide the identity of the associated withdrawing user in surviving messages. An alternative policy could require deleting a message as soon as one of the associated users is deleted. Developers can express this via an `ON DEL ... DELETE_ROW` annotation.

ownCloud's data subjects are users in the `user` table, who can be members of a group (in the `group` table), as defined by the `member` association table. Users own their group memberships, so the developer annotates the `uid` column of `member` with `OWNED_BY`. The group and its associated resources are jointly owned by its members (ownCloud has no notion of group admins). Hence, the developer applies the `OWNS` annotation to the `gid` foreign key from `member` to `group`.

ownCloud's `share` table contains records of users sharing files with others. This table specifies the file's owner (i.e., its original creator) via the `uid_owner` column, which is a direct FK to the `user` table. The developer thus annotates this column with `OWNED_BY`. The `share_with` and `share_with_group` columns are also FKs that eventually lead to the `user` table, but indicate that the file is shared with (rather than owned by) these users. The developer therefore annotates them with `ACCESSED_BY`.

### 4.3 Data Ownership Graph

K9db builds the DOG from developers' annotations by inserting DOG edges in the underlying FK direction for `OWNED_BY` and `ACCESSED_BY`, and against the FK direction for `OWNS` and `ACCESSES`. Thus, DOG edges always point towards a data subject table, unlike foreign keys.

When tables have a chain of annotated foreign keys, K9db

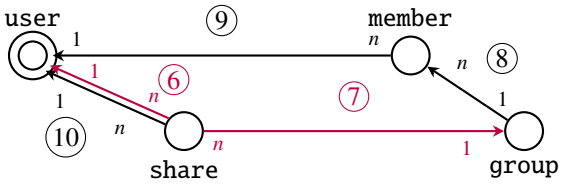



Figure 8: The DOG for ownCloud file sharing. Red edges are access-typed. Note the variable ownership (Fig. 6b) between `member` and `group`, as `member` rows are the group's owners.

adds an edge to the DOG that establishes a *transitive* ownership relationship (Figure 6a). For example, in Lobsters (Figure 7), a story's taggings have no direct references to the story's author. Instead, they refer to their story (2), which in turn refers to the author (1). Therefore, edges in the DOG always represent a single step towards a data subject.

The DOG is a multi-graph because two tables can have multiple foreign keys between them. For example, in Lobsters the `messages` table has two foreign keys, one to the sender (4) and one to the receiver of a message (5). Since sender and receiver jointly own a private message—i.e., the message only disappears if both users delete their account—there are two annotated edges between `messages` and `users`.

Access annotations on foreign keys also add edges to the DOG, but these edges are access-typed and distinct from owner-typed edges. For example, in ownCloud (Figure 8) a file is accessible but not owned by users it is shared with, either directly (6) or via a group (7). Differentiating ownership and access edges is important for K9db to correctly handle access and deletion requests.

If the destination of a DOG edge can contain multiple rows corresponding to a single row in the source table, then that row can have multiple owners or accessors. The DOG edge (8) from ownCloud's `group` to `member` is a one-to-many relationship, so a group may have many owners. This is an example of *variable ownership* (Figure 6b), as the number of owners varies depending on the data (i.e., depending on the rows in `member`). Similarly, DOG edges may also express *variable access*, e.g., a single tag in Lobsters may be accessed through many stories (3). This contrasts with the typical situation where the destination of a DOG edge is a primary key or unique column, making it a one-to-one or many-to-one relationship, both specifying a single owner (e.g., (9) and (10)). K9db's DOG metadata stores arity of relationships and K9db handles variable ownership and access appropriately.

### 4.4 Helping Developers Get Annotations Right

`EXPLAIN COMPLIANCE` gives the developer information about the DOG, including heuristic warnings and suggestions about how it may be improved. K9db runs a simple heuristic over the schema to discover column names which indicate user data such as variations on "name", "email" and "password". If a table with such column names is not connected to a data subject in the DOG, K9db suggests to make it owned. This heuristic is most useful to discover missing data subjects, as their tables often contain columns with such names.

`EXPLAIN COMPLIANCE` also reports information that K9db derives from the DOG. For every table, it reports which data subject tables own it, and the paths through the DOG by which they own the table. This essentially shows the developer the closure over the DOG that K9db uses to handle SARs. `EXPLAIN COMPLIANCE` warns developers if a table is owned by many data subjects, e.g., if a DOG path contains multiple variable ownership edges, which can result in multiplicatively many owners. Such liberal sharing is rare in practice and likely the result of a schema or annotation mistake.

### 4.5 Data Ownership Graph Properties

The DOG is *well-formed* if any path through it terminates at a data subject table. K9db rejects any schema that results in a DOG that is not well-formed.

Although the DOG is a graph of tables, its edges represent relations between rows in the source and destination tables based on the values of the underlying FK columns. Each DOG edge maps to a *relation* between rows in the two tables, where matching rows in the destination table own (or access) the rows in the source table. Intuitively, this relation can be evaluated as a query over the destination table, which yields exactly the owning row (or rows, in the case of variable ownership). Well-formedness guarantees that the transitive closure of these relations terminates at data subject tables.

Several key properties follow from this. First, if no matching rows exist in any destination table when evaluating the relations along *all* of the table's outgoing ownership edges, data is orphaned (i.e., has no owner). This gives rise to the necessary (but insufficient[1]) *no orphaned data* compliance condition: any row in a database table connected to the DOG must resolve to $\geq 1$ owning data subjects. Second, the transitive closure of relations corresponding to ownership edges in the DOG, starting from any row, identifies the set of data subjects that own this row. Third, the DOG's reverse transitive closure starting from a row in a data subject table yields:

1. the rows shared with and owned by that data subject, if considering accessor-typed and owner-typed edges; or
2. the rows owned by that data subject, if considering only owner-typed edges.

The former set corresponds to the data that needs returning from a right-to-access request, and the latter identifies the data that needs deleting for a right-to-erasure request, provided no other owners exist.

## 5 Compliant by Construction Storage

In principle, the DOG and its relations are sufficient to identify a data subject's data, and one could imagine adding it as a metadata layer over an existing database. But in practice,

[1] Sufficiency would require the *correct* owners, not just any owner.

compliance is more complex. Although the DOG identifies all data owned by a data subject, K9db needs to take the correct actions on this data. For example, K9db must avoid prematurely deleting jointly-owned data, and deletion must cover backups outside the live database. K9db must also have efficient ways to decide if a given database operation will break compliance, e.g., by violating the *no orphaned data* invariant, something that the DOG alone fails to provide.

K9db therefore introduces ownership as a first-class notion into the storage layer. This makes it simple for K9db to handle SARs, and to enforce invariants that must hold for compliance. Specifically, K9db's storage layer is organized around per-data subject logical "micro-databases" (μDBs), such that each μDB contains all of its data subject's owned data. For jointly-owned data, K9db stores copies of that data in the μDB of every data subject that owns it.

This design has several advantages. First, it ensures data deletion is correct relative to the DOG. When a data subject requests to delete their data, it is sufficient to delete their μDB. Data shared with other data subjects survives as copies in the other μDBs. Second, this design provides an easy way to check whether data is orphaned, as such data can only exist outside of all data subjects' μDBs. Third, this design lets K9db use a per-data subject key to encrypt data in each μDB. This simplifies deletion alongside external and replicated backups of the data, as deleting the owner's key makes all backups and copies inaccessible (i.e., "crypto-shredding").

### 5.1 Storage Layout and Logical μDBs

K9db determines the μDBs to store each row in using the DOG. In a well-formed DOG, every table reaches at least one data subject table via its outgoing ownership edges. K9db splits the contents of such a table into different μDBs, each of which contains the rows owned by a particular data subject, and encrypts them with a key specific to that data subject. A table also includes an orphaned data section that may be used temporarily within sequences of operations (§5.5). A data subject's μDB therefore includes rows from every table that stores data owned by them. Note that even though μDBs store physical copies of rows that have multiple owners, they are a logical abstraction and realized over a single underlying physical datastore (e.g., RocksDB in our prototype).

Viewing the datastore as a whole, a previously single row in a table may now be multiple rows due to copies being stored in each owner's μDB. The value of the primary key of that row refers to all these copies. Internally, K9db identifies the different copies using a pairing of the data subject identifier (the value of its primary key in the data subject table) and the value of the primary key in the row.

K9db maintains on-disk secondary indexes separate from tables and μDBs, which K9db uses to execute queries efficiently. K9db creates an on-disk index for each unique and foreign key column and for the primary key. K9db on-disk indexes differ from traditional database indexes in two key aspects: they map keys to *(μDB identifier, primary key)*, and they point to all copies of any jointly-owned row that match the indexed key. K9db creates a special index for the primary key column(s) of owned tables, which maps the PK value to data subject identifiers that own the corresponding row.

K9db stores tables unconnected to the DOG in the same way as other databases. Such tables contain data that is not owned by any data subject, e.g., all available tags in Lobsters or all majors in a university database, and thus are outside any μDB. Note that this is distinct from orphaned data, which are rows without owners in tables that *are* connected to the DOG.

### 5.2 μDB Integrity

The storage layer maintains an important invariant for compliance, *μDB completeness*: data owned by a data subject is exactly identical to the data stored in their μDB.

To maintain μDB completeness, K9db must identify the μDBs to insert new data into, and correctly apply application updates that change who owns rows. Changes to the data in a table may have cascading effects on who owns data in dependent tables connected to this table via some ownership path in the DOG. For example, changes to the `member` table in ownCloud affect who owns records in the `group` table. K9db utilizes the DOG to handle these situations correctly.

**Inserting Data.** When K9db receives an `INSERT` statement, it uses the DOG to identify the owners of this data. In particular, K9db analyzes the outgoing edges from the DOG vertex for the affected table. For a direct ownership edge, the data subject identifier is already present in the new row in the form of a foreign key. K9db determines this by introspection on the new row and without querying other tables. If an edge indirectly leads to the data subject table, identifying the owner becomes more complex. K9db can find the owner(s) by querying the database along the transitive edges between the table and the data subject. But such a query may be expensive—for example, the DOG for the Shuup e-commerce application [53] contains a chain of five edges from the payments table to the owning data subject. Instead, K9db memoizes the query by building and maintaining in-memory *ownership indexes*, which essentially provide "shortcut" relations over the DOG that point directly to the owning data subjects. In practice, K9db can often avoid or reuse ownership indexes (§6.1).

**Cascading Updates.** `INSERT`, `UPDATE`, or `DELETE` statements may have cascading effects on the ownership of records in dependent tables. After applying such statements to their target table, K9db identifies dependent tables from the DOG. It then queries the rows in each dependent table that match the updated row. K9db moves or copies the matched rows between μDBs appropriately, and cascades again into any further dependent tables. K9db requires no additional indexes to perform this matching efficiently, as it can rely on standard on-disk indexes over foreign keys' source and destination columns. In many cases, K9db avoids cascades via optimizations based on foreign key integrity (§6.1).

### 5.3 Handling Subject Access Requests

K9db needs to handle two types of SARs: the *right to access* and the *right to erasure*. K9db handles both with a similar high level procedure: (*i*) K9db traverses the DOG to identify all tables and edges connected to the data subject; (*ii*) K9db finds the data owned by the data subject in their µDB; (*iii*) K9db locates data accessed, but not owned, by the data subject in other µDBs; and (*iv*) K9db performs anonymization as specified by the developers in the schema.

For either type of request, K9db identifies the data subject's data by following paths in the DOG, starting from the data subject table, and moving against incoming edges. A path that consists solely of ownership edges signifies data owned by the data subject, while paths that contain one or more access edges reflect accessed data. K9db locates the relevant rows in a table before moving on to any dependent tables. For every incoming edge, K9db uses the rows it located in the parent table to identify dependent rows in the dependent table. K9db finds these either in the same µDB for ownership paths, or in other µDBs using on-disk indexes for access paths.

After traversing an edge and retrieving data in its source table, K9db selects the anonymization annotations in the schema that apply to that edge. The anonymization annotations specify the columns to anonymize (e.g., the sender of a chat message). For access requests, K9db anonymizes retrieved rows before sending them back to the client. On deletion requests, K9db removes the data subject's µDB from the database, and anonymizes any remaining copies of the data, which it locates in other µDBs using on-disk indexes.

### 5.4 Atomicity, Consistency, Isolation, and Durability

A single SQL statement may result in several underlying operations over K9db's storage, as it may update rows in several µDBs or cascade over dependent tables. It is critical for compliance that we ensure that these updates are all ACID, to avoid data races that could lead to a non compliant state (e.g., by creating orphaned data, or breaking the µDB completeness invariant). Therefore, K9db executes every SQL statement as a single statement ACID transaction (similar to MySQL). This includes all underlying operations over any µDBs and all updates to on-disk secondary indices or the integrated in-memory cache (§6.2). Our prototype does not support general multi-statement SQL transactions yet (see §7).

K9db guarantees that concurrent SQL statements have *repeatable reads* isolation, which is the default in MySQL. Any weaker isolation level is insufficient for compliance, as it cannot guarantee that K9db's compliance invariants hold in the presence of concurrent updates.

### 5.5 Compliance Transactions

An application may itself perform operations that risk violating compliance. Consider the example from ownCloud shown in Figure 9: ① the application deletes user "A"'s membership in group 1, of which "A" is the last remaining

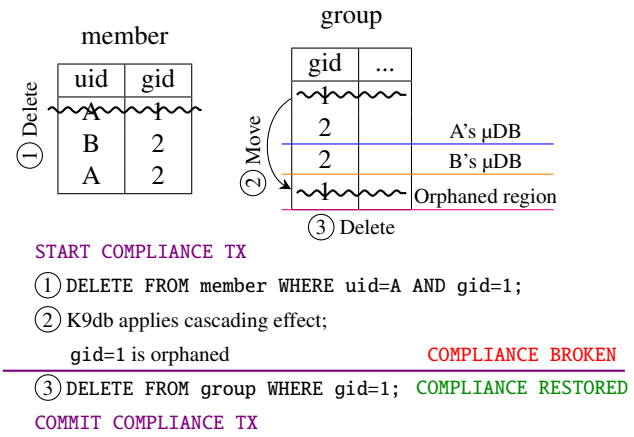



Figure 9: K9db's *compliance transactions* help developers check that the database is in a compliant state after multiple operations (here, (1) deleting the last owner of a group, and (3) then deleting the group). Without a compliance TX, K9db would report an error instead of applying step (2).

member. This deletion from `member` has a cascading effect on the dependent `group` table. Since the group with `gid` 1 no longer has any owners, K9db ② moves it into the table's orphaned data region. This breaks compliance, as it violates the DOG's *no orphaned data* invariant. A correct application must now perform some operation that restores compliance, e.g., by deleting group 1 in a separate SQL operation, which ③ removes the orphaned row, restoring the invariant.

K9db supports this pattern with the idea of a *compliance transaction* (CTX). A CTX wraps a set of operations that may temporarily violate compliance, but commits only if the database is back to a compliant state at the commit point. Within a CTX, K9db stores orphaned data in orphaned regions attached to each table. On subsequent operations that reintroduce owners for this data, K9db migrates the rows from the orphaned regions to the corresponding µDBs; if deleted, K9db removes the data. At the end of a CTX, K9db ensures that every record moved to the orphaned region during the CTX has an owner again (or was deleted), and produces an error to the developer otherwise.

Finally, K9db forbids statements that write to the orphaned region unless they are part of a CTX. In particular, step ① in Figure 9 will error unless contained in a CTX. This means that developers need to modify applications that contain such patterns to use CTXs when necessary. Requiring such limited modification is desirable, as disallowing compliance-breaking changes outside of CTX helps developers identify issues and forces them to fix buggy and incompliant applications. For example, K9db would reject a buggy version of ownCloud that does not clean up groups with no members ③. Introducing a CTX allows an application to have benign temporary incompliance; if K9db instead required applications to only

perform operations that move the database between compliant states (e.g., deleting groups before deleting their last member), it would likely require more substantial rewrites.

CTX are different from regular SQL transactions, which serve to ensure consistency under concurrent execution. CTX are lightweight and required for compliance, while SQL transactions are expensive and web applications often (but not always) avoid them. In a privacy-compliant database with SQL transactions, each such transaction must also be a CTX.

# 6 Query Execution

When K9db executes a query, it must identify the µDBs affected to locate the relevant rows. Depending on the operation, this may involve finding one or all copies of shared rows.

Queries that refer to a single table, such as `DELETE` and `UPDATE` statements, and most `SELECT` queries issued by web applications (e.g., point lookups), run directly against K9db's µDBs with the aid of on-disk indexes. K9db analyzes the columns that appear in the `WHERE` condition of the query, and selects the index that matches the most columns. Like other databases, K9db finds all the rows that may match the query using the selected index, and then filters these rows with any remaining columns. If no index matches, K9db runs a scan over the table. Developers may create additional indexes using `CREATE INDEX`, similar to traditional databases.

When data has multiple owners, an index may refer to multiple copies of the same row. For `DELETE` and `UPDATE`, K9db atomically operates over all these copies, ensuring that all copies are consistent. K9db may need to remove or add some of the affected rows from/to µDBs, and may need to cascade into dependent tables as described in §5.2. For `SELECT` queries, K9db identifies a single copy of each matching row and skips any remaining index entries for other copies. This avoids overheads for deduplicating copies of the row.

K9db serves some complex `SELECT` queries from materialized view, described in §6.2.

## 6.1 Optimizations

K9db speeds up query execution and reduces its memory footprint with a set of optimizations designed to avoid deep cascades and to reduce the number of in-memory ownership indexes (§5.2) required. Some of these optimizations rely on *foreign key integrity*, which K9db enforces (like many other databases) to prevent application operations that result in dangling foreign keys. With FK integrity, rows cannot be inserted into a table if they contain references to non-existent rows in a destination table, and rows in the destination table cannot be deleted as long as source table rows refer to them.

**Avoiding Cascades.** K9db needs to cascade into dependent tables along incoming DOG edges to update dependent rows affected by a write (i.e., those owned by a modified row). But FK integrity guarantees that no such rows exist when K9db handles `INSERT` and `DELETE` queries to a table $T$ that is the destination of a FK from a dependent table. This lets K9db skip cascades along $T$'s incoming DOG edges if the edge is in FK direction; otherwise, K9db must cascade.

**Ownership Indexes.** K9db relies on two techniques to reduce the number of ownership indexes. First, multiple incoming DOG edges that require an ownership index and point to the same column of a table (usually the primary key) may reuse the same index. Second, K9db omits ownership indexes for edges in the DOG that correspond to `OWNS` annotations, such as the edge from `group` to `member` in ownCloud. These edges point in opposite direction to the underlying foreign key. FK integrity ensures that a row must exist at the source of such an edge (e.g., `group`) before any rows referring to it can be inserted to the destination table (e.g., `member`). Hence, K9db always inserts new rows from the source table into the orphaned region, and defers moving them to the correct µDB to future inserts into destination tables in the DOG (which must cascade), as discussed in §5.5. These optimizations, for example, help K9db create only one ownership index for Lobsters (which gets re-used three times), and avoid the need for any ownership indexes in ownCloud.

**Queries With Inlined Owners.** SQL Statements sometimes directly refer to the owner of their target rows, e.g., by constraining a foreign key that corresponds to an ownership edge in the DOG. Queries that fit this pattern are common in the web applications: e.g., in Lobsters, `SELECT * FROM stories WHERE author = ?` selects stories by their author, which is an annotated foreign key to `users`. K9db detects this situation by statically analyzing the `WHERE` condition and determines the relevant µDB without an on-disk index lookup.

## 6.2 Materialized Views

K9db serves complex `SELECT` queries, such as joins, aggregations, and those that reorder data, from materialized views. This design makes sense for two reasons. First, it is simple and avoids the need to engineer a sophisticated query planner that understands the nuances of ownership and indexes to efficiently execute these queries over K9db's µDBs. Second, developers often cache the results of complex `SELECT` queries in external systems (e.g., memcached). Privacy compliance while using an external cache requires setting appropriate expiration policies for the cache [59, §4.5] or explicit invalidation of cache entries related to a data subject if they request deletion of their data. This can be painful for developers and may require manually tracking metadata, e.g., when caching aggregates over many data subjects' data. Instead, K9db provides an integrated privacy-compliant cache using materialized views.

When K9db receives a complex `SELECT` query for the first time, it creates a materialized view and serves further instances of the query from it, until the view is removed or times out. K9db keeps the materialized views up to date via an incremental, streaming dataflow computation triggered by writes to µDBs, as well as µDB deletion. This makes inserts, updates, and deletes more expensive, but speeds up reads.

K9db updates the materialized views atomically prior to acknowledging the corresponding operation to the client. This, along with our storage layer, ensures *repeatable reads* isolation for concurrent operations whether cached or not.

K9db's ownership indexes are special-case materialized views, maintained with the same dataflow infrastructure.

# 7 Implementation

Our K9db prototype consists of 35k lines of C++, 500 lines of Rust, and 2k lines of Java. It relies on RocksDB for µDB storage, on Apache Calcite [6] for query planning, and on libsodium [15] for encryption. Our implementation is similar to the MyRocks MariaDB storage engine [30], but extends it with compliance and µDB capabilities.

**MySQL Compatibility Layer.** K9db exposes a MySQL binary protocol interface, so unmodified applications can treat K9db as a MySQL server. The interface to K9db's materialized views is primarily through prepared SQL statements: when an application registers a prepared statement, K9db creates a view if necessary and serves future executions of the prepared statement from it. Developers can also create additional views manually.

**Storage.** K9db relies on RocksDB for persistent data storage. Each table in the schema is a RocksDB column family. Rows in K9db are keyed by a combination of their owner and primary key, to uniquely identify each owner's copy of a row. Our prototype stores rows ordered by their owner identifier, and uses that identifier as a RocksDB prefix. This allows it to extract and delete µDBs using RocksDB prefix iterators. Our prototype creates and maintains on-disk indexes as RocksDB column families, and formats their content to allow writes to retrieve all the copies of a row, and reads to retrieve a single arbitrary copy, skipping the rest. Like MySQL, K9db creates indexes for primary, unique, and foreign keys.

**Encryption at Rest.** K9db uses hardware-accelerated AES256-GCM to encrypt all data in a µDB with the key of its owner. The key (µDB identifier, primary key) associated with every row is encrypted deterministically with a global key to allow consistent lookup. This has leakage, but is sufficient to satisfy the GDPR's "security of processing" requirement (Art. 32), which is often interpreted to require encryption of data at rest [4]. It is possible to use blind indexes [5] which also allow consistent lookup but reduce leakage. K9db's design is independent of the particular encryption scheme used, and can benefit from future advances in searchable encryption. Information in materialized views and secondary indexes remains unencrypted, but K9db deletes it when deleting a user's data. K9db destroys the decryption key when a user removes their account, making any remaining backups inaccessible.

**ACID.** K9db executes each application SQL statement in a RocksDB transaction, which is based on row-level locking. This includes all updates to secondary indices (similar to MyRocks) and all µDBs and cascade operations. As in MyRocks, K9db serves reads from a consistent RocksDB snapshot. K9db also updates all relevant materialized views prior to committing. Unlike MyRocks, K9db enforces foreign key integrity and appropriately locks FK targets during execution. Overall, this ensures that concurrent SQL statements are atomic and consistent with *repeatable reads* isolation, which is the default in MySQL and MyRocks.

**View Updates.** K9db's materialized view updates follow a standard design akin to differential dataflow [32, 35] and Noria [18]. Each table in the schema is associated with an input vertex in the dataflow graph, and when K9db performs updates to a table, it injects the updates into its dataflow input vertex. The dataflow processes the updates through a sequence of operators to derive an incremental update to the materialized view (or secondary index), and applies this update. Dataflow operators are stateless (e.g., projections, filters, unions) or stateful (e.g., joins, aggregations). K9db's materialized views are indexed for ordered and unordered lookups.

**Limitations.** Our prototype lacks support for general, multi-statement SQL transactions. These are rare in web applications, and can be supported using existing RocksDB primitives and techniques for versioned dataflow processing [31, 35]. While our prototype does not yet support schema changes, RocksDB is schema-oblivious, and our prototype's storage layer could be extended to support schema changes with some engineering effort, using similar techniques to MyRocks. Finally, K9db's dataflow graph operators sometimes store copies of a record; by using a record pool, our prototype's memory footprint could be reduced.

# 8 Evaluation

We evaluate K9db with three applications, Lobsters [27], ownCloud [43], and Shuup [53]. We ask three questions:

1. What is K9db's impact on end-to-end application performance? (§8.1)
2. What is the impact of K9db's design features on performance? (§8.2)
3. What effort by application developers does using K9db require? (§8.3)

We run experiments on a Google Cloud `n2-standard-16` VM, storing databases on a local SSD. Our baselines use MariaDB v10.6.5 (a MySQL fork) with the RocksDB-based MyRocks storage engine, and memcached v1.6.10.

## 8.1 Application Performance

We start by analyzing K9db's performance with two applications: Lobsters and ownCloud.

### 8.1.1 Lobsters

Lobsters (lobste.rs) is an open-source discussion board, similar to Reddit. Lobsters currently lacks GDPR compliance [26], and has a schema that consists of 19 tables, which store posts, comments, nested replies, upvotes, invitations and other information. We annotated this schema for K9db with three `DATA_SUBJECT` tables, 14 `OWNED_BY`, one `ACCESSES`, and

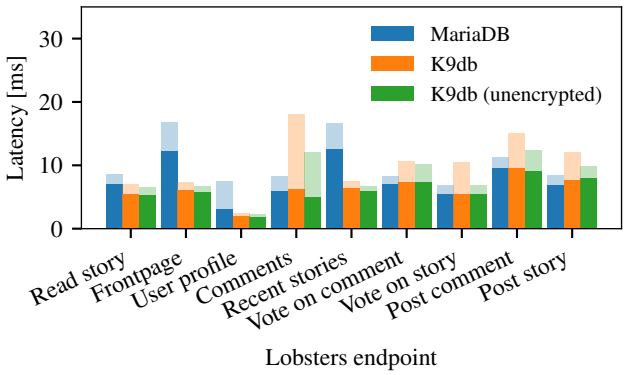



Figure 10: K9db matches or beats MariaDB's median (solid) and 95$^{th}$ percentile (shaded) latency on the Lobsters workload, and encryption has low overheads except on the "Comments" endpoint, which reads thousands of rows in the tail.

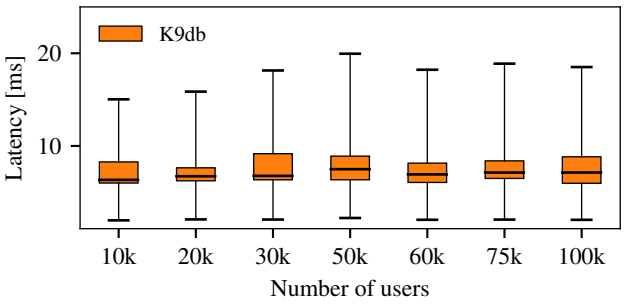



Figure 11: K9db's 95$^{th}$%ile latency on the Lobsters workload remains stable as the number of users (and thus, µDBs) increases. Each bar shows a distribution of endpoint latencies.

two anonymization annotations (details in §8.3). We use an existing open-source, open-loop benchmark for Lobsters based on public workload statistics [19]. The benchmark models ten endpoints in the Lobsters webapp that correspond to different pages and each issue between six and fifteen SQL queries, most of which are reads. We load the database with data that models the current production Lobsters deployment (15k users, 100k stories, 313k comments, and 416k votes) [19]. K9db therefore maintains 15k logical µDBs in this experiment. We compare MariaDB, and K9db with and without data encryption. (Encryption with per-user keys isn't possible in the MariaDB baseline.) Lobsters on most requests runs an expensive query to determine the user's recently read stories. This query joins four tables, including the (large) stories and comments tables. This query is slow in MariaDB (≈30ms) and dominates its latency for all endpoints, while K9db serves this query from a materialized view. To make the comparison fair, we remove the expensive query in the MariaDB baseline. A good result for K9db would show latencies comparable to MariaDB for all endpoints, and a low overhead for encryption.

Figure 10 shows the results. Endpoints that mostly read (on the left) benefit from K9db's materialized views and are up to 2.1× faster than in MariaDB, but endpoints with many writes (on the right) are comparable in both systems. This makes sense, as K9db performs similar work to MariaDB, except that some read queries are served from materialized views, and writes need to be encrypted and must update any corresponding views. K9db without encryption is on-par with K9db in most endpoints. For the "Comments" endpoint, K9db is 2.1× slower than MariaDB and 1.5× slower than K9db without encryption in the 95$^{th}$ percentile. This happens when the endpoint retrieves comments and votes on a popular story from the database, which requires K9db to decrypt thousands of records. Developers could manually add materialized views in K9db to speed up this endpoint, at the cost of additional memory. Other endpoints read fewer rows or rely on (unencrypted) materialized views. This shows that K9db achieves good performance for a practical web application, and that encryption has acceptable cost. All further experiments show results for K9db with encryption enabled.

We chose the load in this experiment to saturate the hardware for the MariaDB baseline (≈ 760 pages/second, which results in 10k queries/second) and used the same load for K9db. K9db supports a up to a 4.8× higher load without latency degradation, thanks to its caching for complex queries via materialized views; we compare to a caching MariaDB+memcached baseline below.

**Subject Access Requests.** We now measure the time required by K9db to satisfy SARs. We issue an access and a deletion request for each of the top 1000 users with most data in the database, and run these requests sequentially through K9db SARs API. Performance of SARs is secondary as they are rare operations and can be executed asynchronously. A good result shows that K9db handles SARs correctly (which it does by construction) and within reasonable time. In our experiment, K9db on average takes 1 ms to retrieve and 45 ms to delete the correct data for a user.

**Scalability.** We designed K9db to have performance independent of the number of µDBs. We confirm this using the Lobsters benchmark with different numbers of users. Adding users increases the number of µDBs and the amount of data in the database, but keeps the average amount of data per user constant. A good result for K9db would show latencies remaining constant as the number of users grows.

Figure 11 shows the results as box-and-whisker plots over the nine endpoints (i.e., the bottom and top whiskers are the fastest and slowest endpoints, respectively). K9db's latency remains constant as the number of users—and, consequently, µDBs—grows, because K9db satisfies queries either from µDBs directly, via indexes, or from materialized views. These results confirm that K9db's logical µDB partitioning is practical for applications with large numbers of users.

**Comparison to Caching Baseline.** In the previous experiment, K9db had an unfair advantage over MariaDB: it serves some data from materialized views, while Mari-

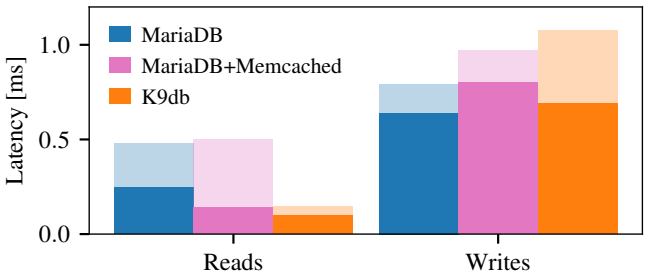



Figure 12: K9db matches MariaDB+memcached on a common Lobsters query (solid: median; shaded: 95$^{\text{th}}$%-ile).

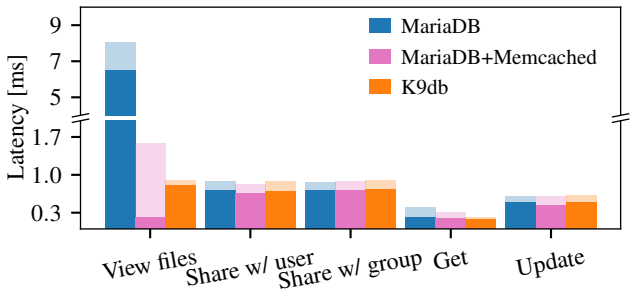



Figure 13: K9db matches the baseline setups' performance on the ownCloud workload (solid: median; shaded 95$^{\text{th}}$%-ile).

aDB recomputes queries every time. We now use one common query from Lobsters to compare three setups: (*i*) standalone MariaDB; (*ii*) MariaDB with an in-memory cache ("MariaDB+Memcached"); and (*iii*) K9db. The MariaDB+Memcached setup is a demand-filled cache [37]: writes invalidate the cached query result in memcached, and the next read re-runs the query against the database when it misses in memcached. In K9db, writes update views via its dataflow graph. We generate a skewed workload with a Zipfian distribution ($s = 0.6$) where 95% of requests in the benchmark read the details of a random story and its vote count, and 5% of requests insert new votes. A good result for K9db would show competitive read performance with memcached and low overheads on write processing (since K9db does more work on writes); and MariaDB+Memcached and K9db would show lower latencies than MariaDB alone.

Our results are in Figure 12. For reads, MariaDB+Memcached and K9db are on par in the median, but K9db has a lower 95$^{\text{th}}$ percentile latency as K9db updates the cache via streaming dataflow, while MariaDB+Memcached queries the database on a read miss. All systems perform similarly on writes, as this query requires little dataflow update work in K9db and the caching baseline must make an extra RPC to invalidate memcached.

**Memory Overhead.** K9db's materialized views and ownership indexes add memory overhead compared to a traditional database. We measure this cost and compare it to a caching setup with memcached. We consider a setup that caches query results that developers would typically store in memcached, such as the output of expensive joins and aggregates. These queries are identical to the ones that K9db caches using materialized views. The experiment caches query results with the query parameters (? in prepared statements) as the key, and the concatenated records as the value. K9db stores additional in-memory data for internal dataflow state and ownership indexes. A good result for K9db would therefore show moderate overheads compared to MariaDB+Memcached.

The Lobsters database is 61 MB on disk, and a typical memcached caching approach stores an additional 97 MB of in-memory state. K9db's memory footprint is 197 MB (3.3× DB size, and 2× memcached's footprint), which includes 6.5MB for the stories ownership index, and 56 MB for caching the expensive query we removed from MariaDB (without this query, K9db's overhead is 2.4× DB size/1.5× memcached). The overhead comes from K9db's dataflow state, which allows K9db to incrementally update materialized views.

### 8.1.2 ownCloud

ownCloud is a popular open-source application that allows users to upload files and share them with other users [43]. Recall ownCloud's schema (Figure 5): each file has a single owner—the original uploader—but users can share files with other users and with groups. Files shared with a group are accessible to all members of the group—i.e., a many-to-many relationship between users and files (a pattern absent in Lobsters). We measure five common queries: (*i*) listing the files a user can access ("view files"); (*ii*) sharing a file with another user ("share with user"); (*iii*) sharing a file with a group ("share with group"); (*iv*) retrieving a file using its primary key ("Get"); and (*v*) updating the retrieved file ("Update"). Our setup uses 100k users who each own three documents; each document is shared uniformly at random with three users and two groups; and each group has five members. Our workload is 95% read and 5% writes, equally split among the two types of sharing and file updates. Reads and writes target users drawn from a Zipf distribution ($s = 0.6$). We batch ten reads and measure the per-request latency for the same setups as in the previous experiment. A good result for K9db would show comparable read latency to MariaDB+Memcached and low overheads on writes.

Figure 13 shows the results. "View files", which returns all files shared with a user (directly or via a group), involves five tables and three joins, which MariaDB executes on every read. MariaDB+Memcached and K9db serve precomputed results from memory instead, which is fast. The 95$^{\text{th}}$ percentile for MariaDB+Memcached suffers because it queries MariaDB on a cache miss, which occurs when a query retrieves files of user(s) invalidated by a previous write. K9db is fast and stable because it updates the views via dataflow on writes. All systems perform similarly for the two share queries—a good result for K9db, as it also updates views.

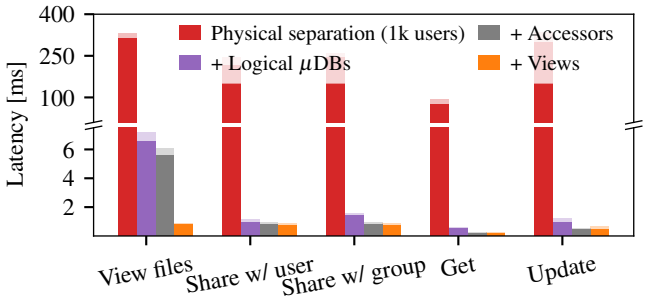



Figure 14: K9db matches the baseline setups' performance on the ownCloud workload (solid: median; shaded 95$^{th}$%-ile).

### 8.2 K9db Design Drill-Down

To evaluate the impact of design decisions central to K9db, we run ownCloud workload from the previous experiment against versions of K9db that disable key components. We start with K9db set up to naïvely store every µDB in its own database (without cross-µDB indexes); without support for accessor edges in the DOG; and without materialized views (i.e., queries always run over data in RocksDB). This guarantees strict separation of user's data, a solution sometimes adopted for GDPR compliance in practice [46, 47], although this lacks support for shared data (accessors) or anonymization. We then add separation into logical µDBs ("+ Logical µDBs"), accessor support ("+ Accessors"), and materialized views ("+ Views"). A good result would show that these features improve K9db's performance.

Figure 14 shows the results. The naïve µDB design is very slow because every query that K9db cannot statically resolve to the affected µDBs requires scanning all µDBs; we only ran this setup with 1k users (vs. 100k for the others). Making µDBs a logical abstraction much improves performance, justifying our design choice. Accessor-typed DOG edges are important for expressivity: without them, ownCloud would be restricted to a policy where users jointly own shared files. In addition, accessor support reduces the number of copies stored and the fan-out of writes, which slightly reduces query latency. Finally, materialized views improve latency of the "View files" query by 5×, as the results are cached in memory. Since the view update is cheap, writes do not suffer much overhead. The runtime of "View files" without no views is comparable to the runtime of the same query in MariaDB (Figure 13). This illustrates that views are beneficial, but not essential to good performance in K9db.

### 8.3 Schema Annotation Effort

To understand the developer effort K9db's schema annotations require, we now consider annotations for three applications (Lobsters, ownCloud, and Shuup [53]) in detail,

**Lobsters.** The Lobsters schema contains 19 tables. To use K9db, we had to annotate the schemas for eight tables. Three tables (`users`, `invitations`, and `invitation_requests`) contain data subjects. We annotated two FKs in each of `hats`, `messages`, and `moderations` with `OWNED_BY` to model joint ownership. We annotated 8 other tables with a single `OWNED_BY`. For example, `votes` has multiple foreign keys that lead to the `users` table (one direct, two indirect), and thus requires a single `OWNED_BY` annotation to disambiguate and ensure votes are stored with the voter, rather than the author of the story or comment voted on. Finally, we used one `ACCESSES` in `taggings`, and two anonymization rules in `messages`, as shown in Figure 4.

**ownCloud.** ownCloud's schema has 51 tables. We focused on the file sharing core, which consists of six tables and has the most complex relationships. In addition to the annotations in Figure 5, we added an `OWNS` annotation to the FK in the `share` table that points to the corresponding file in the `file` table (omitted from Figure 5 for brevity).

ownCloud's original schema "overloads" the `share_with` column to either hold a user or a group ID, and includes a `share_type` column to distinguish these cases. K9db could support such de-normalized schema with more advanced conditional annotations; for our benchmarks, we modified the schema to track users and groups in separate columns.

**Shuup.** Shuup [53] is an open source e-commerce platform and supports customers with accounts, guests who do not have accounts, and shop owners, all of whom have GDPR rights. The Shuup code lets users request their account to be anonymized, but retains information for tax compliance, e.g., payment data, customers' countries of residence, and tax ID numbers, a form of data retention provided for in the GDPR.

Shuup provides GDPR compliance via a manually-implemented module with 4k lines of Python code (2.7k lines of implementation and 1.3k lines of tests), developed in 137 commits over three years. At the time of writing, Shuup's anonymization behavior is inconsistent; it only anonymizes default shipping and billing addresses, but retains previous addresses in cleartext in the `mutable_address` table [55]. Moreover, downloading data for a user is not supported [54].

We implemented Shuup's anonymization policy in K9db using all annotations (Figure 3) over 17 of Shuup's 278 tables. We annotate `personcontact` with `DATA_SUBJECT`. This table stores natural persons, and has FKs to their contact information (in `contact`) and their logins (in `auth_user`) if they have accounts. Thus, `personcontact` contains users with and without accounts, i.e., guests. Using K9db, Shuup correctly anonymizes data, lets users download the data and fixes the bug of not anonymizing previous default addresses.

Shuup's schema has several tables that might correspond to data subjects. K9db's `EXPLAIN COMPLIANCE` helps developers understand that they need to annotate `personcontact`. An incompliant (but plausible) alternative would be to annotate `auth_user`, the login details table. This results in `contact` being unconnected to the DOG, as there are no foreign keys to `auth_user`. The `personcontact` table has such a foreign key, but it is nullable (e.g., for guests who lack

| Application | Tables | Data Subject | Owner | Access | Anon |
|---|---|---|---|---|---|
| Commento [12] | 12 | 3 | 8 | 1 | 3 |
| ghChat [1] | 6 | 1 | 7 | 2 | 4 |
| HotCRP [22] | 26 | 2 | 15 | 10 | 7 |
| Instagram clone [50] | 19 | 1 | 18 | 1 | 0 |
| Mouthful [25] | 3 | 1 | 1 | 0 | 0 |
| Schnack [48] | 5 | 1 | 1 | 0 | 0 |
| Socify [34] | 19 | 1 | 10 | 0 | 0 |

Figure 15: K9db requires few `DATA_SUBJECT`, ownership (`OWNED_BY` and `OWNS`), access (`ACCESSED_BY`, `ACCESSES`), and `ANON` annotations to support real web applications.

accounts), and thus some of its rows will be stored in µDBs and others in the orphaned region. `EXPLAIN COMPLIANCE` helps developers identify and rectify these issues:

```
Table "contact": GLOBAL
 [Compliance Warning] Column "email" suggests personal
      data, but the table is not connected to any owners.
Table "personcontact": in µDB for auth_user.id
 [Compliance Warning] Table has owners, but nullable
      foreign key may prevent correct deletion of data.
```

A developer might also annotate `contact` with `DATA_SUBJECT`, but that table includes entries for customers and companies. Annotating it makes companies into data subjects, which duplicates company-related tables across µDBs. `EXPLAIN COMPLIANCE` also alerts developers to this.

**Other Applications.** Our schema annotations were sufficient to express reasonable compliance policies for seven additional applications (Figure 15). We briefly highlight several interesting patterns in these applications.

In ghChat [1], a chat application for GitHub, and the Instagram clone [50], a group is owned exclusively by its admin and accessed by its members. This is unlike ownCloud, which lacks group admins and has members jointly own the group.

Mouthful [25] is a commenting service that embeds in a host application (e.g., a blog) to allow users to comment on the host content (e.g., a blog post). Mouthful has no notion of users; instead, the host application provides a string that represents the user identity alongside the comment they posted. We added a `DATA_SUBJECT` table to store user identifiers, and created a FK constraint from the `Comment` table's `author` column to it.

Finally, the HotCRP [22] review system associates data subjects to papers via a many-to-many `PaperConflict` table. The table has a `conflictType` column that specifies the relationship, such as "co-author" or "institutional conflict". While this schema is normalized in the traditional SQL sense, it is not normalized for ownership: rows with the co-author type signify ownership, while other conflict types do not imply any ownership or access rights over the paper. We resolved this by adding a new `PaperAuthors` table that only stores authorship associations, and refer to papers from it using `OWNS`. We reserve the existing `PaperConflict` to record other conflict types with an un-annotated reference to papers.

**Migrating Applications to K9db.** We identify some common challenges when migrating applications to K9db. First, annotating an application schema requires knowledge of the application functionality and its compliance policy, but also summarizes the policy in an easy-to-maintain way alongside the schema. Many web applications also lack explicit FK constraints in their schema; developers must identify the columns that act as implicit FKs and annotate them if needed.

Second, applications often have schemas that are not normalized in the traditional SQL sense (e.g., ownCloud's `share_with`) or with regards to ownership (e.g., HotCRP's `PaperConflict`). Developers must normalize these schemas by introducing new columns or tables, and apply the corresponding changes to the application code. K9db could support such schemas via new annotations that condition on other columns, but this would complicate the annotation language and DOG model. Instead, K9db guides developers to good, normalized schema designs.

Finally, applications with variable ownership (e.g., ownCloud, Shuup, HotCRP) often have endpoints that temporarily orphan data. Developers must wrap such endpoints in compliance transactions in order to use K9db. This modification is relatively unobtrusive, and K9db can be configured to automatically wrap sessions in a CTX. This alleviates the need to manually introduce CTX to applications that open new sessions for each endpoint or sequence of operations, but is not suitable for applications with long-lived sessions.

## 9 Conclusion

K9db is a new database system that achieves compliance with the requirements of privacy laws by construction.

K9db models data ownership to capture the ownership patterns of real world applications, and handles requests for access and deletion correctly. K9db matches or exceeds the performance of a widely-used database and manual caching setup, and supports the privacy requirements of real-world applications. K9db is open-source and available at https://github.com/brownsys/K9db.

## Acknowledgements

We are grateful to Deniz Altınbüken, Hannah Gross, Frans Kaashoek, Franco Solleza, Lillian Tsai, and the ETOS group at Brown for helpful feedback on drafts of this paper. Feedback from the anonymous reviewers and our shepherd, Natacha Crooks, greatly improved the paper. We also thank Vedant Gupta, Mithi Jethwa, and Colton Rusch for contributions to K9db's implementation.

This research was supported by NSF awards CNS-2045170 and DGE-2039354, by a Google Research Scholar Award, and by a gift from VMware.

## References


[1] aermin. *ghChat (react version)*. URL: https://github.com/aermin/ghChat (visited on 05/02/2021).

[2] Archita Agarwal, Marilyn George, Aaron Jeyaraj, and Malte Schwarzkopf. "Retrofitting GDPR Compliance onto Legacy Databases". In: *Proceedings of the VLDB Endowment* 15 (Dec. 2021).

[3] Muneeb Ali, Jude Nelson, Ryan Shea, and Michael J. Freedman. "Blockstack: A Global Naming and Storage System Secured by Blockchains". In: *Proceedings of the 2016 USENIX Annual Technical Conference (ATC)*. Denver, Colorado, USA, June 2016, pages 181–194.

[4] Amazon Web Services. *Navigating GDPR Compliance on AWS: Encrypt Data at Rest*. URL: https://docs.aws.amazon.com/whitepapers/latest/navigating-gdpr-compliance/encrypt-data-at-rest.html (visited on 05/05/2021).

[5] Scott Arciszewski. *Building Searchable Encrypted Databases with PHP and SQL*. May 2017. URL: https://paragonie.com/blog/2017/05/building-searchable-encrypted-databases-with-php-and-sql.

[6] Edmon Begoli, Jesús Camacho-Rodríguez, Julian Hyde, Michael J. Mior, and Daniel Lemire. "Apache Calcite: A Foundational Framework for Optimized Query Processing Over Heterogeneous Data Sources". In: *Proceedings of the 2018 International Conference on Management of Data*. Houston, Texas, USA, 2018, 221–230.

[7] National Congress of Brazil. *Lei Geral de Proteção de Dados [Brazilian General Data Protection Law]*. English translation by Ronaldo Lemos, Daniel Douek, Sofia Lima Franco, Ramon Alberto dos Santos and Natalia Langenegger. URL: https://iapp.org/media/pdf/resource_center/Brazilian_General_Data_Protection_Law.pdf (visited on 06/11/2020).

[8] Lukas Burkhalter, Nicolas Küchler, Alexander Viand, Hossein Shafagh, and Anwar Hithnawi. "Zeph: Cryptographic Enforcement of End-to-End Data Privacy". In: *Proceedings of the 15th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*. Virtual Event, July 2021, pages 387–404.

[9] California Attorney General. *Privacy Enforcement Actions*. URL: https://oag.ca.gov/privacy/privacy-enforcement-actions (visited on 05/06/2021).

[10] California Legislature. *The California Consumer Privacy Act of 2018*. June 2018. URL: https://leginfo.legislature.ca.gov/faces/billTextClient.xhtml?bill_id=201720180AB375.

[11] Tej Chajed, Jon Gjengset, M. Frans Kaashoek, James Mickens, Robert Morris, and Nickolai Zeldovich. *Oort: User-Centric Cloud Storage with Global Queries*. Technical report MIT-CSAIL-TR-2016-015. MIT Computer Science and Artificial Intelligence Laboratory, Dec. 2016.

[12] Adhityaa Chandrasekar. *Commento*. URL: https://github.com/adtac/commento (visited on 05/02/2021).

[13] Andrew Chung, Subru Krishnan, Konstantinos Karanasos, Carlo Curino, and Gregory R. Ganger. "Unearthing inter-job dependencies for better cluster scheduling". In: *Proceedings of the 14th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*. Banff, Canada, Nov. 2020, pages 1205–1223.

[14] 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: *Proceedings of the 29th USENIX Security Symposium (USENIX Security)*. Banff, Canada, Aug. 2020.

[15] Frank Denis. *The Sodium cryptography library*. 2013. URL: https://download.libsodium.org/doc/.

[16] Facebook. *Permanently Delete Your Facebook Account*. URL: https://www.facebook.com/help/224562897555674?helpref=faq_content (visited on 05/21/2023).

[17] Thailand Government Gazette. *Personal Data Protection Act*. Unofficial English translation. URL: https://thainetizen.org/wp-content/uploads/2019/11/thailand-personal-data-protection-act-2019-en.pdf (visited on 06/11/2020).

[18] Jon Gjengset, Malte Schwarzkopf, Jonathan Behrens, Lara Timbó Araújo, Martin Ek, Eddie Kohler, M. Frans Kaashoek, and Robert Morris. "Noria: dynamic, partially-stateful data-flow for high-performance web applications". In: *Proceedings of the 13th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*. Carlsbad, California, USA, Oct. 2018, pages 213–231.

[19] Peter Bhat Harkins. *Lobste.rs access pattern statistics for research purposes*. Mar. 2018. URL: https://lobste.rs/s/cqnzl5/lobste_rs_access_pattern_statistics_for#c_hj0r1b (visited on 03/12/2018).

[20] PRS Legislative Research India. *The Personal Data Protection Bill, 2019*. URL: https://www.prsindia.org/billtrack/personal-data-protection-bill-2019 (visited on 06/11/2020).

[21] Zsolt István, Soujanya Ponnapalli, and Vijay Chidambaram. "Software-Defined Data Protection: Low Overhead Policy Compliance at the Storage Layer is within Reach!" In: *Proceedings of the VLDB Endowment* 14.7 (Mar. 2021), pages 1167–1174.

[22] Eddie Kohler. *HotCRP conference review software*. URL: https://github.com/kohler/hotcrp (visited on 07/22/2020).

[23] Tim Kraska, Michael Stonebraker, Michael Brodie, Sacha Servan-Schreiber, and Daniel Weitzner. "SchengenDB: A Data Protection Database Proposal". In: *Proceedings of the 2019 VLDB Workshop Towards Polystores that manage multiple Databases, Privacy, Security and/or Policy Issues for Heterogenous Data (Poly)*. Los Angeles, California, USA, Aug. 2019, pages 24–38.

[24] Maxwell Krohn, Alex Yip, Micah Brodsky, Robert Morris, and Michael Walfish. "A World Wide Web Without Walls". In: *Proceedings of the 6th Workshop on Hot Topics in Networks (HotNets)*. Atlanta, Georgia, USA, Nov. 2007.

[25] Viktoras Kuznecovas. *Mouthful*. URL: https://github.com/vkuznecovas/mouthful (visited on 05/02/2021).

[26] Lobste.rs. *Privacy: Lobsters*. URL: https://lobste.rs/privacy (visited on 05/01/2021).

[27] Lobsters Developers. *Lobsters News Aggregator*. Mar. 2018. URL: https://lobste.rs (visited on 03/02/2018).

[28] Connor Luckett, Andrew Crotty, Alex Galakatos, and Ugur Cetintemel. "Odlaw: A Tool for Retroactive GDPR Compliance". In: *Proceedings of the 37th IEEE International Conference on Data Engineering (ICDE)*. Chania, Greece, Apr. 2021.

[29] Essam Mansour, Andrei Vlad Sambra, Sandro Hawke, Maged Zereba, Sarven Capadisli, Abdurrahman Ghanem, Ashraf Aboulnaga, and Tim Berners-Lee. "A Demonstration of the Solid Platform for Social Web Applications". In: *Proceedings of the 25th International Conference Companion on World Wide Web (WWW)*. Montréal, Québec, Canada, 2016, pages 223–226.

[30] MariaDB. *MyRocks – MariaDB Knowledge Base*. URL: https://mariadb.com/kb/en/myrocks/ (visited on 12/06/2022).

[31] Frank McSherry, Andrea Lattuada, Malte Schwarzkopf, and Mothy Roscoe. "Shared Arrangements: practical inter-query sharing for streaming dataflows". In: *Proceedings of the VLDB Endowment* 13.10 (June 2020), pages 1793–1806.

[32] Frank McSherry, Derek G. Murray, Rebecca Isaacs, and Michael Isard. "Differential dataflow". In: *Proceedings of the 6th Biennial Conference on Innovative Data Systems Research (CIDR)*. Asilomar, California, USA, Jan. 2013.

[33] Meta Platforms, Inc. *RocksDB: A persistent key-value store for fast storage environments*. URL: http://rocksdb.org/ (visited on 12/10/2022).

[34] Sudharsanan Muralidharan. *Socify: open source social network using Ruby on Rails*. URL: https://github.com/scaffeinate/socify (visited on 05/02/2021).

[35] Derek G. Murray, Frank McSherry, Rebecca Isaacs, Michael Isard, Paul Barham, and Martín Abadi. "Naiad: a timely dataflow system". In: *Proceedings of the 24th ACM Symposium on Operating Systems Principles (SOSP)*. Farmington, Pennsylvania, USA, Nov. 2013, pages 439–455.

[36] European Network and Information Security Agency. *Privacy and data protection by design: from policy to engineering*. 2015. URL: https://data.europa.eu/doi/10.2824/38623.

[37] Rajesh Nishtala, Hans Fugal, Steven Grimm, Marc Kwiatkowski, Herman Lee, Harry C. Li, Ryan McElroy, Mike Paleczny, Daniel Peek, Paul Saab, David Stafford, Tony Tung, and Venkateshwaran Venkataramani. "Scaling Memcache at Facebook". In: *Proceedings of the 10th USENIX Conference on Networked Systems Design and Implementation (NSDI)*. Lombard, Illinois, USA, Apr. 2013, pages 385–398.

[38] Noria Contributors. *Noria Lobsters benchmark*. 2020. URL: https://github.com/mit-pdos/noria/tree/3edd3ad55d2564493f7456d27abb41abf0169def/applications/lobsters.

[39] NOYB: European Center for Digital Rights. *GDPRHub: CNIL SAN-2020-008*. URL: https://gdprhub.eu/index.php?title=CNIL_-_SAN-2020-008 (visited on 05/06/2021).

[40] NOYB: European Center for Digital Rights. *GDPRHub: CNIL SAN-2020-018, Nestor SAS*. URL: https://gdprhub.eu/index.php?title=CNIL_-_SAN-2020-018 (visited on 05/06/2021).

[41] NOYB: European Center for Digital Rights. *GDPRHub: GPDDP 9485681, Vodafone Italia*. URL: https://gdprhub.eu/index.php?title=Garante_per_la_protezione_dei_dati_personali_-_9485681 (visited on 05/06/2021).

[42] ownCloud GmbH. *GDPR compliant cloud storage*. URL: https://owncloud.com/gdpr (visited on 12/01/2021).

[43] ownCloud GmbH. *owncloud – share files and folders, easy and secure*. URL: https://owncloud.com (visited on 12/01/2021).

[44] "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)". In: *Official Journal of the European Union* L119 (May 2016), pages 1–88.

[45] Brent Robinson. *Crypto shredding: How it can solve modern data retention challenges*. 2019. URL: https://medium.com/@brentrobinson5/crypto-shredding-how-it-can-solve-modern-data-retention-challenges-da874b01745b.

[46] Alexander Rubin. *40 million tables in MySQL 8.0 with ZFS*. URL: https://www.percona.com/blog/2018/09/03/40-million-tables-in-mysql-8-0-with-zfs/ (visited on 05/03/2021).

[47] Alexander Rubin. *One Million Tables in MySQL 8.0*. URL: https://www.percona.com/blog/2017/10/01/one-million-tables-mysql-8-0/ (visited on 05/03/2021).

[48] schnack! *schnack.js*. URL: https://github.com/schn4ck/schnack (visited on 05/02/2021).

[49] Malte Schwarzkopf, Eddie Kohler, M. Frans Kaashoek, and Robert Morris. "GDPR Compliance by Construction". In: *Proceedings of the 2019 VLDB Workshop Towards Polystores that manage multiple Databases, Privacy, Security and/or Policy Issues for Heterogenous Data (Poly)*. Los Angeles, California, USA, Aug. 2019.

[50] Faiyaz Shaikh. *React-Instagram-Clone-2.0*. URL: https://github.com/yTakkar/React-Instagram-Clone-2.0 (visited on 05/02/2021).

[51] Supreeth Shastri, Vinay Banakar, Melissa Wasserman, Arun Kumar, and Vijay Chidambaram. "Understanding and Benchmarking the Impact of GDPR on Database Systems". In: *Proceedings of the VLDB Endowment* 13.7 (Mar. 2020), pages 1064–1077.

[52] Supreeth Shastri, Melissa Wasserman, and Vijay Chidambaram. "How Design, Architecture, and Operation of Modern Systems Conflict with GDPR". In: *Proceedings of the 11th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud)*. July 2019.

[53] Shuup Commerce, Inc. *Shuup Open-Source E-Commerce Platform*. URL: https://github.com/shuup/shuup (visited on 12/05/2021).

[54] Shuup Contributors. *GDPR - Download Data button doesn't return any data*. URL: https://github.com/shuup/shuup/issues/2614 (visited on 12/13/2021).

[55] Shuup Contributors. *GDPR - shuup_mutaddress rows not anonymized*. URL: https://github.com/shuup/shuup/issues/2612 (visited on 12/13/2021).

[56] Griffin Thorne. *GDPR Meets its Match ... in China*. July 2019. URL: https://www.chinalawblog.com/2019/07/gdpr-meets-its-match-in-china.html (visited on 06/04/2020).

[57] Frank Wang, Ronny Ko, and James Mickens. "Riverbed: Enforcing User-defined Privacy Constraints in Distributed Web Services". In: *Proceedings of the 16th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*. Boston, Massachusetts, USA, Feb. 2019, pages 615–630.

[58] Lun Wang, Joseph P. Near, Neel Somani, Peng Gao, Andrew Low, David Dao, and Dawn Song. "Data Capsule: A New Paradigm for Automatic Compliance with Data Privacy Regulations". In: *Proceedings of the 2019 VLDB Workshop Towards Polystores that manage multiple Databases, Privacy, Security and/or Policy Issues for Heterogenous Data (Poly)*. Los Angeles, California, USA, Aug. 2019, pages 3–23.

[59] Juncheng Yang, Yao Yue, and K. V. Rashmi. "A Large-Scale Analysis of Hundreds of In-Memory Key-Value Cache Clusters at Twitter". In: *ACM Transactions on Storage* 17.3 (2021).

# A Artifact Appendix

**Abstract**

Our open source artifact contains our prototype implementation of K9db. It also includes the harnesses and scripts for running and plotting the experiments described in this paper.

Our prototype provides a MySQL-compatible interface layer, which applications and developers can use to issue SQL statements and queries to and retrieve their results. Our prototype is compatible with the standard MySQL connectors and drivers for several languages, including C++, Rust, and Java. It is also compatible with the command line MySQL and MariaDB clients.

### Scope

Our prototype serves as a demonstration of the following:

1. The application scenarios described in the paper work with K9db and its schema annotations.
2. K9db's system design and guarantees can be realized with a familiar MySQL-compatible interface suitable for web applications.
3. The performance of compliant-by-construction databases is comparable to traditional databases, such as MariaDB.

### Contents

**K9db.** The artifact includes our prototype implementation and its MySQL-compatiblity layer. The artifact contains instructions for building, running, and using this K9db.

**Application Harnesses.** The artifact includes harnesses for Lobsters, a Reddit-like discussion board (§8.1.1), and ownCloud (§8.1.2), a file sharing application. The harnesses create the database schema and load the database with data; they also execute loads with representative queries, and measure the time required to process them. We used these harnesses to evaluate our prototype and the baselines shown in our experiments. The Lobsters harness is a pre-existing open source harness that we adapted to work with our prototype [38].

**Documentation.** The artifact wiki on GitHub contains a tutorial on using K9db and its schema annotations. The artifact also includes unit and end-to-end tests that validate that our prototype handles application SQL operations correctly and provides correct compliance with SARs.

### Hosting

Our artifact is hosted on GitHub at `https://github.com/brownsys/K9db`. The version of the repository corresponding to this paper is available at `https://github.com/brownsys/K9db/releases/tag/osdi2023`, with commit hash *df2bcdffa05f70f508fad95a11e2a6de8a7efe14*. The corresponding wiki commit hash is *c720b085ca34edc16246f296991e623a29933f9b*.

### Requirements

We developed our prototype on x86-64 machines running Ubuntu 20.04 and 22.04. We provide a Docker container that includes the necessary software dependencies. We ran our experiments on Google Cloud using `n2-standard-16` machines with a local SSD.