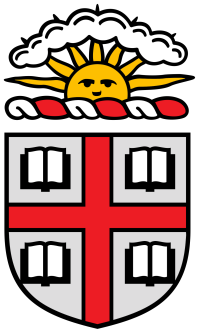
Funhouse: A Hall of Mirrors Database

Hannah Gross

Advisor: Malte Schwarzkopf

Reader: Shriram Krishnamurthi



Department of Computer Science

Brown University

Providence, RI

April 2023

Contents

Acknowledgements

Many people contributed to this work - as well as to my time here at Brown - but I would specifically like to mention:

- my rommates, who showed me just how joyful life can be;
- the ETOS lab group, including Lily, for their support, patience, and good vibes;
- and Malte, for being the best mentor and friend a guy could ever ask for.

I am so grateful to all of you ♡

Abstract

Organizations need to give insiders access to sensitive customer data, but compromised insiders with broad access are the reason for many damaging data leaks. Ideally, access control would limit insider access to a minimum, while still letting people do their jobs. Access control systems exist to determine a user's permission for a resource, and there has been much research about how to incorporate context and environment into those decisions (e.g. attribute based access control). But these systems deny access if a query might leak sensitive data, leaving applications to deal with denied queries, and ensure that queries only touch data the querier is allowed to see. This thesis explores a way of integrating flexible and contextual access into a model where every query can be answered (albeit perhaps with redacted data).

We sketch the design of a new kind of database that by construction redacts data. It shows employees limited views of the data by default, but allows them to flexibly upgrade their access as needed. Our design combines redactions over data with capability-based authorization to temporarily upgrade an insider's level of access. A prototype demonstrates that such a design could be practical.

1 Introduction

1.1 Problem

Fine-grained access control in databases is hard, since it trades off better security through tighter permissions against the ability to flexibly query data. Database administrators often choose flexibility when faced with this tradeoff, which results in overly broad permissions. This risks serious data leaks, as compromised employees or third party accounts typically have access to lots of personal customer information. For example, a single employee's account at Starwood Hotels leaked 5.2M guests' information including passport numbers and payments [18]. Similar leaks have occurred at retail companies, Uber, Twitter, and even cybersecurity companies [5, 25, 30, 31].

We believe that overpermissioning arises because employees' data access needs are dynamic. Who needs access to what constantly changes, and access control systems must adapt accordingly. For example, customer service representatives at a hotel must access bookings and content moderators must access social posts. However, at any given point in time, they will need access to only one or two bookings or posts, not all of them. And which exact bookings or posts they need access to can be determined based on their workflow context: what causes the access (e.g., a customer phone call or a flagged post) can tell us what the employee is working on and thus what data they need access to.

In a perfect world, every employee has access to exactly the data they need at any given moment in time, no more and no less. In this world, a database access control system would meet both an organization's needs to restrict access to user data in order to protect their customers, and insiders' need to access data to do their job.

Much research has been done around making access control dynamic, and a lot of it is centered around making Access Control Lists (ACLs) more aware of the context/environment in which they are invoked [12, 16, 33]. Although we believe that this is moving in the right direction, we note that this sort of "gatekeeper" of resources model is limited in the actions

it can take: it can either allow access, or deny it, forcing applications to deal with denied queries. In the extreme this may lead to a crashed application, and will in any case lead to a disruption in the current workflow. Early on during the scoping of the problem we wanted to address, we made the decision that we want to go beyond this binary, and the systems we build should be able to answer every query. This means that decisions about who has access to what might have an answer of "yes but with some modifications". In other words, our design is trying to unify two dimensions of flexibility: support for constantly changing permissions, and support for a spectrum of access levels that go beyond having access or not.

In its simplest form, redacting data that is returned could mean just removing sensitive rows/columns. However, this is not far removed from simply denying access to certain resources. We want to build a system that supports a broader range of data redaction policies that match practical use cases.

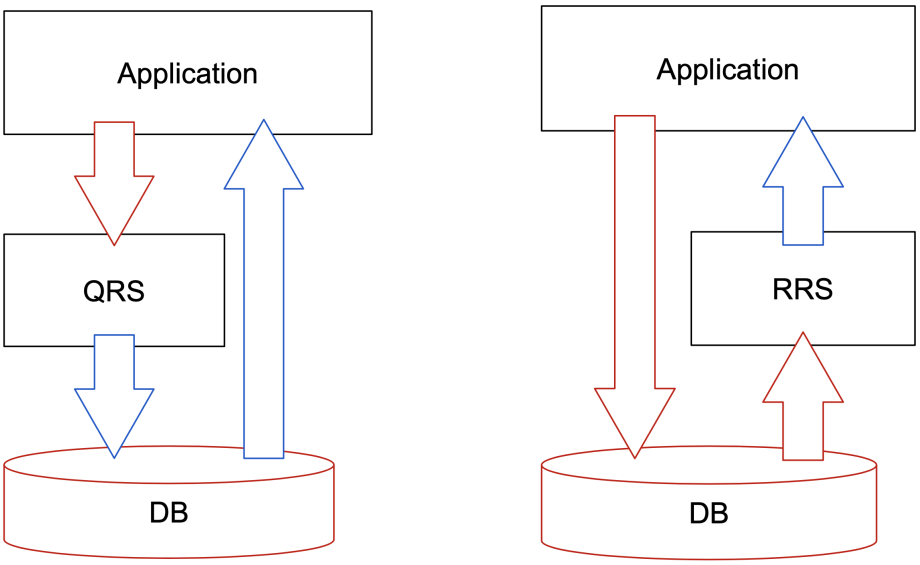
For example, it could allow modifications of data, so that, rather than getting an empty credit card table, an employee might get one back with random numbers. This also allows for more redactions to be feasible: developers, for example, need access to realistic data in order to see what users might see, but don't actually need access to the sensitive information.

Another example is that correlations might need to be hidden – for example between users and posts, between guests and bookings, or between addresses and names. In order to preserve the data while hiding the correlations, one could imagine a type of redaction that creates fake new entities (e.g., users) and re-associates data to them (e.g., addresses).

1.2 Background

The setup that this thesis addresses is the following: there is an application back-end, through which different users access data, and then there is a database, which contains data including sensitive information. The goal is to place into this setup a system whose purpose is to apply redactions to data before it reaches the application.

We first consider a design in which this new system is separate from the database, and



(a) Query Rewriting System

(b) Result Rewriting System

Figure 1: (a): a query rewriting system: red (potentially dangerous) queries go in, are rewritten to take into account/additionally apply redactions, and run on a database with sensitive information, returning a redacted result;
(b): a result rewriting system: red (potentially dangerous) queries are run on a database with sensitive information, the result is checked and redactions are applied, returning a redacted result

acts as a proxy to it. Such a design has the advantage of being deployable as an add-on to current databases. Given this constraint, there are two places where enforcement can happen.

The first is that it could be a query rewriting system (QRS), which edits a query before it is even run on the database. The rewritten query then itself performs the redactions in the database and returns redacted data (Figure 1a). There are existing systems that do this, e.g., Qapla [21], but in this section we consider query rewriting systems in general.

A QRS needs to be able to interpret the SQL enough to understand what the query is asking for, and then potentially rewrite the query to avoid reading sensitive data (or, in service of more versatile redactions, might modify or add data). Understanding the query well enough to do this necessarily means reasoning about the query and data in the database:

it requires the QRS to parse the SQL string to understand what tables and columns are accessed as well as what rows are contained in the result; on top of which the QRS must then rewrite the query to implement the desired redactions. On top of this complicated reasoning, the result is a more complex query that is likely to take longer to execute (because it does additional work of removing/modifying/adding data).

The second place that enforcement could happen is in a result rewriting system (RRS), which redacts the results of the query - e.g., when adding noise in systems that offer differential privacy (DP) guarantees [17, 20, 24]. By applying the redaction to the query result, the query execution itself happens over the raw data and is independent of the redactions (Figure 1b). A RRS, similarly to a QRS, also needs to reason about the query, but in a more indirect way: the RRS needs to be able to reverse-engineer where the data came from, and then understand how to post-hoc apply redactions. This may be simple for some types of redactions (e.g., adding noise to aggregations), but will be more complicated for others (e.g., modifying credit card numbers, or creating new users and re-associating bookings with them). Thus, for both QRS and RRS, we come to the conclusion that a system supporting a wide definition of redactions would end up needing to reason about the query execution process. This is undesirable: the QRS/RRS would need to understand query semantics and take into account data-combining operations like joins and aggregations, which duplicates functionality that already exists in the database and introduces complexity and potential for mistakes.

In order to explore further options, we let go of a previously held assumption: that the data-redacting system is not part of the database. If it is integrated with the database, there are now two further options available: data redaction can change the data stored, or it can change the process of how queries are executed. We will find that the former is promising, but leads to undesirable levels of complexity, and from there derive our approach (§3), which changes how queries are executed. Given an approach that changes the data that is stored, a strawman approach would make a copy of the database for each potential querier, and each

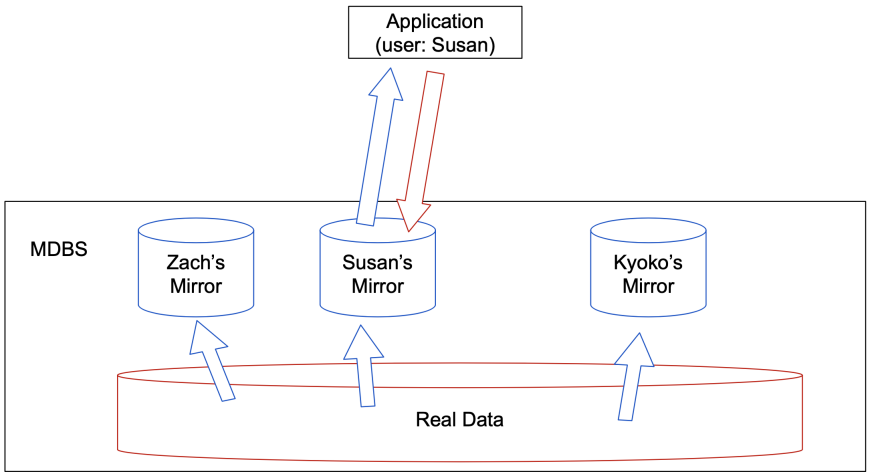


Figure 2: MDBS now changes the database contents

copy contains the information that querier is allowed to see. In this case, all queries from employee Susan run on Susan's mirror (Figure2), which contains a copy of the database appropriately redacted for Susan.

Broadly, this approach of creating a multi-database system (MDBS) is seen in e.g., MultiverseDB [19]. And creating a system that at the very least simulates this reality is desirable: it is conceptually simple, and obviously correct.

The two obvious problems with MDBS are (1) its space complexity, and (2) the fact that changing permissions would require effectively re-deriving the contents of the entire database (to re-create the copy, with whatever permission changes reflected in the data now contained in that copy).

The space issue could be mitigated by creating a limited number of mirrors of the database (as in, materialized copies of the database) that hold the modified (and therefore freely query-able) data. As a base case, the system could return the results from just running on that mirror. Changing permissions could then be encapsulated in just the deltas to the baseline mirror of the database.

Realizing this is where the approach becomes untenable, consider our assumption that users' context and permissions frequently change. Imagine the following scenario: as a

baseline, a customer service representative (CSR) at a hotel does not have access to guests' booking information (i.e. they are removed in the mirror). Now, they are on the phone with a guest, Caroline, so they temporarily have higher permissions where Caroline's bookings are revealed to them. Creating a whole new mirror of the database is impractical, since we expect changes in permissions to be a common occurrence: on the next phone call, the CSR would need a new mirror that contains the new guest's bookings (and no longer contains Caroline's).

Storing just the changes in data imposes less space overhead, which in this case would mean storing the rows that the temporary permissions revealed. The rows would need to be added to any results for this user. If a query touches just the bookings table, the rows can be appended to the result – although this must into account the queries filters, projections, or aggregations, which then becomes a problem of incremental query execution [3, 14]. This model breaks down even more on joins: what if the user now wants to join the table where they have an extra row with some other table? We would have to now entirely re-run the query to get any joined data for the newly visible row.

In summary, this model works well for straightforward querying over redacted data, but supporting changing permissions seems to require both computational and intellectual complexity that is unsatisfying (and, as it turns out, unnecessary).

1.3 Our Approach

The Funhouse database [1], centers around three core principles: limited default access, which respects the principle of least privilege [26]; temporarily upgrading access to capture context; and putting the machinery for enforcing the access control rules into the database itself.

A Funhouse database always answers any query it receives, but returns partial or redacted data, customized to the querying employee and their situational context. The default access level for most employees will be heavily restricted by redacting database contents, but it

[1] Named for funhouse mirrors, which distort the images they display

forms the baseline from which they upgrade their access as needed. For example, a hotel customer service representative (CSR) arrives at their job with minimal access to most customer information (e.g., sees only dummy emails and anonymized hotel bookings).

Cheap and easy upgrades that integrate with practical application workflows are core to Funhouse. Database admins and frontend developers can use upgrades liberally to integrate granting access to data into a workflow where it is needed. For example, when a customer calls the CSR to update their booking, the CSR gets an upgrade that deanonymizes this specific booking.

Instead of enforcing policies before performing queries or trying to redact the actual data in storage, Funhouse integrates access control—redactions and upgrades—into the query execution engine. Access upgrades remove redactions: if the query engine no longer applies redacting transformations in a query, then the real data becomes visible to the querier.

Because redactions are evaluated and applied per query, it's easy for employees to use different access control permissions in different queries by using different upgrades. Because Funhouse supports changing access policies via upgrades, Funhouse addresses a threat model centered on damage limitation rather than perfect security guarantees. This is realistic about the fact that insiders do need to have access to data, while still addressing the need for practical systems that provide protection against curious insiders (e.g., employees snooping on specific customers) and mass data exfiltration using insider accounts.

1.4 Challenges

Making Funhouse a reality requires addressing several open challenges.

Challenge 1: How to specify redactions and upgrades, and what can they express? Redactions should be expressive enough to capture fine-grained data access policies. We can imagine many different types of redaction, ranging from simple data removal or modification, to complex transformations that show completely synthetic data. Upgrades should be able to partially or fully cancel any redaction, e.g., restore access to only part of a removed data

object.

Challenge 2: How to authorize upgrades while avoiding overprompting? One could imagine a system that automatically allows updates, but this would seriously degrade the system's level of security. Explicit upgrade authorization may seem more secure, but frequent use of upgrades to temporarily elevate employees' access may be burdensome, as each upgrade needs to be authorized by someone.

Challenge 3: How to apply redactions correctly and efficiently? One strawman approach, which works correctly for simple redactions such as data modification and removal, first redacts the contents of the input tables, and then executes the SQL query over those redacted temporary tables. But this may be inefficient: the redaction operator may transform many rows and columns only for the rest of the query execution to discard them. In addition, some redactions might need to be placed elsewhere in query execution: e.g., a redaction adding DP noise to aggregations must be applied right before results are returned.

Challenge 4: What are reasonable write semantics over redacted data? Funhouse must determine reasonable semantics for writing to redacted data (if allowed at all), given that redactions hide some data in the database from a writing client.

2 Design

We now present a high-level overview of a possible Funhouse design, and initial solutions to these challenges.

2.1 Overview

Figure 3 shows the Funhouse-application workflow. Applications use Funhouse in place of their database, with an additional setup step to establish redaction policies. Regular day-to-day interactions include upgrades made by the application on behalf of employees when they need more access to data while doing their jobs, and the usual database interactions of reads and writes. Funhouse applies redactions (with upgrades) to query results, returning

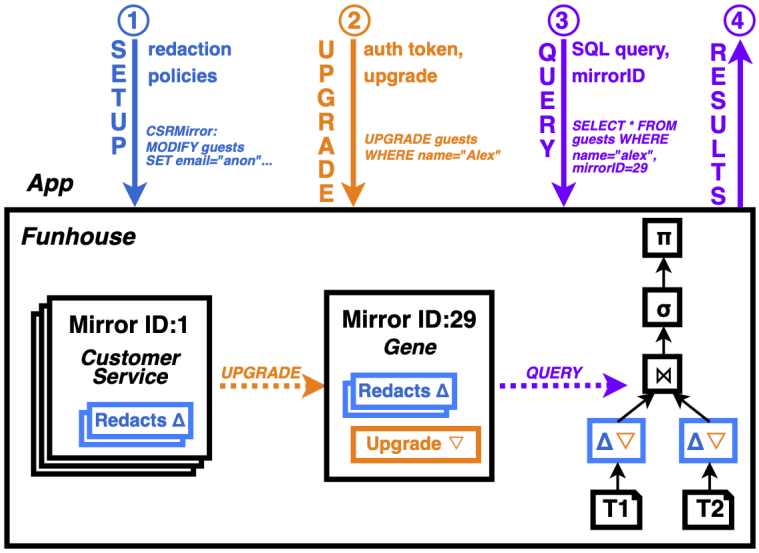


Figure 3: (1) Developers perform a one-time setup to add static redaction policies to Funhouse; (2) Employee clients dynamically upgrade their data access when required to perform their jobs; (3/4) Funhouse executes queries and applies redactions and upgrades as specified, returning (potentially redacted) query responses to the client.

potentially redacted query responses to the client. Funhouse also keeps an audit log of upgrades authorized and used.

Funhouse organizes sets of redaction operations into *mirrors*. Mirrors are a logical concept which conceptually represent a redacted version of the database. During setup, application developers specify the different employee roles and what set of redactions to apply to each role. Funhouse then internally creates a mirror for each role, storing the associated redactions.

2.2 Redactions

Funhouse internally represents redactions as a combination of a predicate and a type of redaction operation: the predicate specifies which data the operation applies to, and the redaction operation specifies a transformation of the data. Three conceivable types of redaction are: removal, which removes the selected data entirely, modification, which changes the cell

values, and decorrelation, which in a central table creates fake entities, and in tables with foreign keys to the central table overwrites foreign keys to point to the newly created fake entities. From the application's perspective, defining a redaction could look like `MODIFY guests SET passportNum=XXXX WHERE true`.

2.3 Upgrades

Funhouse must authorize upgrades prior to application. There are two conceivable workflows for Funhouse to authorize upgrades: higher-up employees can grant access to data unredacted for them, and data subjects (external application users) can grant access to data that is tied to their identity (e.g.,, a hotel guest granting a CSR access to their bookings).

If a data subject is authorizing an upgrade, then Funhouse checks that the upgrade only exposes rows tied to the data subject. It does this by analyzing the upgrade predicate and ensuring that it restricts affected rows through a comparison with an identifier for the data subject. If another employee is the one authorizing an upgrade, Funhouse translates both the upgrade's predicate and the authorizing employee's redactions' predicates into boolean logic, and feeds the logic formulas to an SMT solver. If the result is unsatisfiable (meaning there can be no data that both the upgrade and the authorizing insider's redactions apply to), Funhouse knows that the upgrade only reveals data the authorizing insider can already see, and thus permits the authorization.

Upgrade authorization partly determines the level of security that Funhouse is able to achieve. Funhouse's redactions are guaranteed to hide the existence and/or content of redacted data from a client, but if the client obtains a properly authorized upgrade to access the data then they can see (and leak) the data. However, Funhouse also creates an audit trail that helps detect anomalous behavior by insiders and raises the alarm before large quantities of data can be extracted.

Funhouse represents upgrades internally as a predicate with a timestamp that marks the end of the upgrade's validity, e.g., `UPGRADE guests.passportNum WHERE guestID=5,`

`"2019-01-19 03:14:07"`. Funhouse integrates upgrades with redactions that affect the same table columns, so that upgrades nullify redaction effects. Integration occurs via combining upgrades' predicates with the appropriate redactions' predicates. In the example, the upgraded redaction would be `MODIFY guests SET passportNum=XXX WHERE true AND NOT guestID=5`.

Given an authorized upgrade, Funhouse creates a temporary individual mirror for the requesting employee. This mirror contains exactly the same redactions as the mirror for the employee's role, but also integrates the upgrade with corresponding redactions that affect the same table columns. This allows upgrades to affect only one employee's access levels, while other employees in the same role still observe the original redactions in the role's mirror.

2.4 Query Processing

Funhouse internally plans and executes queries like any SQL database, but adds redaction application to the query plans. It does this by creating a new kind of operator: the redaction operator, which is added to both logical and physical plans.

The placement of the operator within the query is critical for correctness. This placement is made simpler by the fact that every redaction that Funhouse supports can be evaluated and executed on each row in isolation, because it means that Funhouse can (and in fact must) apply redactions right on top of the input tables (Figure 4).

For example, if Funhouse processed `SELECT * FROM bookings WHERE guest.email = caroline@gmail.com` and applied the filter over guest.email *before* applying redactions that anonymize email addresses, Funhouse would still return Caroline's bookings even if the email is anonymized in the returned results. Instead, by redacting the email first, Funhouse ensures that the filter returns only bookings that satisfy the filter after redaction.

The operator itself scans the input table and applies a given redaction if two conditions are met: (*i*) if the redaction's predicate applies, and (*ii*) if there are no upgrades that also would select the given row. In other words, redactions are applied for a given row only if the

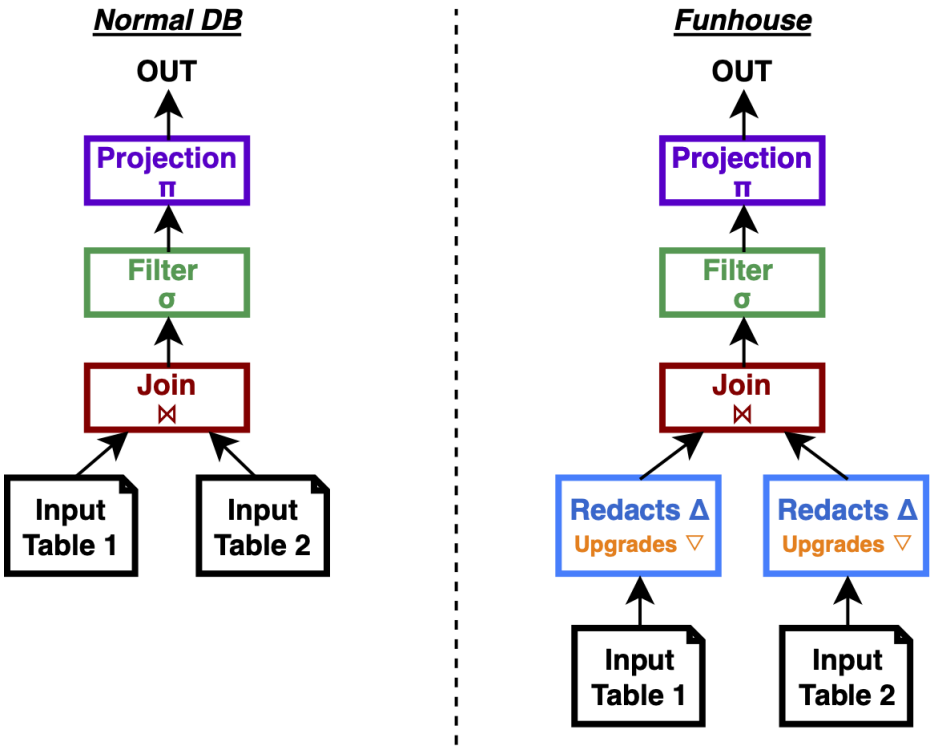


Figure 4: Funhouse adds a new redaction (Δ) operator to query processing, which redacts input data. Redaction application occurs prior to any other operators to produce query results as if the input tables in fact contain redacted data.

redaction's predicate selects the row and no upgrade does.

2.5 Query Optimization

This naive-but-correct version of Funhouse is simple, but inefficient: applying redactions to all tables and rows touched by a query may be more work than is necessary. But because Funhouse makes redactions an explicit operator in the query plan, it is free to optimize query execution by reordering operators so that redactions get applied to fewer rows, while still maintaining correctness. To achieve this, Funhouse can push selective operators (e.g., filters, projections) down but preserves all read-write dependencies: the Funhouse query optimizer should never swap two operators where the child operator reads a column that the parent operator modifies, and vice versa. For example, Funhouse does not move a filter that reads

column A down past a redaction that writes to column A. Importantly, Funhouse can only apply these optimizations because it reasons about data redactions as part of its query plan.

2.6 Writes

Funhouse chooses to enforce that clients can only write to data that has no redactions applied to it (i.e., data they can see because it is unredacted or falls under a current upgrade). This avoids complications like leaking the existence of hidden rows if someone tries to write a row that already exists, but that row is removed in their mirror.

3 Implementation

3.1 Prototype Details

We built a prototype of Funhouse in 5,300 lines of C++. It is an in-memory database that supports most SQL queries. It has internal data structures for storing the contents of the database, as well as mirrors, redaction operators, and upgrades.

Mirrors are initialized with a set of Redactions, defined via a map from table name to a vector of Redactions. Mirrors also have methods to bless Upgrades (for both internal and external users), and a method to apply a given Upgrade. Lastly, Mirrors contain a PseudoEntityStore, which is used to keep track of the fake entities created by any decorrelation redactions (so that they can remain stable across query invocations).

Redactions as a baseline contain a Predicate (which is an internal data structure that can evaluate whether it applies to a given Row), as well as a vector of Upgrades (initially empty). Redactions also have a field defining their type, and there are subtypes of Redaction for different kinds of Redactions. Two are straightforward:

1. `RemoveRedaction`: contains no further fields.

2. `ModifyRedaction`: for every column to be changed, contains a function that maps from an original Cell value to a modified one.

The third, decorrelation, is more involved and refers to a central table and dependent tables that have foreign keys into them. It also requires a differentiation between `pseudoEntities` that are additional vs ones that replace existing entities, which is reflected in the following two types of redactions:

1. `DecorrelateRedaction`: this is created for both the central table (i.e., the one where the new entities are to be stored) and other tables (where the foreign keys into the central table need to be overwritten). Thus, this `Redaction` type needs to know what table it refers to, whether that table is the central table, and the key column (to either be overwritten, or by which to index to find PseudoEntity replacing the real one).

2. `AddPseudoEntitiesRedaction`: is very similar to a `DecorrelationRedaction`, except that here instead of replacing existing entities, the `PseudoEntities` are appended to the table (i.e., they exist as additional users).

Finally, all Redactions have at least the following methods:

1. `Apply(Row row, std::optional<PseudoEntityStore*> pe-store)`: applies the redaction to a given row (in place).

2. `DoesRedactionApply(Row row)`: returns a boolean indicating whether the redaction applies to the given row.

3. `AddUpgrade(std::shared-ptr<Upgrade> new-upgrade)`: adds the upgrade to the current Redaction.

4. `GetColumnsRead()`: returns the columns that the redaction reads (helpful for optimization).

5. `GetColumnsWritten()`: returns the columns that the redaction writes (helpful for optimization).

In order to implement query optimization, Funhouse iterates over the graph of the logical plan in topological order, and moves down data-reducing operations (i.e. filters and projections) as specified by a set of optimization rules. In doing so, Funhouse keeps track of whether the plan has changed, i.e. whether it was able to move an operation down. Funhouse continues to iterate over the graph until it has done a full iteration where it was unable to move anything. Funhouse also ensures that where possible it removes a Redaction from the plan wholesale. This happens when it moves an operation past a Redaction in a way that completely eliminates from the results any of the output the Redaction would have, for example if all columns written to by the Redaction are being projected away. As a further optimization, in the physical plan contiguous scan operations are all executed in the same pass over the input data. The prototype relies on the Z3 [22] constraint solver to check whether an insider can authorize an upgrade for another insider. It also uses the HyRise SQL parser library [13] in order to parse SQL queries.

3.2 Limitations

The prototype has some limitations. For one, it does not support the full range of possible SQL queries. It lacks support for subqueries, joins beyond two tables, or aliasing in joins or columns in general. The prototype also does not use indexing for optimization, and is fully in-memory. Additionally, Redactions are currently created directly as the Funhouse-internal data type in the application code, but one could imagine developing a more user-friendly interface to extract the necessary information from a different syntax, for example a SQL-like specification interface.

4 Evaluation

We built a prototype of Funhouse in 5,300 lines of C++. Our prototype is an in-memory database that supports most SQL queries, as well as redactions and upgrades. It relies on the Z3 [22] constraint solver to check whether an insider can authorize an upgrade for another

insider.

4.1 Case Study

We evaluate our prototype with a case study of a hotel's database. The database contains tables for guests (the application's data subjects), rooms, bookings, staff, cleanings, and credit cards. We configure a mirror for CSRs, which applies redactions that:

1. anonymize the first and last name, email, and phone number of all guests;
2. hide all payment and credit card information by replacing them with placeholders;
3. decorrelate future bookings and those within the last year, to hide time-sensitive information about guests' whereabouts.

In addition to these redactions, we include an upgrade to reveal an individual guest's data. This upgrade removes any redactions applied to the guest's bookings and the guest's entry in the guests table.

4.2 Results

Our experiment populates the database with a plausible dataset for a small hotel's bookings: 450 guests and 500 bookings over 20 rooms, with other tables populated proportionally. We measure the query runtime for three queries:

(Q1) a query that combines booking and credit card information for a specific booking. This exercises the second redaction, returning placeholder credit card numbers.

(Q2) a query that joins bookings with guest information. This exercises the first and third redactions: all guest information will be hidden, and bookings within the last year will be associated with fake guests created by Funhouse.

(Q3) a query that joins cleaning and booking information to get a specific staff member's cleanings. No redactions will apply to the query result.

We compare the following setups running the query: (*i*) without any redactions; (*ii*) with redactions but without redaction-aware query optimizations; (*iii*) with redaction-aware

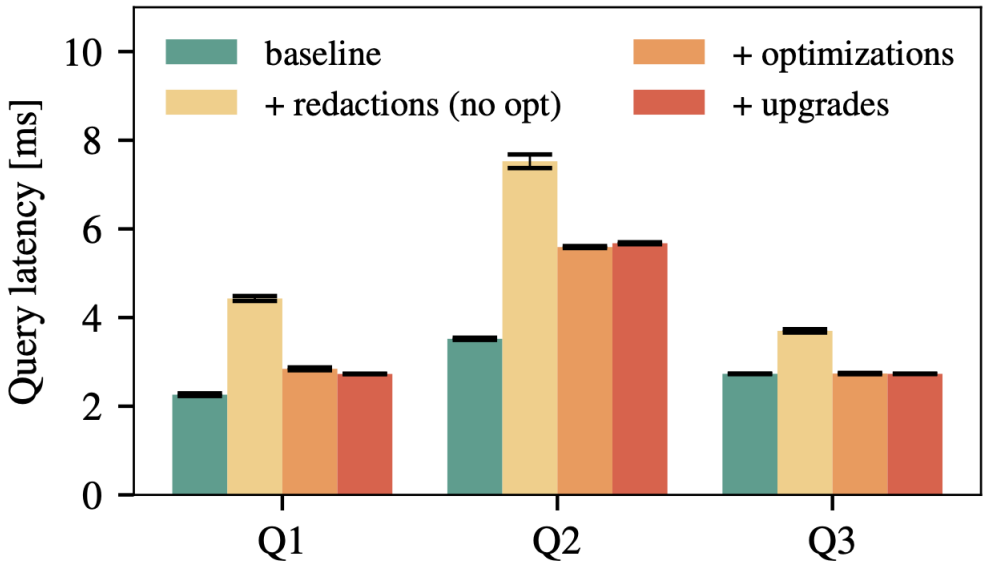


Figure 5: Addition of redactions and upgrades add overheads ($\leq$ 2x) to a baseline without redactions; optimizations reduce overheads to $\leq$ 1.6x or remove them entirely (e.g., Q3).

optimizations; and (*iv*) with redactions, redaction-aware optimizations, and upgrades.

A good result would show that Funhouse's redactions and upgrades add small, acceptable overheads and that redaction-aware query optimizations reduce these overheads.

Figure 5 shows the results. As expected, processing redactions within the query plan adds overhead (1.3–2x). Redaction-aware query optimizations, however, reclaim much of this overhead on Q1 and Q3, and improve Q2's runtime. Figure 6 shows the full rewrite of Q1's logical plan: optimization not only performs filter and projection pushdown as a normal query optimizer would, but also drops the redaction on the bookings table, since the query uses none of the redacted columns. This reduces the number of redactions applied and number of rows modified by the redaction. Q2 sees the highest overhead from redactions, as it returns all bookings and guests with redactions applied. But even Q2 benefits as projection pushdown through the redaction removes several columns the redaction modifies. In Q3, the query does not use any data touched by redactions, and Funhouse's optimizer therefore removes the redactions.

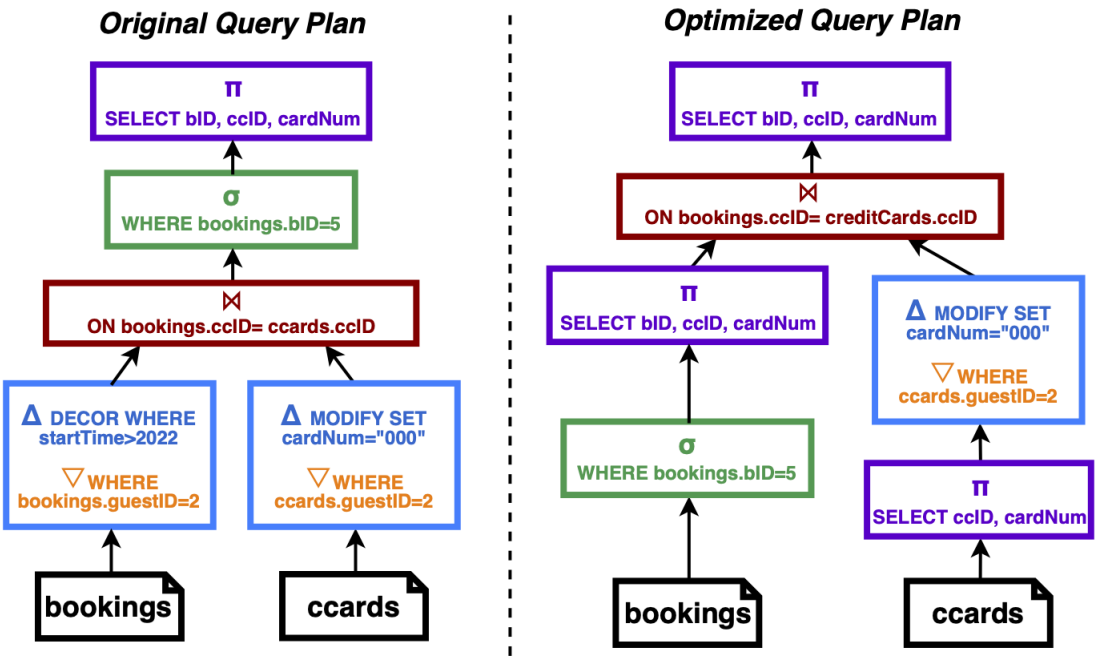


Figure 6: Funhouse's optimizations for Q1 drop an unused redaction over bookings and push a projection through the other redaction.

Overall, redaction-aware optimizations remove the overhead of redactions entirely on Q3, reduce it to 1.2x on Q1, and to 1.6x on Q2. Adding upgrades incurs little to no further overhead, which makes sense, as upgrades merely nullify redactions. The absolute query latency remains within single-digit milliseconds despite redactions and upgrades. Since our prototype is an in-memory database, we measured pure query execution overhead; a persistent database would add I/O cost in all setups that reduces the relative overhead.

These results demonstrate that Funhouse's approach of integrating data redaction and access upgrades directly with query execution is viable.

5 Related Work

Classic ACLs as well as large-scale permission systems like Amazon's IAM [4] and Google's Zanzibar [27] allow users to define access rules to resources. This also includes attribute based access control (ABAC) [8, 23, 34], and any other systems that make access judgments

based on rules and/or other factors. These systems increasingly support more flexible rules with approaches like attribute-based access control, which allows rules to be based on factors beyond the user's role, such as time of day or location; although these ultimately still boil down to a set of predefined rules. These approaches also enforce rigid allow/deny policies that cannot respond to the context of a user's access except via pre-defined attributes. Funhouse seeks to provide more flexibility by allowing applications to integrate upgrades, which are flexibly defined over the database contents (i.e., any attributes expressible in SQL), into their workflow where context requires them.

Works like the Open Policy Agent [1] make the point that it is desirable to "decouple policy from the service's code". However, this is contraindicated by the aforementioned work, which is actually turning around and taking separated policies like ACLs and trying to bring contextual information back into them. Funhouse makes the argument that the baseline access permissions can and should be separate from the service's code and workflow, but that the upgrades are in fact not only entangled with factors visible to the application code, but in fact are to a large extent determined by that context.

Database access control allows users to grant and revoke privileges to users or roles for specific actions (e.g., insert or select) [2, 6, 7, 9, 23, 28, 29, 32]. This allows for fine-grained privileges, but typically only allows or denies access to data, and is meant to support only static policies. Other approaches grant access to (potentially redacted) database views, but require pre-defining views for each user or role [11, 15]. Policy checking systems such as Qapla [21] and BlockAid [35] determine whether a query is allowed or not. Qapla rewrites queries to only select data that is allowed by a policy, and BlockAid transforms SQL queries into logic statements to determine whether the query touches only data contained in views that the querier is allowed to access (without materializing those views). Both BlockAid and Qapla cannot react to contextual cues. Qapla redacts data only via replacement with predetermined values, while BlockAid completely blocks the query from running if the policy disallows access, instead of returning partially-redacted data.

Multiverse Databases [19] create a conceptually separate database for every user, and apply per-user privacy policies. Multiverse DBs show users redacted versions of the base database contents, but realize this by creating per-user materialized views, which come with high space and write processing overheads. To date, no practical performant implementation of multiverse databases exists.

View-based access control systems [10, 15] are in the space of trying to respond to every query, but are less flexible than Funhouse: there can only be a few actually materialized views, and queries have to be aware of exactly what view they need to run on, given the current context. Additionally, changing someone's permissions requires creating an entirely new view definition, as well as potentially re-deriving a materialized view.

Databases focused on k-anonymity or differential privacy (DP) guarantees guarantees [17, 20, 24] have orthogonal goals to Funhouse, and have no notion of role-based privacy levels or modifications to these levels (i.e., Funhouse's upgrades). In the case of DP, these systems support only aggregate queries.

6 Conclusion

6.1 Summary

In conclusion, we have designed a new database that redacts the data that it holds by default, and in a way that is dependent on who is querying. We find that doing so allows us to support two dimensions of desirable flexibility: on one hand no query is rejected, because data returned can be redacted; on the other changing someone's permissions in Funhouse can be a common operation, by the fact that users' access is re-evaluated at every query and only the change needs to be specified (rather than an entirely new policy that includes the change). On a fundamental level, evaluating redaction policies at query time has allowed us to have the redactions apply on the level of the data, but the specifications for them exist on a higher declarative (and thus more flexible) level.

In order to realize this, we create a new operator that understands who the querier is, and what redactions should be applied for them. The operator then applies the given set of redactions to whatever the input was, letting the rest of the query execute as if the data had been in the redacted form all along.

We also designed a way of applying traditional query optimization techniques for operator reordering to the new operator, so that ideally the query only ends up applying redactions (a potentially expensive operation) to data actually returned.

We implemented a prototype that supports a basic set of SQL operations, and three types of redactions. In addition, we created a case study for a hotel database, and used it to verify the correctness of the redactions as well as evaluate the optimizations we implemented.

6.2 Future Work

Our design takes a first stab at addressing the challenges for creating a Funhouse database; we propose potential next steps here.

Upgrade Authorization and Overprompting. Our design allows employees to grant access to data they have unredacted access to. However, upgrade requests may frequently prompt employees to grant their approval, leading to burdensome overprompting. We plan to explore how the database could introduce a notion of rate-limited, audited upgrade *templates* that authorizing employees can sign once, and which automatically authorize a particular set of upgrades. The template could include not only the upgrade content, but also the user role and other attributes that are conditions for automatic approval. Even so, templates reduce oversight on upgrade usage, and a compromised employee might abuse them. Making such templates and their approval user-friendly is an interesting direction for further research, and will likely require new APIs and user interfaces.

Redaction and Upgrade Representation. Our design currently supports a limited form of redaction that selects and applies to one row at a time. Improvements on our design may add support for more complex redactions, e.g., redactions over joined rows, redactions over

aggregations, or redactions that even create fully synthetic databases following some realistic distribution of data values. More complex redactions require reasoning about how to ensure correct and efficient placement of the redaction operator (or operators), and how redactions may compose.

Writes. Our current design only allows writes if the writer has unredacted access to the data to be written. This has the downside that it might lead to frequently failing writes and consequent prompts to the client to upgrade, burdening both the client and authorizer. We plan to explore both relaxing this requirement on writes (and the resulting consequences on write semantics), and directions such as write-only upgrade templates to reduce overprompting.

References

[1] Open Policy Agent. URL: https://www.openpolicyagent.org/.

[2] Rakesh Agrawal, Jerry Kiernan, Ramakrishnan Srikant, and Yirong Xu. "Hippocratic Databases". In: *Proceedings of the 28th International Conference on Very Large Data Bases*. VLDB '02. Hong Kong, China: VLDB Endowment, 2002, pages 143–154.

[3] Yanif Ahmad, Oliver Kennedy, Christoph Koch, and Milos Nikolic. *DBToaster: Higher-order Delta Processing for Dynamic, Frequently Fresh Views*. 2012. arXiv: 1207.0137 [cs.DB].

[4] AWS. *AWS Identity and Access Management (IAM)*. 2023. URL: https://aws.amazon.com/iam/ (visited on 01/30/2023).

[5] Peter Borner. *Morrisons data leak*. Oct. 2018. URL: https://thedataprivacygroup.com/us/blog/morrisons-data-leak/ (visited on 01/30/2023).

[6] Kristy Browder and Mary Ann Davidson. "The Virtual Private Database in Oracle9iR2". In: *Oracle Technical White Paper, Oracle Corporation* 500.280 (2002).

[7] Ji-Won Byun and Ninghui Li. "Purpose based access control for privacy protection in relational database systems". In: *The VLDB Journal* 17.4 (July 1, 2008), pages 603–619.

[8] casbin. URL: https://casbin.org/.

[9] Surajit Chaudhuri, Tanmoy Dutta, and S. Sudarshan. "Fine Grained Authorization Through Predicated Grants". In: *2007 IEEE 23rd International Conference on Data Engineering*. 2007, pages 1174–1183.

[10] D.E. Denning, S.G. Akl, M. Heckman, T.F. Lunt, M. Morgenstern, P.G. Neumann, and R.R. Schell. "Views for Multilevel Database Security". In: *IEEE Transactions on Software Engineering* SE-13.2 (1987), pages 129–140.

[11] Dorothy E Denning, Selim G Akl, Mark Heckman, Teresa F. Lunt, Matthew Morgenstern, Peter G. Neumann, and Roger R. Schell. “Views for multilevel database security”. In: *IEEE Transactions on Software Engineering* 2 (1987), pages 129–140.

[12] Daniel J. Dougherty, Kathi Fisler, and Shriram Krishnamurthi. “Specifying and Reasoning About Dynamic Access-Control Policies”. In: *Automated Reasoning*. Edited by Ulrich Furbach and Natarajan Shankar. Berlin, Heidelberg: Springer Berlin Heidelberg, 2006, pages 632–646.

[13] github. *C++ SQL Parser*. URL: `https://github.com/hyrise/sql-parser`.

[14] 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: *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*. Carlsbad, CA: USENIX Association, Oct. 2018, pages 213–231.

[15] Patricia P Griffiths and Bradford W Wade. “An authorization mechanism for a relational database system”. In: *ACM Transactions on Database Systems (TODS)* 1.3 (1976), pages 242–255.

[16] Xin Jin, Ram Krishnan, and Ravi Sandhu. “A Unified Attribute-Based Access Control Model Covering DAC, MAC and RBAC”. In: *Data and Applications Security and Privacy XXVI*. Edited by Nora Cuppens-Boulahia, Frédéric Cuppens, and Joaquin Garcia-Alfaro. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, pages 41–55.

[17] Kristen LeFevre, David J. DeWitt, and Raghu Ramakrishnan. “Incognito: Efficient Full-Domain K-Anonymity”. In: *Proceedings of the 2005 ACM SIGMOD International Conference on Management of Data*. SIGMOD ’05. Baltimore, Maryland: Association for Computing Machinery, 2005, pages 49–60.

[18] Mariott International, Inc. *Marriott Announces Starwood Guest Reservation Database Security Incident*. Nov. 2018. URL: https://news.marriott.com/news/2018/11/30/marriott-announces-starwood-guest-reservation-database-security-incident (visited on 01/30/2023).

[19] Alana Marzoev, Lara Timbó Araújo, Malte Schwarzkopf, Samyukta Yagati, Eddie Kohler, Robert Morris, M. Frans Kaashoek, and Sam Madden. "Towards Multiverse Databases". In: *Proceedings of the 17th Workshop on Hot Topics in Operating Systems (HotOS)*. 2019, pages 88–95.

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

[21] Aastha Mehta, Eslam Elnikety, Katura Harvey, Deepak Garg, and Peter Druschel. "Qapla: Policy Compliance for Database-Backed Systems". In: *Proceedings of the 26th USENIX Security Symposium (USENIX Security)*. Vancouver, British Columbia, Canada, Aug. 2017, pages 1463–1479.

[22] Leonardo de Moura and Nikolaj Bjørner. "Z3: An Efficient SMT Solver". In: *Tools and Algorithms for the Construction and Analysis of Systems*. Edited by C. R. Ramakrishnan and Jakob Rehof. Berlin, Heidelberg: Springer Berlin Heidelberg, 2008, pages 337–340.

[23] Ravi Murthy and Eric Sedlar. "Flexible and Efficient Access Control in Oracle". In: *Proceedings of the 2007 ACM SIGMOD International Conference on Management of Data*. SIGMOD '07. Beijing, China: Association for Computing Machinery, 2007, pages 973–980.

[24] Arjun Narayan and Andreas Haeberlen. "DJoin: Differentially Private Join Queries over Distributed Databases". In: *Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation*. OSDI'12. Hollywood, CA, USA: USENIX Association, 2012, pages 149–162.

[25] Charlie Osborne. *Trend Micro reveals rogue employee sold data of up to 120,000 customers*. Nov. 2019. URL: https://www.zdnet.com/article/trend-micro-reveals-insider-threat-exposing-customer-data/ (visited on 01/30/2023).

[26] Palo Alto Networks. *What Is the Principle of Least Privilege?* 2023. URL: https://www.paloaltonetworks.com/cyberpedia/what-is-the-principle-of-least-privilege (visited on 01/30/2023).

[27] Ruoming Pang, Ramon Caceres, Mike Burrows, Zhifeng Chen, Pratik Dave, Nathan Germer, Alexander Golynski, Kevin Graney, Nina Kang, Lea Kissner, Jeffrey L. Korn, Abhishek Parmar, Christina D. Richards, and Mengzhi Wang. "Zanzibar: Google's Consistent, Global Authorization System". In: *2019 USENIX Annual Technical Conference (USENIX ATC '19)*. Renton, WA, 2019.

[28] Shariq Rizvi, Alberto Mendelzon, S. Sudarshan, and Prasan Roy. "Extending Query Rewriting Techniques for Fine-Grained Access Control". In: *Proceedings of the 2004 ACM SIGMOD International Conference on Management of Data*. SIGMOD '04. Paris, France: Association for Computing Machinery, 2004, pages 551–562.

[29] *Row-level security - SQL server*. URL: https://learn.microsoft.com/en-us/sql/relational-databases/security/row-level-security (visited on 01/30/2023).

[30] Bruce Schneier. *The Uber Hack Exposes More Than Failed Data Security*. Sept. 2022. URL: https://www.nytimes.com/2022/09/26/opinion/uber-hack-data.html (visited on 01/30/2023).

[31] Twitter Support. *Twitter Support Investigation*. Jan. 2023. URL: https://twitter.com/TwitterSupport/status/1283591844962750464?s=20 (visited on 07/15/2020).

[32] Jean Yang, Travis Hance, Thomas H Austin, Armando Solar-Lezama, Cormac Flanagan, and Stephen Chong. “Precise, dynamic information flow for database-backed applications”. In: *ACM Sigplan Notices* 51.6 (2016), pages 631–647.

[33] E. Yuan and J. Tong. “Attributed based access control (ABAC) for Web services”. In: *IEEE International Conference on Web Services (ICWS'05)*. 2005, page 569.

[34] E. Yuan and J. Tong. “Attributed based access control (ABAC) for Web services”. In: *IEEE International Conference on Web Services (ICWS'05)*. 2005, page 569.

[35] Wen Zhang, Eric Sheng, Michael Chang, Aurojit Panda, Mooly Sagiv, and Scott Shenker. “Blockaid: Data Access Policy Enforcement for Web Applications”. In: *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*. Carlsbad, CA: USENIX Association, July 2022, pages 701–718.