DeCor: Ensuring the Right To Be Forgotten in Web Applications

Anonymous Authors

Abstract

Web services increasingly face legal requirements to protect users' privacy (e.g., the GDPR). These laws require companies to properly delete and anonymize users' data when a user requests to remove their data. And beyond current legal requirements, many users would rest easier if they knew that they can unsubscribe from a web service at any time, without losing their data and with the opportunity to come back later.

But unsubscribing users without compromising either the leaving user's privacy or the application experience for the remaining users is difficult. Removing too much information meets privacy requirements, but may yield surprising application behavior (e.g., disappearing content); removing too little risks exposing the identity of unsubscribed users via inference attacks.

In this paper, we propose DeCor, a new system for safe unsubscription and resubscription of users in database-backed web services. DeCor helps application developers specify fine-grained privacy policies to meet de-identification requirements, and goes beyond: with DeCor, it is possible for users to switch between a privacy-preserving unsubscribed mode and an identity-revealing subscribed mode at any time without permanently losing their data. This facilitates important new web service paradigms, such as users granting a time-limited "lease" of data to a service instead of having a permanent service account.

1 Introduction

Web application operators increasingly face pressure from legal requirements that grant users new rights over their data. For example, both the European Union's General Data Protection Regulation (GDPR) [3] and California's Consumer Privacy Act (CCPA) [1] grant users the right to request erasure of information related to them. These laws, for the first time, require companies to support *unsubscription* of users from their services on request.

Unsubscription requires more than merely deleting a user's data: correct operation of a service often requires that some information remains, in transformed or anonymized form, after a user departs. [malte: Give an example!] It is difficult for developers to determine and implement the database transformations required to correctly unsubscribe a user, and to continously maintain this feature. Consequently, the majority of web services today either lack unsubscription support (requiring manual labor to handle e.g., GDPR deletion requests), or leave substantial information behind that potentially harms user privacy.

In this paper, we describe DeCor, a system that provides abstractions and mechanisms that help developers of databased-backed web applications achieve correct, privacy-compliant user unsubscription without onerous labor. Application developers using DeCor specify a declarative *unsubscription policy*, which indicates how the database contents need to change to meet de-identification requirements on user unsubscription. DeCor then turns this policy into a set of concrete database operations that remove, anonymize, and structurally decorrelate user data, and executes them. But DeCor goes beyond simply unsubscribing users: it makes it possible for users to switch between a privacy-preserving unsubscribed mode and an identity-revealing subscribed mode at any time without permanently losing their data. This facilitates new web service designs, such as the user granting a time-limited "lease" of data to a web service instead of having a permanent service account. When a user resubscribes, DeCor both re-imports missing data and *recorrelates* the remnants of the user's prior data with the user account, restoring the user to her original subscribed state as much as possible.

DeCor makes the key observation that a user's data entities (e.g., posts and upvotes) can leak identifying information in two ways: (1) directly via its content, and (2) indirectly via the correlations it has to other data entities in the system. Correlations can range from obviously

identifying (e.g., posts correlated with a user clearly belong to that user), to subtly perilous: posts correlated with a particular user-generated tag most likely belong to the tag's author, and posts liked by the same group of users likely belong to a friend of the group.

Many web applications (e.g., Lobst.rs and Reddit) have come up with solutions to de-identify direct content by anonymizing unique identifiers, leaving arbitrary user-generated content out of scope. Other applications (e.g., Facebook and Twitter) delete (most of the) user's data. The former optimizes the amount of data the application keeps, but also retains all correlations between data entities; the latter can lead to confusing semantics for the application (e.g., nonsensical comment threads) and the loss of useful application data. In both cases, accounts cannot be restored after deletion. [lyt: Applications such as Facebook or Twitter do support user account deactivation by hiding (most) of the user's data, but retain all user data with their identifiers in their databases.]

Developers using DeCor can specify policies that both anonymize unique identifiers, and remove identifying information stemming from correlations between data entities. DeCor's support for fine-grained decorrelation policies gives developers options beyond either completely retaining these links or deleting of all of the user's data and their dependees (while still supporting these options). The complexity of implementating such policies is hidden from the developer: DeCor automatically performs decorrelation in a way that allows for recorrelation upon resubscription, while still achieving performance comparable to today's widely-used databases and requiring no modification of application schemas.

1.1 The goal of decorrelation

Decorrelation ideally breaks connections between data entities that would otherwise allow an adversary to determine that two entities belong to the same (unsubscribed) user: post-decorrelation, an adversary should not be able to distinguish between a scenario in which two data entities have been generated by two distinct users, and one in which they were both generated by the same (unsubscribed) user. If two of a user's entities cannot be correlated back to the user, then at most one of these entities may leak identifying information about the user.[1] Thus, if the content of individual data entities is appropriately anonymized, then perfect decorrelation prevents identifying information from being leaked via correlations.

[1] Assume for a contradition that both entities leak identifying information: then both entities are more likely to have been generated by a particular identity than any other, contradicting our assumption.

1.1.1 Threat Model

An adversary aims to relink decorrelated entities to unsubscribed users after DeCor performs decorrelation. We make the following assumptions about such an adversary:

- An adversary can perform only those queries allowed by the application API, i.e., can access the application only via its public interface.
- An adversary cannot perform application queries to the past or search web archives: information from prior application snapshots may reveal exactly how data records were decorrelated from unsubscribed users.
- An adversary cannot gain identifying information from arbitrary user-generated content (for example, a reposted screenshot, or text in user stories or comments). Decorrelation seeks to remove identifying information from user-generated data that can be enumerated or follows a specific pattern (e.g., a birthday or email address), and application metadata (e.g., date of postings, database ID columns).

1.1.2 Problems with decorrelation taken too far.

Decorrelation taken to the extreme breaks all correlations between entities recursively related to the user being anonymized, replacing these correlations with ghost correlations. This removes as much correlation-based identifying information as possible while keeping data entities present, but in practice, completely decorrelated data entities may be useless to the application. Applications may lack a meaningful way to generate ghost entities: what does it mean for a story to become many ghost stories? And even if ghosts can be generated, the noise and data pollution from ghost entities may instead affect the accuracy and semantics of the application: users may see meaningless content, comment threads may be disjoint and scattered, and highly-ranked content may suddenly lose votes.

DeCor offers developers a way to decorrelate as much as possible *while still retaining application semantics*. As the next section describes, DeCor provides decorrelation specification primitives with which developers can specify which (and how) entities can be decorrelated from other entities, and which correlations must be kept for application correctness.

2 Background and Related Work

2.1 Anonymization in Web Applications

Today's web applications are incentivized to retain as much user data on their platform as possible, both to sell

and to process for the application's targeting, and to increase the utility of their platform for other users (e.g., keeping reviews or votes for other users of the application to use to judge a product).

When users on web applications delete their accounts, some applications (e.g., Lobst.rs and Reddit) choose to de-identify user data upon account deletion by replacing unique user identifiers such as usernames with global placeholders ("anonymous") or a pseudonym ("anonymous_mouse"). Such applications have privacy policies allowing them to retain a deleted users' content. Other applications (e.g., Facebook and Twitter) delete (most of the) user's data and its dependencies [2]. The first choice optimizes the amount of data the application retains, while the second may lead to confusing semantics for the application (e.g., nonsensical comment threads). In both cases, accounts cannot be restored after deletion.

Applications such as Facebook or Twitter do support user account deactivation by hiding the user's data, but retain all user data with their identifiers in their databases. [lyt: Facebook keeps a "log" of data versions, but this wouldn't be a problem if everything in the database is ghosted!]

DeCor's decorrelation primitives provide a way for applications to state more flexible privacy policies, trading off data retention with de-identifying a user. DeCor also supports resubscription without allowing the application to retain identifying user data in its database.

2.2 Data Anonymization

Prior research on k-anonymization, pseudonymization (more for static analysis purposes than used in web applications).

While psuedo/anonymization techniques still allow for reidentification/reconstruction attacks), DeCor provides better anonymization than prior approaches, which decorrelate only coarsely by associating remnants of data with a global placeholder for all deleted users, or do not decorrelate at all (e.g., replacing usernames with a pseudonym, as in the AOL dataset case [lyt: TODO]

2.3 Differentially Private Queries

PINQ, a data analysis platform created by McSherry [7], provides formal differential privacy guarantees for any query allowed by the platform. Data analysts using PINQ can perform transformations (Where, Union, GroupBy, and restricted Join operations) on the underlying data records prior to extracting information via aggregations (e.g., counts, sums, etc.) The aggregation results include added noise to meet the given privacy budget ε, ensuring that analysts only ever receive ε-differentially-private results. PINQ calculates the privacy loss of any given query based on transformations and aggregations to be performed; if a particular query exceeds a predefined privacy budget, PINQ refuses to execute the query.

Differential privacy provides a formal framework that defines the privacy loss a user incurs when the user's data is included in the dataset. However, the setting of differential privacy that of DeCor in several important ways.

First, web applications' utility often derives from visibility into individual data records. PINQ restricts JOIN transformation and exposes information only through differentially-private aggregation mechanisms. While sufficient for many data analyses, this approach severely hinders web application functionality. DeCor supports all application queries, even ones that expose individual records, such as `SELECT story.content, story.tag FROM stories LIMIT 10`. However, this makes formally defining the privacy guarantees of DeCor more complex than simply applying DP. DP provides formal guarantees for results such as sums or averages because such results are computed using well-defined mathematics, allowing us to neatly capture the total knowledge gained by the adversary. However, in the world of web applications, the knowledge gained via queries that may reveal individual data records cannot be so easily defined: adversaries may learn information from the friends who liked the (ghosted) story, or the time and location the story was posted.

Thus, we cannot, in general, use the DP approach of asking, "by how much do answers to adversary queries change with the presence of a users' decorrelated data?" While we can precisely define which data records an adversary sees with and without a user's decorrelated data, we cannot numerically quantify the knowledge gained by the adversary in the same way as we can quantify the percentage change to a sum or average. Instead, DeCor reduces identifying information in the changes induced by a users' (decorrelated) data, rather than reducing the amount of change itself.

This is demonstrated by how PINQ and DeCor differ in masking entities associated with unique keys. PINQ restricts JOINs to group by unique keys. As an (imperfect) example, selecting a JOIN of stories with users via the UID would result in a record, one per UID, which represents the "group" of all stories associated with a user. A normal join would create a separate record per story, each associated with the user of that UID.

Instead of lumping all matching stories together, DeCor explodes the users' UID into individual unique keys (ghosts), one per story. By PINQ's analysis of stable transformations, this leads to unbounded stability (as the number of different results produced by queries is proportional to the potentially very large number of stories for that user). In PINQ's DP framework, such unbounded stability

leads to unbounded privacy loss: one users' data could change aggregation results in "significant" ways unless huge amounts of noise were added.

DeCor and PINQ's approaches can be seen as complementary: for queries that are more analytic in nature (for example, population statistics used by the web application to measure usage or trends), DeCor can utilize PINQ's technique to provide differentially private guarantees (for a particular snapshot of the dataset). DeCor's decorrelation, in fact, adds more noise than necessary in these cases: a count of unique users who posted about cats will be more imprecise because each post by a decorrelated user is now associated with a unique (ghost) user.

2.4 Deletion Privacy

The right to be forgotten has also been formally defined by Garg et al. [4], where correct deletion corresponds to the notion of leave-no-trace: the state of the data collection system after a user requests to be forgotten should be left (nearly) indistinguishable from that where the user never existed to begin with. While DeCor uses a similar comparison, Garg et al.'s formalization assumes that users operate independently, and that the centralized data collector prevents one user's data from influencing another's.

Potentially other related works:

- Deceptive Deletions for protecting withdrawn posts
- Contextual Integrity
- ML Unlearning
- Database IFC (sensitive data flowing down edges?)

3 Design

DeCor models an application's schema as a graph of data *entities*. Each entity type corresponds to an application data table, such as stories, users, or votes. Entities are linked in the entity graph by foreign key relationships: table columns that act as foreign keys to other tables create child-parent relationships between entities. DeCor also includes abstract entities in the graph, where the keys may be non-referential identifiers that refer to abstract, non-table entities (e.g., a `thread_id` column in the comments table). Edges in the graph—foreign or abstract key relationships—represent correlations between the nodes of the graph, namely individual entities.

3.1 Specifying Decorrelation Policies

The developer provides a two-part specification to DeCor:

1. Ghost entity generation policies on entity types, and
2. Decorrelation policies for foreign/abstract key relationships specifying if and how instances of these edge types should be decorrelated.

The following describes the options for the different parts of the developer-provided specification in more detail.

3.1.1 Ghost Entity Generation

Developers can choose between the following ghost generation policies, which apply at per-column granularity.

- **Cloned:** All ghosts have the same value as the original entity for this column. Recorrelation returns an entity with the original value.
- **Generated:** All ghosts have a generated value for this column. Generated values are chosen to be random, a default value, or generated from the original value via a function provided by the developer. If the developer provides a reversible function (e.g., has encrypted the original value with a user-specific key), then the original value is restored upon recorrelation; otherwise, recorrelation returns a generated value in place of the original.

  If the column represents a foreign key into another table, then either the referenced table must have a ghost generation policy, or the foreign key is set to an existing entity or placeholder ghost (e.g., the foreign key is set to point to "deleted story"). It is up to the developer to ensure referential integrity in the latter case.
- **Single Clone, Generated Remainder:** One ghost has the same value as the original entity; all other ghosts have generated values. Recorrelation returns an entity with the original value.

3.1.2 Decorrelation Policies

Policy 1: Decorrelate. Application developers specify that the relationship may be decorrelated. DeCor generates a ghost parent entity for the child entity according to the parent entity's ghost generation policy, replacing the child's key with the new ghost parent key.

Resubscription (and recorrelation) removes any created ghost entities and restores the relationship back to the original parent key instance.

Policy 2: Do not decorrelate. Application developers specify that this foreign or abstract key relationship cannot be decorrelated. Developers select from the following subpolicy options:

- **Delete**: The child entity and its dependencies are deleted. This option should be selected if this type of

key relationship cannot be decorrelated while retaining application semantics, but keeping the relationship would reveal too much identifying information. Upon recorrelation, however, these entities cannot be restored.

- **Retain**: Nothing is done on either side of the relationship. This option should only be selected if the developer knows that the relationship cannot leak identifying information.
- **Achieve sensitivity threshold** $0 \leq \sigma \leq 1$: A sensitivity threshold for an foreign/abstract key relationships specifies the maximum proportion of edge instances of a relationship type that may connect to *sensitive* entities (i.e., entities transitively correlated to the initial entity being decorrelated).

  At a high level, the sensitivity threshold for a particular edge type estimates how much identifying information may leak from edge instances of that type. For any given edge type, developers can determine an appropriate sensitivity threshold by approximating how much identifying information may be leaked if edges of this type with the same parent key *all* correlate (even indirectly) back to the entity being decorrelated. In other words, what happens if all children of edges of this type (with the same parent) are sensitive?

  For example, consider the foreign key relationship from stories to tags. If all the stories tagged with the same parent tag in the entity graph were written by some (unsubscribed) user, would the story-tag correlation be problematic? The answer may be yes: perhaps tags are customizable by the user, and any story with that tag will clearly belong to the unsubscribed user. In other cases, the answer may be no: even though the tag is only correlated with sensitive stories, the tag indicates nothing about who may have authored the stories.

  For many cases, the answer may lie somewhere in the middle: it is problematic if *all* of children of edge of this type are sensitive, but perhaps it is acceptable if only a fraction of children of this edge type are sensitive. The maximum value for this fraction is the sensitivity threshold. For example, a reasonable sensitivity threshold might be $\sigma = 0.1$ for stories-tag key relationships: less than 10% of all stories with a specific tag key should have been correlated (even indirectly) with an (unsubscribed) user.

  DeCor either generates ghost children entities with this edge type until the sensitivity threshold is met (if a ghost generation policy is specified for the child type); upon resubscription, DeCor removes any generated ghost entities and edges. Note that if the generated ghosts are easily distinguished from actual entities, there is little privacy benefit from generating ghost entities to meet the threshold.

  If the children entity type has no associated ghost generation policy, DeCor removes sensitive children and their dependencies until the threshold is met. Upon recorrelation, however, these entities cannot be restored.

  If $\sigma = 0$, then this corresponds to removing all edges of this type with sensitive children (equivalent to a Delete policy); if $\sigma = 1$, DeCor does nothing (equivalent to a Retain policy).

3.1.3 DeCor's Execution Algorithm

Given this specification and an entity to be decorrelated as input, DeCor acts as follows:

1. **Parent-Child Traversal:** DeCor traverses the entity graph starting from the input entity, going down parent to child edges (and halting if it detects a cycle). As it traverses, DeCor collects the edges it has traversed.
2. **Parent-Child Decorrelation:** Post-traversal, DeCor acts on each edge instance according to the specified decorrelation relationship policy for that edge's type: if no policy is specified, DeCor does nothing. If edges can be decorrelated, DeCor generates ghost parent entities and new edges between child and ghost parent entities using the appropriate ghost entity generation policy. If edges cannot be decorrelated and should be retained or deleted, DeCor does nothing or removes the child and edge respectively.

   If there is a sensitivity threshold for the edge's type, DeCor ensures the sensitivity of the edge is below t's threshold, providing the edge type and the edge's parent key to the procedure described in Section 3.1.4.
3. **Child-Parent Decorrelation:** Next, DeCor takes the children of all traversed edge instances, and considers the set of edges from these children to other parents *not* traversed by DeCor during the decorrelation phase. (In other words, these children entities have multiple key relationships to several parent entities, one of which is connected via a chain of parent-child edges to the input entity).

   Intuitively, children of edges traversed by DeCor share a connection with the initial entity being decorrelated. Edges *from* these children to other parent entities may thus leak sensitive identifying information.

   DeCor acts on these edges according to the specified decorrelation relationship policy for each edge's type.

If these edges can be decorrelated, DeCor generates ghost parent entities for each sensitive child. If these edges cannot be decorrelated and should be retained or deleted, DeCor does nothing or removes the child and edge respectively.

Otherwise, if there is a sensitivity threshold for the edge type, then DeCor limits the proportion of edges of that type that connect to sensitive entities (the children of traversed edge instances) to below the threshold. For each edge with type t in this set of edges, DeCor ensures the sensitivity of the edge is below t's sensitivity threshold, providing the edge type and the edge's parent key to the procedure described in Section 3.1.4.

DeCor optionally allows developers to specify that edges have *weaker* decorrelation policies in the child-to-parent direction than from parent-to-child: this allows expression of policies where it is safe to retain links if *only the child* is sensitive, but where the link should be decorrelated, removed, or desensitized if *both* the child and parent are sensitive. For example, perhaps a user wants to ensure that their link to sent messages are decorrelated, but links from the message to the recipient users can still be retained.

An example of these three decorrelation steps is shown in Figure 3.1.4.

3.1.4 Achieving the Sensitivity Threshold

Let E be the subset of edges traversed by DeCor in Step 1 of execution. Given an edge type t with sensitivity threshold σ_t, and a parent key k of an instance of edge type t,

- DeCor computes $N_{sensitive}$, the number of edges of type t with parent k in the entity graph that share a child node with edges in E.
- DeCor computes N_{all}, the total number of edges of type t with parent k in the entity graph of edge.
- DeCor computes $N_{sensitive}/N_{all}$, the *sensitivity* of edges of type t with parent k.
- If the sensitivity exceeds σ_t and the child entity type has an associated ghost entity policy, DeCor generates ghost children and edges of type t from these children to parent key k. This lowers the sensitivity by increasing N_{all}. Note that this may also create other ghost parents for the generated ghost children if these children have more than one column representing a foreign key relationship.
- Otherwise, if the sensitivity exceeds σ_t but ghosts cannot be generated, DeCor removes the children of edges in E_t with parent k, thus lowering the sensitivity by lowering $N_{sensitive}$.

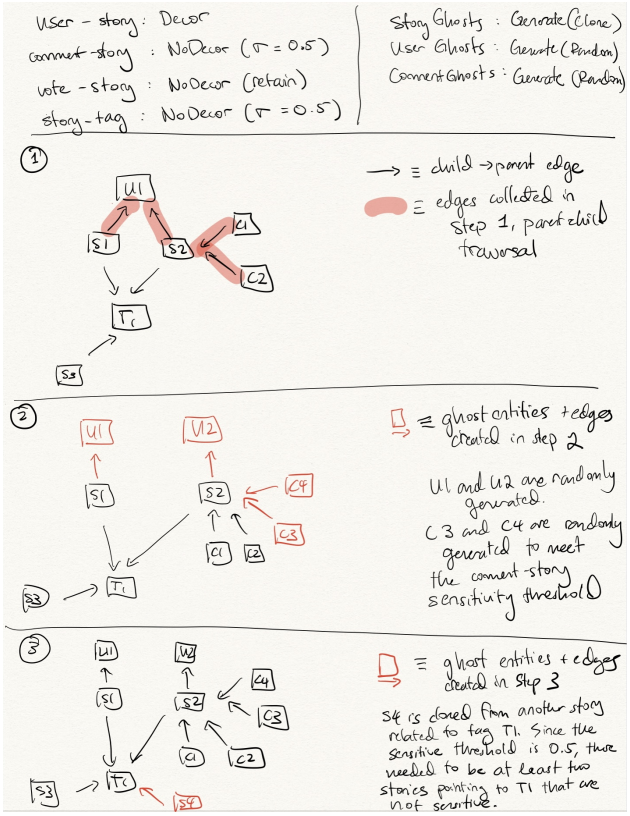

Figure 1: Examples of DeCor's execution.

Note that the initial sensitivity for an edge of type t with parent k from Step 2 (parent-child decorrelation) is 1! Because DeCor traverses from parent to child, if one edge of type t with parent k was collected by DeCor, then *all* edges of type t with parent k were collected by DeCor.

However, the initial sensitivity for an edge of type t with parent k from Step 3 (child-parent decorrelation) may be very low: other parents of children touched by DeCor may have many non-sensitive children (e.g., a tag may have many stories not authored by the user being decorrelated).

4 Implementation

DeCor introduces in-memory shim layer on top of a MySQL database, allowing DeCor to intercept and process queries sent by the application frontend. To amortize the cost of unsubscription and resubscription, DeCor preemptively creates, stores, and links ghost parent entities to child entities if an update creates an edge that may be decorrelated. DeCor builds in-memory materialized views

```
pub enum GeneratePolicy {
    Random,
    Default,
    Custom(FnMut(ColumnValue) -> ColumnValue),
    ForeignKey,
}
pub enum GhostColumnPolicy {
    CloneAll,
    CloneOne(gp: GeneratePolicy),
    Generate(gp: GeneratePolicy),
}
pub type GhostPolicy;
   HashMap<Column, GhostColumnPolicy>;
pub type EntityGhostPolicies =
   HashMap<EntityName, GhostPolicy>;

pub enum DecorrelationPolicy {
    NoDecorRemove,
    NoDecorRetain,
    NoDecorSensitivity(f64),
    Decor,
}
pub struct KeyRelationship {
    child: EntityName,
    parent: EntityName,
    column_name: String,
    parent_child_decor_policy: DecorrelationPolicy,
    child_parent_decor_policy: DecorrelationPolicy,
}
pub struct ApplicationPolicy {
    entity_type_to_decorrelate: EntityName,
    ghost_policies: EntityGhostPolicies,
    edge_policies: Vec<KeyRelationship>,
}
```

Figure 2: DeCor-provided types to specify key relationships and ghost generation policies.

on top of the underlying database, exposing ghost entities only if the true entity has been decorrelated, and real entities otherwise. Updates propagate to the materialized views when the underlying database is updated. DeCor answers application queries using these materialized views, hiding the complexity of ghost entity and decorrelation management.

DeCor is build in 5K LoC of Rust, and supports a subset of MySQL. DeCor adds `UNSUBSCRIBE [user]` and `RESUBSCRIBE [user] WITH [user_data]` calls to the query language. [lyt: Should I mention details like query parsing, single-threaded?] Application developers specify and provide an `ApplicationPolicy` using the objects shown in Figure 4 [lyt: Why is this showing the wrong figure number??].

4.1 Data Objects

In addition to the original application data tables, DeCor stores a persistent mapping from entity ID (EID) to a set of ghost IDs (GID). This mapping is cached in the shim layer for performance. Only ghost entities that correspond to a true parent entity are stored in the mapping: auxiliary ghosts that are created to introduce noise (reduce sensitivity), or to satisfy referential integrity—all their children are also ghosts—are not linked to the real parent EID.

DeCor stores the in-memory materialized views for each datatable, using hashtables and btrees for table indexes. An in-memory cache of the graph of parent-child entity key relationships built on top of the views is also stored for unsubscription and resubscription performance.

4.2 Handling Normal Execution Queries

Reads. Reads are answered by the materialized views. Because the materialized views are kept up-to-date with any application writes, no queries to the underlying datatables are performed. [lyt: I'm not sure if it's appropriate or interesting to put some of the MV query handling stuff here? (Like processing joins, etc.)]

Inserts. When the application inserts an entity, DeCor first checks (using the developer-provided policy) whether the entity is a child entity (i.e., it has a foreign key to another data table). and whether the policy specifies that this child-parent key relationship should be decorrelated during unsubscription.

If so, DeCor preemptively creates a *ghost parent* entity using the appropriate entity generation policy, and adds this ghost to the parent data table. DeCor then saves the child entity's real parent EID (the child entity's foreign key column value), and adds a mapping from this EID to the new ghost parent GID.[2] Finally, DeCor rewrites the child entity's foreign key reference to correspond to this new ghost parent's GID, and stores the child into the datatable.

Note that generating the ghost parent may recursively generate a set of ghost ancestors for that parent as well (to satisfy referential integrity). These ancestors are also added to the parent data table.

The insert then propagates to the materialized view, inserting the entity with its real parent EID; queries that select for this entity will not see any ghost entities.

If the child-parent key relationship should not be decorrelated, or the inserted entity is not a child, DeCor simply inserts the entity as-is into the datatable, and the materialized view is updated with the entity.

DeCor also updates the parent-child entity graph with the new entity if the entity adds any edges specified by the developer-provided policy. This graph is built on top of the materialized view, and thus contains only those entities exposed by the materialized view to the application.

Updates. When the application updates an entity, DeCor again checks (using the developer-provided policy) whether the entity is a child entity (i.e., it has a foreign

[2] DeCor assumes that the application does not violate referential integrity, namely that any child entity inserted into the system will refer to a corresponding parent in the database.

key to another data table). If the entity is a child, and the policy specifies that this child-parent key relationship should be decorrelated during unsubscription, then DeCor performs one final check of whether the entity's parent key is one of the columns being updated.

If yes, then DeCor gets the current value of the entity parent key in the datatable: because DeCor inserts only GIDs into the datatable for these decorrelatable child-parent links, this value corresponds to some ghost parent GID. DeCor checks its ghosts mapping table for which real parent EID currently maps to this GID, removes this current mapping, and updates the mapping so that the updated parent key value now points to this GID. DeCor updates all non-foreign key columns as normal, with values specified by the application.

If the updated entity is not a child of a decorrelatable edge, the update is performed without any alterations.

Updates are propagated to the materialized view without modifying the values, so that application reads only observe real parent entities. DeCor also updates the parent-child entity graph if any edges between materialized view entities change.

Deletes. When the application deletes an entity, DeCor checks whether the entity is a child entity (i.e., it has a foreign key to another data table). If the entity is a child, and the policy specifies that this child-parent key relationship should be decorrelated during unsubscription, then DeCor gets the current value of the entity parent key in the datatable, which must correspond to some ghost parent GID. DeCor checks its ghosts mapping table for which real parent EID currently maps to this GID and removes this mapping. DeCor also removes the ghost parent with this GID.

In all cases, DeCor removes the entity being deleted. The deletion is propagated to the materialized view and removes the entity being deleted; the parent-child entity graph is modified appropriately.

4.3 Handling Unsubscription

Traversal. DeCor is given the top-level entity type and EID to be decorrelated (typically a user and a user ID). DeCor then performs a (depth-first) traversal of the parent-child entity graph built on top of the materialized views.

DeCor stores all the children entities traversed during decorrelation for post-processing, and handles parent-child edges as follows:

If a parent-child edge has a no-decorrelation-retain policy, DeCor does nothing; if a parent child-edge has a no-decorrelation-remove policy, DeCor removes the child entity and recursively removes all of its descendants.

For every parent-child edge with a decorrelate policy, DeCor looks up the parent EID in the EID to GIDs mapping, saves the corresponding GIDs, and removes the mapping (both persistent and cached). DeCor then updates the child entity in the materialized view to point to a ghost entity instead than the real parent (who is being decorrelated) by choosing one of the corresponding GIDs for this parent and updating the child's parent key with this GID. The parent-child entity graph is updated accordingly to include the ghost parent (note that this does not affect the traversal, which only traverses parent-child edges).

If a parent-child edge has a no-decorrelation-sensitivity policy, DeCor stores the edge for post-processing.

Sensitive Parent-Child Edges. Post-traversal, DeCor considers its collected set of sensitive edges. For every unique parent, DeCor counts how many children attach to the parent for each distinct edge type (e.g, for a particular tag, how many stories are linked to this tag, how many comments are linked to this tag, etc.). For each child count n, and with an edge type sensitivity threshold of σ, DeCor generates n/σ ghost entities of that child type, all of which have an edge to the parent. These ghost entities are inserted both into the materialized views, and into the underlying datatables; however, because they do not correspond to real entities, EID to GID mappings are not stored. If DeCor finds no ghost generation policy for the child entity type, DeCor instead recursively removes the child and its descendants.

Handling Child-Parent Edges. Next, DeCor considers child-parent edges from the collected set of traversed children to parents not seen during DeCor's traversal. If an edge can be decorrelated, DeCor rewrites the materialized child parent key to one of the parent ghost GIDs. The ghost GID is retrieved by querying the parent's EID to GIDs mapping, the corresponding ghost parent is added to the materialized view, and the mapping from parent EID to GID is subsequently removed. These steps ensure that the materialized view reflects that the link between the child and the original parent has been decorrelated.

If an edge has a remove policy, the child and its descendants are removed; if an edge has a retain policy, nothing is done. If an edge e has a senstivity threshold, DeCor counts N_{all}, the total number of edges of the same edge type as e that connect to e's parent. DeCor then counts $N_{sensitive}$, the number of these edges that connect to children that were traversed by DeCor during the first decorrelation step (this number includes e). If e's edge type has sensitivity threshold of σ and $N_{sensitive}/N_{all} > \sigma$, DeCor generates $\lceil N_{sensitive} * \sigma \rceil - N_{all}$ ghost entities of e's child type, all of which have an edge to e's parent. As before, these ghost entities are inserted both into the materialized views, and into the underlying datatables; however, because they do not correspond to real entities, EID to GID mappings are not stored. If DeCor finds no

ghost generation policy for the child entity type, DeCor instead recursively removes the child and its descendants.

Decorrelated Entity Removal and Return Values. Finally, DeCor removes any completely decorrelated entities, namely those children that have no descendants and are linked to only ghost (or no) parents. DeCor returns to the unsubscribing user (1) the removed parent EID to GIDs mappings, (2) any completely decorrelated (and removed) entities, and (3) the GIDs of any additional generated ghosts during sensitivity checks. To ensure that the end user cannot tamper with their application data while unsubscribed, DeCor saves the hash of the top-level entity EID and the returned data to check upon resubscription. [lyt: This is currently in-memory; should I persist it? (Will add another db query, but unsubscribe is already pretty expensive)]

4.4 Handling Resubscription

DeCor is given the top-level entity type and EID to be resubscribed, along with the data returned from a prior unsubscription. DeCor verifies that the hash of the EID and data matches that from the latest unsubscription of this entity.

DeCor first re-inserts any completely decorrelated entities into both the datatables and materialized views.

Next, for all removed parent EID to GIDs mappings, DeCor reinserts these mappings into the persistent and cached EID to GIDs mapping. DeCor then removes the corresponding ghost entities for all GIDs and any ghost ancestors of these GIDS from the materialized views; note that these ghosts still remain present in the persistent data tables for future unsubscription. DeCor then updates all materialized children of these removed ghosts with the original parent EID. Looking up the relevant children is facilitated by the in-memory entity graph.

Finally, DeCor removes any ghost entities from the datatables and materialized views that were created during unsubscription to lower sensitivity.

Consistency and Recovering the Materialized Views [lyt: This isn't actually implemented yet...] Upon a crash, the in-memory materialized views are rebuilt from the underlying datatables and ghost mapping. Non-children datatable entities are simply reinserted into the appropriate materialized view.

If a datatable contains child entities, and these children have ghost parent GIDs as foreign keys (because their child-parent edges can be decorrelated), then DeCor checks, for each child, whether a mapping from a real parent EID to this entity's foreign key GID exists in the persistent EID to GIDs mapping. If such a mapping exists, then this edge has not yet been decorrelated, and the real parent entity should be exposed to the application: DeCor inserts the child entity with the real parent EID into the materialized view.

If the mapping does not exist, then the link to the real parent entity must have been decorrelated at some point: DeCor inserts the unmodified child entity with the GID as foreign key into the materialized view. Note that this may link a child entity with a *different* ghost user than initially decided during unsubscription (where any one of the ghosts corresponding to the real parent could be used as a ghost parent for any child). [lyt: I think this can also be avoided by updating the MV entities with GIDs in monotonically increasing order, since this is how they would have been inserted into the underlying database. It also doesn't seem that important.]

The parent-child graph is constructed appropriately on top of the materialized view, and the EID to GIDs mapping cache repopulated.

DeCor does not currently support atomic unsubscription, resubscription, or recovery behavior. [lyt: TODO]

5 Evaluation

We evaluate DeCor on two metrics: (1) can desired decorrelation policies be easily specified using the provided decorrelation primitives for a range of practical applications?, and (2) can decorrelation be supported with low performance overhead?

Our evaluation exemplifies how a suite of applications (Lobste.rs, [lyt: TODO]) can express a diverse range of privacy policies using DeCor with low developer effort with low overhead.

5.1 Example policies

We provide several variants of an application policy for Lobsters, as well as example policies for a suite of other applications: (1) HotCRP, software for conference reviews; (2) PrestaShop, an open-source e-commerce web application; and (3) an Instagram clone with posts, likes, hashtags, shares, and groups.

Lobsters Lobsters has a total of 20 entities, and these Lobsters policies require specifying at most 17 key relationships and at most 5 ghost entity generation policies (see Figure 5.1 for an example of these policy specifications).

The first policy replicates what the behavior of the current Lobsters deployment upon user unsubscription, with the exception that users are replaced with ghost users rather than a global placeholder. This requires specifying `Decorrelate` policies for all edges from entities with users as parents, resulting in the developer writing 9 edge

```
use GhostColumnPolicy, GeneratePolicy;
let ghost_policies = GhostGenerationPolicy::new(
  ("users",
     [("id", Generate(Random)),
     ("username", Generate(Random)),
     ("karma", Generate(Default(0)))]),
  ...);

let edge_policies = vec![
  KeyRelationship{
    child: "stories".to_string(),
    parent: "users".to_string(),
    column_name: "user_id".to_string(),
    parent_child_decorrelation_policy: Decor,
    child_parent_decorrelation_policy: NoDecorRetain,
  },
  KeyRelationship{
    child: "taggings".to_string(),
    parent: "tags".to_string(),
    column_name: "tag_id".to_string(),
    parent_child_decorrelation_policy:
        NoDecorSensitivity(0.25),
    child_parent_decorrelation_policy: NoDecorRetain,
  },
  ...];

 ApplicationPolicy {
    entity_type_to_decorrelate: "users",
    ghost_policies: ghost_policies,
    edge_policies: edge_policies,
 }
```

Figure 3: Excerpt of the Lobsters Application Policy

policies and one ghost generation policy (for ghost users). While this policy does nothing significant, splitting a user into multiple ghosts as compared to a global placeholder allows the user to resubscribe by identifying their ghost counterparts, whereas a global placeholder does not. Furthermore, splitting users into multiple users as compared to grouping them together upon unsubscription may benefit privacy: global placeholder users indicate that any child of the placeholder belongs to some deleted user, whereas ghost users may more successfully mask which children have been abandoned.

The second policy decorrelates all user-related edges, but additionally desensitizes tags so the taggings associated with sensitive stories are at most 25% the number of stories with that tag. Ghost stories clone randomly chosen existing stories with slight variations on the url. This limits the amount that an observer can determine that a particular story was posted by a particular user simply by knowing that a certain user has posted about the tag topic (e.g., a researcher in visualization). However, Lobsters still retains the content of these tagged stories.

[lyt: TODO] third policy.

HotCRP. HotCRP entities include users (reviewers and paper authors), papers, and paper reviews. There are a total of 25 entities, with 47 foreign key relationships between entities; specifying these key relationships (as a non-maintainer of the HotCRP site) took fewer than 20 minutes.

HotCRP's current privacy policy [6] allows site managers (e.g., program chairs) to indefinitely store and distribute submissions and reviews. Each HotCRP.com user has an associated global profile and a profile for every HotCRP.com site, and must contact the HotCRP maintainers directly to remove these profiles. This corresponds to key relationships with `NoDecorrelate:Retain` policies: a user simply cannot unsubscribe (or resubscribe).

With DeCor, we can express many more granular privacy policies that would otherwise be difficult to implement right in HotCRP. The top-level entity to decorrelate are user profiles: reviewers may want to remove their connections with their reviews, papers, and other metadata (such as papers they have requested to review). However, review and paper artifacts should remain identifiable and present, and correctly linked: (subscribed) reviewers should still see the correct paper for each of their reviews, and authors should see the correct reivews for their papers.

With DeCor, the HotCRP developer can specify that decorrelating a user decorrelates all edges with the user as parent, replacing reviewers (or authors) with ghost user profiles. This goes beyond pseudonymization: these reviews or papers are no longer grouped together by any (even anonymized) identity. Furthermore, papers by the user can be decorrelated from their reviews, comments, and conflicts, all of which in aggregate may reveal information about the unsubscribing user, or information that the user may wish to keep private (such as the total review scores for the paper). To maintain semantic correctness, paper entities have a `CloneAll` ghost generation policy: this ensures each child review or comment still has a semantic connection to their parent paper, and a HotCRP reviewer always observes the correct paper for any of their reviews, even if the paper is a ghost of the original.

PrestaShop. PrestaShop entities include customers, products, orders, carriers, employees, and shops (as well as many other entities such as product categories, languages, countries, shopping cart, etc.) There are 214 total different entities.

Due to the large scope of the schema, we automatically generate policies for the 324 key relationships between them with `NoDecorrelation:Retain` policies, which means that DeCor does, by default, no work upon decorrelation and recorrelation. (We could also imagine that a `Decorrelate` or `NoDecorrelate:Remove` policy could be the default here, but we choose a policy by default that will not break application semantics, instead of a policy that provides the strongest possible unsubscription privacy).

By simply providing a ghost customer generation policy and modifying all key relationships with customer parents to `Decorrelate` or `Remove` policies, PrestaShop can ensure that customer information is

appropriately de-identified when a customer unsubscribes. Currently, PrestaShop has explicit GDPR policies for its merchants when a customer requests to delete their account: their personal account details (age, email, address) are removed, but order invoices and abandoned carts are transferred to an anonymous account [lyt: TODO https://doc.prestashop.com/display/PS17/Complying+with+the+GDPR]. DeCor allows PrestaShop to go beyond pseudonymity, creating individual (ghost) accounts for each cart and order invoices of a customer, and also allowing a customer to retrieve these orders and abandoned carts if she rejoins the application.

DeCor also can help PrestaShop keep attributes such as the country or language of a user for statistical analyses (used by merchants to analyze product popularity). Ghost customers can generate links to countries nearby a set of countries similar to the original user; if this still reveals too much sensitive information, the location can be desensitized (adding additional generated ghost customers not related to the original user for noise).

This process requires going through each of the individual key relationships, which must be done to generate the schema in the first place.

Instagram Clone.

5.2 Performance

Experimental Setup. The performance experiments reported are run on Intel Xeon E5-2660 v3 CPUs, with a single-threaded client and DeCor's shim layer each pinned to a single core. DeCor runs on top of MariaDB (v10.4.13).

We compare the application execution performance on four systems, all of which configure MariaDB to store the database on a ramdisk to avoid disk IO overheads.

1. `NoShim`: queries are directly issued to a MariaDB instance;
2. `Shim`: queries are intercepted by the shim layer, which does nothing more than send queries to the MariaDB instance and return the results back to the client;
3. `ShimParsing`: queries are intercepted the shim layer, parsed into a SQL AST (which is discarded), and then sent to the MariaDB instance; the shim then returns the results to the client.
4. `DeCor`: queries are intercepted by the shim layer and parsed into a SQL AST. DeCor then processes the parsed query, potentially introducing ghost entities and sending additional queries to the MariaDB backend. DeCor then returns results to the client.

We compare `NoShim` and `Shim` to define the cost of query interception; `Shim` and `ShimParsing` to define the cost of query parsing; and `ShimParsing` and `DeCor` to define the cost of adding decorrelation and recorrelation support. `Shim` provides an upper bound for DeCor's (single-threaded) performance, as the costs of query parsing and decorrelation are not necessarily fundamental.

Lobste.rs Performance DeCor utilizes the trawler workload [5] for Lobsters, which emulates production Lobsters traffic according to a recorded production workload. The database initially begins with 3K users, 20K stories, and 60K comments (1/2 the size of the the real Lobsters deployment), and issues queries according to the recorded traffic distribution. The query distribution skews towards reads (recent and frontpage stories) over writes (votes, commenting, and posting of stories); the workload assumes that users with popular content are also more active and likely to post stories, comment, and vote.

The benchmarks perform 10000 actions, each performing on average 8 application queries. We measure (1) overall throughput, (2) the distribution of application query latencies, (3) the query multiplication per application query performed by DeCor, and (4) the storage cost (in-memory and on-disk).

The first benchmark measures DeCor's performance during *normal execution*, namely when only read and update actions performed by Lobsters are executed. The second benchmark measures DeCor's performance with users unsubscribing 10% of the time (approximately 1000 actions are unsubscriptions). If an unsubscribed user is chosen to perform an action, this user is first resubscribed. `NoShim`, `Shim`, and `ShimParsing` simply delete or insert the chosen user on unsubscription or resubscription respectively; this provides a very conservative estimate of the amount of work these baseline systems would need to do if a user withdrew their data from the system.

[lyt: Currently debugging the benchmark... (some error with repeat keys, hopefully won't take too long)] To complete 10K actions during normal execution, `NoShim` takes 293s, `ShimOnly` takes TODO, `ShimParse` takes 316s; and `DeCor` takes 185s. We show the latency results of the second of the Lobsters policies described above in Figure 5.2. [lyt: TODO memory usage] [lyt: Should we also record the amount of time to initialize?]

To complete 10K actions with 10% of actions being unsubscribe, `NoShim` takes 263s, `ShimOnly` takes 277s, `ShimParse` takes 284s; and `DeCor` takes TODO. [lyt: TODO memory usage]

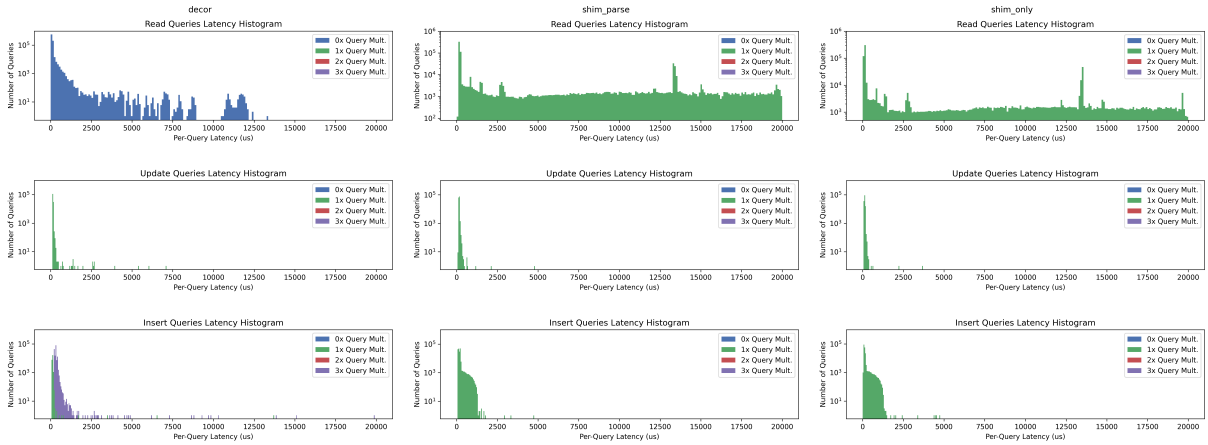

Figure 4: Comparison of DeCor's latency during normal execution.

6 Discussion and Future Work

6.1 Limitations

single-threaded; recovery / eventual consistency during un/resubscribe; subset of MySql;

6.2 Maintaining Aggregate Accuracy.

DeCor optionally allows for entities to be decorrelated without affecting queries which specifically return aggregation results.

Queries that specifically perform aggregations and return statistical measures (e.g., the count of number of users in the system, or the number of stories per user), can return significantly different results. This affects the utility of the data for the application: for example, if the application relies on the number of stories per tag to determine hot topics, these would be heavily changed if ghost tags were created. In addition, the adversary may learn which entities are ghosts: for example, an abnormally low count of stories per tag might indicate to an adversary that these tags are ghost tags. [lyt: But perhaps it's ok if an adversary can tell what's a ghost, as long as it can't tell which user each ghost is correlated with.]

DeCor stores and separately updates answers to aggregation queries; these answers are updated when queries update the data tables, and these queries do not read from the application tables (which may contain ghost records).

An alternate solution might analyze the aggregations performed by application queries, and then introduce ghosts that lead to the same (or close-enough) aggregation result. For example, if a tag is split into ghost tags, one per story associated with the tag, but the application still would like the count of stories for this tag to be high, one of the ghost tags can be populated with many ghost stories to retain the count of stories per tag. DeCor would remove any ghost stories that were created upon recorrelation. Note that this solution 1) requires that generating ghosts is admissible, and 2) may be impossible for certain combinations of aggregations (e.g., queries that return both the average stories per tag and also the total number of stories).

[lyt: I don't *think* differential privacy really should be applied here, because we'd also face the issue of running out of privacy budget. Adding noise might ensure that the impact of any one (real/ghost) user is very little, but it has its own noise/utility tradeoff. Furthermore, the amount of noise necessary if many ghost users are created might be too large.]

[lyt: Note that the application can also store this information if it wants...]

References

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

[2] Katriel Cohn-Gordon, Georgios Damaskinos, Divino Neto, Joshi Cordova, Benoi't Reitz, Benjamin Strahs, Daniel Obenshain, Paul Pearce, and Ioannis Papagiannis. "DELF: Safeguarding deletion correctness in Online Social Networks". In: *29th USENIX Security Symposium (USENIX Security 20)*. USENIX Association, Aug. 2020.

[3] "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.

[4] Sanjam Garg, Shafi Goldwasser, and Prashant Nalini Vasudevan. "Formalizing Data Deletion in the Context of the Right to be Forgotten". In: *Annual International Conference on the Theory and Applications of Cryptographic Techniques*. Springer. 2020, pages 373–402.

[5] Jon Gjengset. *Trawler*. Version 0.10.2. URL: https://github.com/jonhoo/trawler.

[6] Eddie Kohler. *HotCRP.com privacy policy*. URL: https://hotcrp.com/privacy.

[7] Frank McSherry. "Privacy Integrated Queries". In: *Communications of the ACM* 53 (Sept. 2010), pages 89–97.