Qrlew: Query Rewriting for Differential Privacy

Nicolas Grislain[1] **Paul Roussel**[1] **Victoria de Sainte Agathe**[1]

[1]Sarus Technologies
128 rue la Boétie
75008, Paris, France
contact@sarus.tech

Abstract

This paper introduces *Qrlew*, an *open source* library that can parse SQL queries into *Relations* — an intermediate representation — that keeps track of rich data types, value ranges, and row ownership; so that they can easily be rewritten into *differentially-private* equivalent and turned back into SQL queries for execution in a variety of standard data stores.

With *Qrlew*, a *data practitioner* can express their data queries in standard SQL; the *data owner* can run the rewritten query without any technical integration and with strong privacy guarantees on the output; and the query rewriting can be operated by a privacy-expert who must be trusted by the owner, but may belong to a separate organization.

1 Introduction

In recent years, the importance of safeguarding privacy when dealing with personal data has continuously increased. Traditional anonymization techniques have proven vulnerable to re-identification, as demonstrated by numerous works (Archie et al. 2018; Dwork et al. 2017; Narayanan and Shmatikov 2008; Sweeney, Abu, and Winn 2013). The total cost of data breaches has also significantly increased (IBM 2023) and governments have introduced stricter data protection laws. Yet, the collection, sharing, and utilization of data holds the potential to generate significant value across various industries, including healthcare, finance, transportation, and energy distribution.

To realize these benefits while managing privacy risks, researchers have turned to *differential privacy (DP)* (Wood et al. 2018; Dwork, Roth et al. 2014), which has become the *gold standard* for privacy protection since its introduction by Dwork et al. in 2006 (Dwork et al. 2006) due to its provable and automatic privacy guarantees.

Despite the availability of powerful open-source tools (Kotsogiannis et al. 2019; Holohan et al. 2019; The OpenDP Team 2023; Google 2022; OpenMined and Google 2023; Google 2023b; Johnson et al. 2020; Berghel et al. 2022; Yousefpour et al. 2021), DP adoption remained limited and many organizations sticked to more manual and *ad-hoc* approaches. Reasons for this lack of adoption are probably complex and multiple but one could name: the lack of awareness on privacy risks; the loss of utility in the results; and the perceived complexity of the existing solutions considering they all require, either some expertise in differential privacy, or the use of new interfaces to express data processing tasks, or even to integrate new execution engines in their data stack. *Qrlew* (Grislain et al. 2023) has been designed to relieve these problems by providing the following features:

Qrlew **provides automatic output privacy guarantees** With *Qrlew* a *data owner* can let an analyst (*data practitioner*) with no expertise in privacy protection run arbitrary SQL queries with strong privacy garantees on the output.

Qrlew **leverages existing infrastructures** *Qrlew* rewrites a SQL query into a *differentially private* SQL query that can be run on any data-store with a SQL interface: from lightweight DB to big-data stores. This removes the need for a custom execution engine and enables *differentially private analytics with virtually no technical integration*.

Qrlew **leverages synthetic data** Synthetic data are an increasingly popular way of *privatizing* a dataset (Bowen and Snoke 2019; McKenna, Miklau, and Sheldon 2021; Canale et al. 2022; Sablayrolles, Wang, and Karrer 2023; Castellon et al. 2023). Using jointly: *differentially private* mechanisms and *differentially private* synthetic data can be a simple, yet powerful, way of managing a privacy budget and reaching better utility-privacy tradeoffs.

2 Definitions

Datasets and Privacy Units (PU)

In this paper, *datasets* refer to a collection of elements in some domain $\mathcal{X}$, labelled with an identifier $i \in \mathcal{I}$ identifying the entity whose privacy we want to protect. This entity will be called *Privacy Unit* (PU) and the identifier will be referred to as *Privacy ID* (PID). Let $\mathcal{D}$ be the set of datasets of arbitrary sizes with a privacy unit.

Differential Privacy (DP)

Let $\mathcal{M}$ be an algorithm that takes a dataset as input and produces a randomized output. The algorithm $\mathcal{M}$ is said to satisfy ε, δ-differential privacy if, for all pairs of adjacent datasets $D, D' \in \mathcal{D}$, and for all measurable sets S in the range of $\mathcal{M}$:

$$\Pr[\mathcal{M}(D) \in S] \leq e^{\varepsilon} \cdot \Pr[\mathcal{M}(D') \in S] + \delta$$

Adjacent datasets

Datasets $D, D' \in \mathcal{D}$ are adjacent if they are equal up to the addition or removal of all entries sharing the same PID. Note that this is a slightly unusual and restricted definition of adjacency, suited to our practical needs. It is close to that used in the *user-level differential privacy* literature (Liu et al. 2020; Wilson et al. 2019) where one user can have many samples.

3 Assumptions and Design Goals

In this work, we assume the *central model of differential privacy* (Near 2020), where a trusted central organization: hospital, insurance company, utility provider, called the *data owner*, collects and stores personal data in a secure database and whishes to let untrusted *data practitioners* run SQL queries on its data.

At a high level we pursued the following requirements:

- Ease of use for the *data practitioners*. The *data practitioners* are assumed to be a data experts but no privacy experts. They should be able to express their queries in a standard way. We chose SQL as the query language as it is very commonly used for analytics tasks.
- Ease of integration for the *data owner*. As SQL is a common language to express data analysis tasks, many datastores support it from small embedded databases to big data stores.
- Simplicity for the *data owner* to setup privacy protection. Differential privacy is about capping the sensitivity of a result to the addition or removal of an individual that we call *privacy unit*. *Qrlew* assumes that the *data owner* can tell if a table is public and, if it is not, that it can assign exactly one *privacy unit* to each row of data. In the case there are multiple related tables, *Qrlew* enables to define easily the *privacy units* for each tables transitively.
- Simple integration with other privacy enhancing technologies such as *synthetic data*. To avoid repeated privacy losses or give result when a DP rewriting is not easily available (e.g. when the query is: `SELECT * FROM table`) *Qrlew* can use *synthetic data* to blend in the computation.

These requirements dictated the overall *query rewriting* architecture and many features, the most important of which, are detailed below.

4 Architecture and main features of *Qrlew*

The *Qrlew* library, solves the problem of running a SQL query with DP guarantees in three steps. First the SQL query submitted by the *data practitioner* is parsed and converted into a *Relation*, this *Relation* is an intermediate representation that is designed to ease the tracking of data types ranges or possible values, to ease the tracking of the *privacy unit* and to ease the rewriting into a DP *Relation*. Then, the rewriting into DP happens. Once the relation is rewritten into a DP one, it can be rendered as an SQL query string and submitted to the data store of the *data owner*. The output can then safely be shared with the *data practitioner*. This process is illustrated in figure 1.

4.1 Qrlew Intermediate Representation

As the SQL language is very rich and complex, simply parsing a query into an abstract syntax tree does not produce a convenient representation for our needs. Therefore, it is converted into a simpler normalized representation with properties well aligned with the requirements of Differential Privacy: the *Relation*. A *Relation* is a collection of rows adhering to a given *schema*. It is a recursively defined structure composed of:

Tables This is simply a data source from a database.

Maps A *Map* takes an input *Relation*, filters the rows and transform them one by one. The filtering conditions and row transforms are expressed with expressions similar to those of SQL. It acts as a `SELECT exprs FROM input WHERE expr LIMIT value` and therefore preserve the *privacy unit* ownership structure.

Reduces A *Reduce* takes an input *Relation* and aggregates some columns, possibly group by group. It acts as a `SELECT aggregates FROM input GROUP BY expr`. This is where the rewriting into DP will happen as described in section 4.4.

Joins This *Relation* combines two input *Relations* as a `SELECT * FROM left JOIN right ON expr` would do it. The privacy properties are more complex to propagate in this case.

It may also be a static list of values or a set operation between two *Relations*, but those are less important for our uses.

This representation is central to *Qrlew*; all the features described below are built upon it. A *Relation*, along with all the sub-*Relations* it depends on, will be called the *computation graph* or the *graph* of a *Relation*.

4.2 Range Propagation

Most DP mechanisms aggregating numbers require the knowledge of some bounds on the values (see (Dwork, Roth et al. 2014)). Even if some bounds are known for some *Relations* like source *Tables*, it is not trivial to propagate these bounds through the steps of the computation.

To help with range propagation, *Qrlew* introduces two useful concepts:

- The concept of k-*Interval*, which are finite unions of at most k closed intervals. A k-*Interval* can be noted:

$$I = \bigcup_{i=1}^{j \leq k} [a_i, b_i]$$

Note that the union of k-*Interval*s may not be a k-*Interval* as it may be the union of more than k intervals. Unions of many intervals can be simplified into their convex envelope interval, which are often sufficient bounds approximations for our use cases:

$$J = \bigcup_{i=1}^{j > k} [a_i, b_i] \subseteq \left[\min_i a_i, \max_i b_i\right]$$

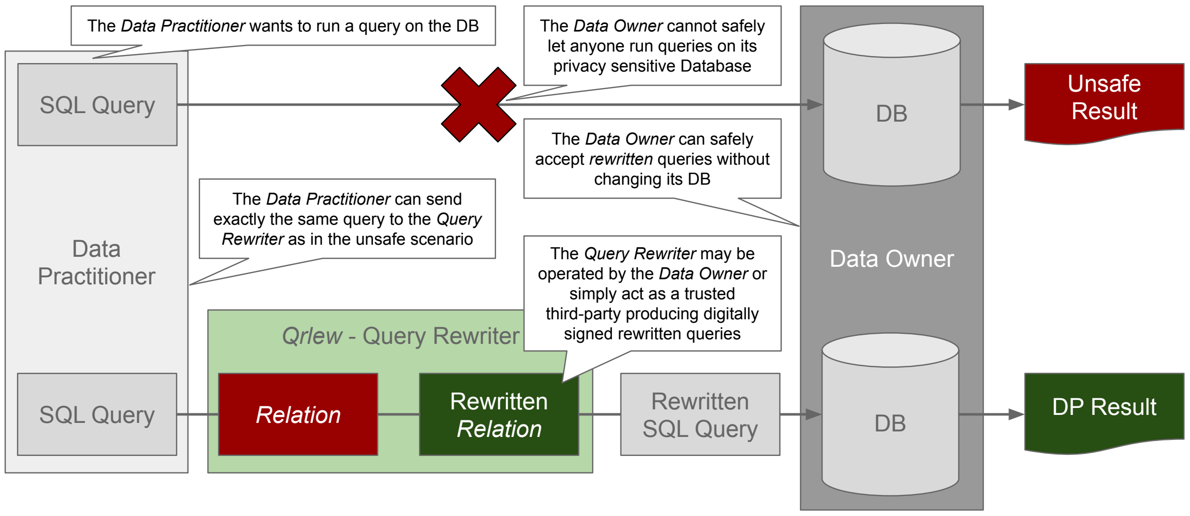

The *Data Practitioner* wants to run a query on the DB
SQL Query
The *Data Owner* cannot safely let anyone run queries on its privacy sensitive Database
DB
Unsafe Result
The *Data Owner* can safely accept *rewritten* queries without changing its DB
Data Practitioner
The *Data Practitioner* can send exactly the same query to the *Query Rewriter* as in the unsafe scenario
Data Owner
Qrlew - Query Rewriter
The *Query Rewriter* may be operated by the *Data Owner* or simply act as a trusted third-party producing digitally signed rewritten queries
SQL Query
Relation
Rewritten *Relation*
Rewritten SQL Query
DB
DP Result

Figure 1: The rewritting process occurs in three stages: The *data practitioner*'s query is parsed into a *Relation*, which is rewritten into a DP equivalent and finally executed by the the *data owner* which returns the privacy-safe result.

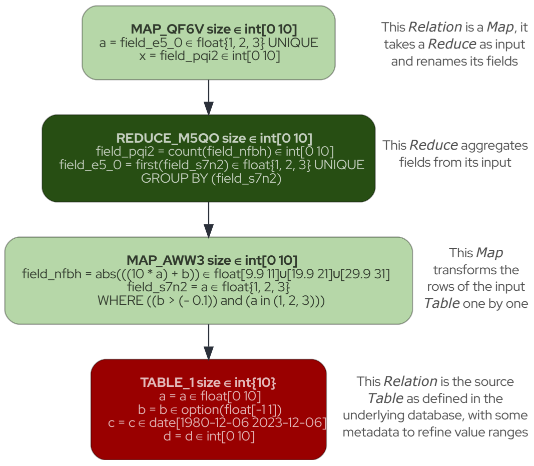

MAP_QF6V size ∈ int[0 10]
a = field_e5_0 ∈ float{1, 2, 3} UNIQUE
x = field_pqi2 ∈ int[0 10]
This *Relation* is a *Map*, it takes a *Reduce* as input and renames its fields
REDUCE_M5QO size ∈ int[0 10]
field_pqi2 = count(field_nfbh) ∈ int[0 10]
field_e5_0 = first(field_s7n2) ∈ float{1, 2, 3} UNIQUE
GROUP BY (field_s7n2)
This *Reduce* aggregates fields from its input
MAP_AWW3 size ∈ int[0 10]
field_nfbh = abs(((10 * a) + b)) ∈ float[9.9 11]∪[19.9 21]∪[29.9 31]
field_s7n2 = a ∈ float{1, 2, 3}
WHERE ((b > (- 0.1)) and (a in (1, 2, 3)))
This *Map* transforms the rows of the input *Table* one by one
TABLE_1 size ∈ int{10}
a = a ∈ float[0 10]
b = b ∈ option(float[-1 1])
c = c ∈ date[1980-12-06 2023-12-06]
d = d ∈ int[0 10]
This *Relation* is the source *Table* as defined in the underlying database, with some metadata to refine value ranges

Figure 2: *Relation* (*Map*) associated to the query: `SELECT a, count(abs(10*a+b)) AS x FROM table_1 WHERE b>-0.1 AND a IN (1,2,3) GROUP BY a`. The arrows point to the inputs of each *Relation*. Note the propagation of the data type ranges.

- And the concept of *piecewise-monotonic-functions*[1], which are functions $f : \mathbb{R}^n \rightarrow \mathbb{R}$ whose domain can be partitioned in cartesian products of intervals: P_j on which they are *coordinatewise-monotonic*. The image of a cartesian product of n k-*Intervals* by a *piecewise-monotonic-function* can be easily computed as a k-*Interval*. Indeed, let I be:

$$I = I_1 \times I_2 \times \ldots \times I_n = \bigcup_{\substack{1 \leq i_1 \leq k \\ \cdots \\ 1 \leq i_n \leq k}} [a_{i_1}, b_{i_1}] \times \ldots \times [a_{i_n}, b_{i_n}]$$

If f is *piecewise-monotonic*, then one can show that on each partition P_j where it is *coordinatewise-monotonic*, if we note:

$$I_j = I \cap P_j = \bigcup_{\substack{1 \leq j_1 \leq k \\ \cdots \\ 1 \leq j_n \leq k}} [a_{j_1}, b_{j_1}] \times \ldots \times [a_{j_n}, b_{j_n}]$$

$$f(I_j) = \bigcup_{\substack{1 \leq i_1 \leq k \\ \cdots \\ 1 \leq i_n \leq k}} \mathrm{Conv}\left(f\left(\{a_{i_1}, b_{i_1}\} \times \ldots \times \{a_{i_n}, b_{i_n}\}\right)\right)$$

where $\mathrm{Conv}\left(f\left(\{a_{i_1}, b_{i_1}\} \times \ldots \times \{a_{i_n}, b_{i_n}\}\right)\right)$ can be efficiently computed in n steps, without testing all the 2^n combinations, thanks to the coordinatewise monotony of f on P_j. Then $f(I) = \bigcup_j f(I_j)$, of which we can derive the bounding: $f(I) \subseteq \mathrm{Conv}\left(\bigcup_j f(I_j)\right)$ when the number of terms in the union exceeds k.

The notion of k-*Interval* is convenient for tracking value bounds as it can express natural patterns in SQL such as:

- `WHERE x>0 AND x<=1`, which translates into the implied $x \in [0, 1]$;
- `WHERE x IN (1,2,3)`, which is also easily expressed as a k-*Interval*: $x \in [1, 1] \cup [2, 2] \cup [3, 3]$

The idea of *piecewise-monotonic-function* is also very useful as in SQL many standard arithmetic operators ($+$, $-$, $\star$, $/$, $<$, $>$, $=$, $!=$, $\ldots$) and functions (`EXP`, `LOG`, `ABS`, `SIN`, `COS`, `LEAST`, `GREATEST`, ...) are trivially *piecewise-monotonic-function* (in one, two or many variables).

[1]Which is a shorthand name for what would be better called: *piecewise-coordinatewise-monotonic-functions*

Listing 1: Example of *privacy unit* definition for a database with three tables holding users, orders and items records. Each user is protected individually by designating their `ids` as PID. Orders are attached to a user through the foreign key: `user_id`. Items's ownership is defined the same way by specifying the lineage: `item -> order -> user`.

```
privacy_unit = [
  ("users",[],"id"),
  ("orders",[
    ("user_id","users","id")
  ],"id"),
  ("items",[
    ("order_id","orders","id"),
    ("user_id", "users", "id")
  ],"id")
]
```

Most of the range propagation in *Qrlew* is based on these concepts. It enables a rather simple and efficient range propagation mechanism, leading to better utility / privacy trade-offs.

4.3 Privacy Unit Definition

Tables in a database rarely come properly formatted for privacy-preserving applications. Many rows in many tables may refer to the same individual, hence, *adding or removing an individual* means *adding or removing many rows*. To help the definition of the privacy unit *Qrlew* introduces a small Privacy Unit (PU) description language. As exemplified in listing 1, PU definition associates to each private table in a database a path defining the PID of each row. For a table containing the PU itself, like a `users` table for example, the PU definition will look like `("users",[],"id")`, where `id` is the name of a column identifying the user, like its name. If the database defines tables related to this tables, the way the tables are related should be specified following this scheme: $(\mathtt{tab}_1, path, \mathtt{pid})$ where $\mathtt{tab}_1$ is the name of the table for which the PID is defined, `pid` is the name of the column defining the PID in the table referred by *path* and *path* is a list of elements of the form $[(\mathtt{ref}_1, \mathtt{tab}_2, \mathtt{id}_2), \ldots, (\mathtt{ref}_{m-1}, \mathtt{tab}_m, \mathtt{id}_m)]$ where $\mathtt{ref}_{i-1}$ is a column in $\mathtt{tab}_{i-1}$ — usually a foreign key — referring to $\mathtt{tab}_i$ with a column of referred id $\mathtt{id}_i$ — usually a primary key. Following the path of tables referring to one another, we end up with the table defining the PID (e.g. `users`).

This small PU description language allows for a variety of useful PID scenarii, beyond the simple, but restrictive *privacy per row*.

4.4 Rewriting

Rewriting in *Qrlew*, refers to the process of altering the *computation graph* by substituting computation *sub-graphs* to *Relations* (see figure 3) to alter the properties of the result. This substitution aims to achieve specific objectives, such as ensuring privacy through the incorporation of differentially private mechanisms. The rewriting process (see figure 3) happens in two phases:

- a *rewriting rule allocation* phase, where each *Relation* in the *computation graph* gets allocated a *rewriting rule* (RR) compatible with its input and with the desired output property;
- a *rule application* phase, where each *Relation* is rewritten to a small *computation graph* implementing the logic of the rewriting and stitched together with the other rewritten *Relations*.

Before we decribe these phases into more details, let's define the various properties we may want to guarantee on each *Relation* and the ones we need for the output.

Privacy Properties and Rewriting Rules Each *Relation* can have one of the following properties:

Privacy Unit Preserving (PUP) : A *Relation* is PUP if each row is associated with a PU. In practice it will have a column containing the PID identifying the PU.

Differentially Private (DP) A *Relation* will be DP if it implements a DP mechanism. A DP *Relation* can be safely executed on private data and the result be published. Note that the *privacy loss* associated with the DP mechanism has to be accurately accounted for (see section 5).

Synthetic Data (SD) In some contexts a *synthetic data* version of source tables is available. Any *Relation* derived from other SD or Public *Relations* is itself SD.

Public (Pub) A relation derived from public tables is labeled as such and does not require any further protection to be disclosed.

Published (Pubd) A relation is considered Published if its input relations are either Public, DP, in some cases SD, or Published themselves. It can be considered as Published but with some more care like the need to account for the privacy loss incurred by its DP ancestors.

These properties usually require some rewriting of the computation graph to be achieved. The requirements for a specific *Relation* to meet some property are embodied in what we call: *rewriting rules*. A *rewriting rule* has input requirements, and an achievable output *property* that tells what *property* can be achieved by rewriting provided the input *property* requirements are fulfilled. Each *Relation* can be assigned different *rewriting rules* depending on their nature: *Map*, *Reduce*, etc. and the way they are parametrized.

Rewriting Rules can be — for instance — PU propagation rules of the form:

- $\varnothing \to PUP$ for private *Tables* with a simple rewriting consisting in taking the definition of the privacy unit and computing the PID column.
- $PUP \to PUP$ for *Maps* (or for *Reduce* when the PID is in the `GROUP BY` part) with a rewriting consisting in propagating the PID column from the input to the output.
- $(PUP, PUP) \to PUP$ (or its variants with one published input) for *Join* and a rewriting consisting in adding the PID in the `ON` clause.

Another key *Rewriting Rules* is $PUP \to DP$ for *Reduces*, it simply means that if the parent of the *Relation* can be rewritten as PUP, then we can rewrite the relation to be

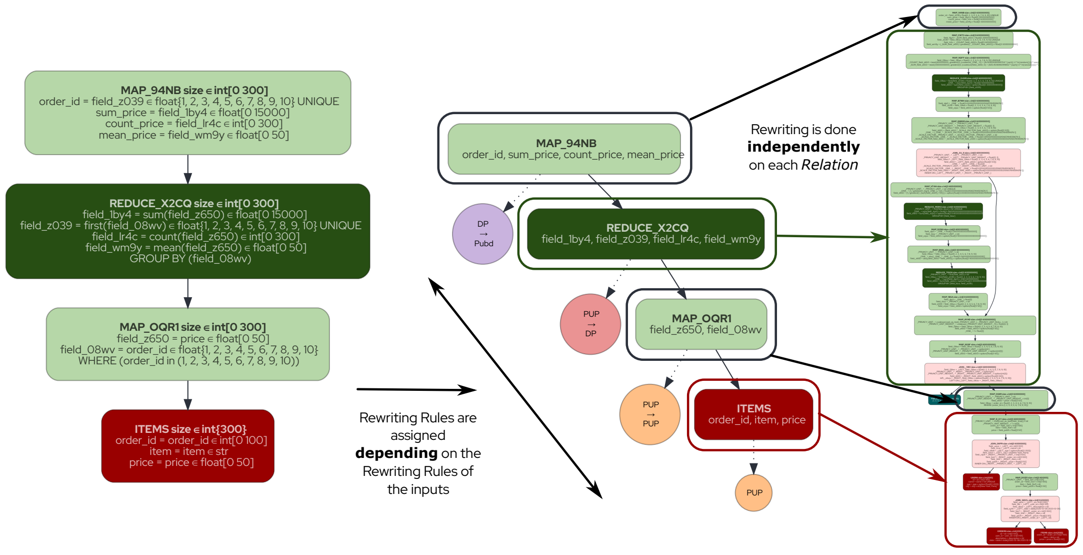

MAP_94NB size ∈ int[0 300]
order_id = field_z039 ∈ float{1, 2, 3, 4, 5, 6, 7, 8, 9, 10} UNIQUE
sum_price = field_1by4 ∈ float[0 15000]
count_price = field_lr4c ∈ int[0 300]
mean_price = field_wm9y ∈ float[0 50]
REDUCE_X2CQ size ∈ int[0 300]
field_1by4 = sum(field_z650) ∈ float[0 15000]
field_z039 = first(field_08wv) ∈ float{1, 2, 3, 4, 5, 6, 7, 8, 9, 10} UNIQUE
field_lr4c = count(field_z650) ∈ int[0 300]
field_wm9y = mean(field_z650) ∈ float[0 50]
GROUP BY (field_08wv)
MAP_OQR1 size ∈ int[0 300]
field_z650 = price ∈ float[0 50]
field_08wv = order_id ∈ float{1, 2, 3, 4, 5, 6, 7, 8, 9, 10}
WHERE (order_id in (1, 2, 3, 4, 5, 6, 7, 8, 9, 10))
ITEMS size ∈ int{300}
order_id = order_id ∈ int[0 100]
item = item ∈ str
price = price ∈ float[0 50]
Rewriting Rules are assigned depending on the Rewriting Rules of the inputs
MAP_94NB
order_id, sum_price, count_price, mean_price
DP → Pubd
REDUCE_X2CQ
field_1by4, field_z039, field_lr4c, field_wm9y
PUP → DP
MAP_OQR1
field_z650, field_08wv
PUP → PUP
ITEMS
order_id, item, price
PUP
Rewriting is done independently on each Relation

Figure 3: The rewriting process happens in two phases: a *rewriting rule allocation* phase, where each node in the *computation graph* gets allocated a *rewriting rule* (RR) compatible with its input and with the desired output property; and a *rule application* phase, where each *Relation* is rewritten according to its allocated RR.

DP by substituting DP aggregations to the original aggregations of the *Reduce*.

One easily see that by simply applying $PUP \to PUP$ and $PUP \to DP$ rules, one can propagate the privacy unit across the computation graph of a *Relation* and compute some DP aggregate such as a noisy sum or average.

Rewriting Rule Allocation The first phase of the rewriting process consists in allocating one and only one rule to each *Relation*. This is done in three steps illustrated in figure 4:

Rule Setting We assign the set of potential rewriting rules to each *Relation* in a computation graph.

Rule Elimination Only feasible rewriting rules are preserved. A rewriting rule that would require a PUP input is only feasible if its input Relation has a feasible rule outputting a PUP *Relation*.

Rule Selection All feasible allocations of one rewriting rule per *Relation* are listed, a score depending on the desired ultimate output property is assigned to each allocation and the highest scoring allocation is selected. Then, a simple split $\left(\frac{\varepsilon}{n}, \frac{\delta}{n}\right)$ of the overall privacy budget (ε, δ) depending on the number of $PUP \to DP$ rules: n is chosen.

In the computation graph, while each node's multiple rewriting rules might suggest a combinatorial explosion in the number of possible feasible allocations, this is mitigated in practice. The pruning of infeasible rules, dictated by the requirement for most relations to have a PUP input for a DP or PUP outcome, significantly reduces the complexity. Hence, despite the theoretical breadth of possibilities, the actual number of feasible paths remains manageable, avoiding substantial computational problems in practice.

Rule Application Once the first phase of rule allocation is achieved, starts the second phase: *rule application*, as illustrated in figure 3. In the allocation phase, a *global rewriting scheme* was set in the form of an allocation satisfying a system of requirements; in the rewriting phase, each rewriting rule is applied *independently* for each *Relation*. This is possible because once a rewriting rule is applied to a *Relation*, the *Relation* is transformed into a computation graph of *Relations* whose ultimate inputs are compatible (same schema, i.e. same columns with same types, plus the new columns provided by the property achieved) with the inputs of the original *Relation* and the ultimate output is also compatible with the output of the original *Relation* so that rewritten *Relations* can be stitched together in a larger graph the same way the original *Relations* were connected: see figure 3.

5 Privacy Analysis

When rewriting, a user can require the output *Relation* to have the *Published* property. All *rewriting rules* with *Published* outputs require their inputs to be either *Public*, *DP*, *SD* or *Published* themselves. We assume synthetic data provided to the system are differentially private, so the privacy of the result depends on the way *Qrlew* rewrites *Reduces* into *DP* equivalent *Relations*.

All *rewriting rules* with *DP* outputs require the input of the *Reduce* to be *PUP* so we can assume a PID column clearly assign one and only one PU to each rows of the rewritten input. The *Reduce* is made DP by:

- Making sure the aggregate columns of the *Reduce* are computed with differentially private mechanisms.

- Making sure the grouping keys of the `GROUP BY` clause are either public or released through a differentially private mechanism.

Protecting aggregation results The protection of aggregation functions is carried out in two steps. Given that all currently supported aggregations (`COUNT`, `SUM`, `AVG`, `VARIANCE STDDEV`) can be reduced to sums, our focus will be on `SUM` aggregations, i.e. the computation of partial sums of a column for different groups: $j \in \{1, \ldots, m\}$, of rows.

Let the column be a vector of N real numbers: $x = (x_1, \ldots, x_N) \in \mathbb{R}^N$. We note: $\pi_k = i \in \{1, \ldots, n\}$ the PID and $g_k = j \in \{1, \ldots, m\}$ the grouping key associated to x_k. We want to compute all the sums:

$$S_j = \sum_{g_k=j} x_k$$

with some DP guarantees. To this end we:

1. *Limit the contribution of each* privacy unit *to the sum*: We represent the contribution of each PU: i, by a vector: s_i whose components are the partial sums within each of the m groups: $s_i = (s_{i,1}, \ldots, s_{i,m})$, where:

   $$s_{i,j} = \sum_{\substack{\pi_k=i \\ g_k=j}} x_k$$

   The s_i's ℓ^2 norms are then clipped to c:

   $$\overline{s_i} = \left(\overline{s_{i,j}}\right)_j = \left(\frac{s_{i,j}}{\max\left(1, \frac{\|s_i\|_2}{c}\right)}\right)_j$$

   See section 8 for more details.

2. *Add gaussian noise to each group*: The clipped contributions are summed and perturbed with gaussian noise $\nu = (\nu_1, \ldots \nu_m) \sim \mathcal{N}\left(0, \sigma^2 I_m\right)$:

   $$\widetilde{S_j} = \sum_{i=1}^{n} \overline{s_{i,j}} + \nu_j$$

   With $\sigma^2 = \frac{2\ln(1.25/\delta)\cdot c^2}{\varepsilon^2}$. Note that the vector of sums has ℓ^2 *Global Sensitivity* of c, so this is an application of the *Gaussian Mechanism* (see: theorem A.1. in (Dwork, Roth et al. 2014)) and the mechanism is ε, δ-differentially private.

Protecting grouping keys When the grouping keys from a are derived from the data, they are not safe for publication. Following (Korolova et al. 2009; Wilson et al. 2019), we use a mechanism called *τ-thresholding* to safely release these grouping keys. Note that, thanks to *range propagation* (see section 4.2), some groups are already public and need no differentially private mechanism to be published. Ultimately, the rewriting of: `SELECT sum(x) FROM table GROUP BY g WHERE g IN (1, 2, 3)` as a DP equivalent will not use *τ-thresholding*, while `SELECT sum(x) FROM table GROUP BY g` will most certainly do if nothing more is known about `g` beforehand.

To summarize the various mechanisms used in *Qrlew* to date: the rewriting of *Reduces* with $PUP \to DP$ rules requires the use of *gaussian mechanisms* and *τ-thresholding* mechanisms; then the DP mechanisms used in all the rewritings are aggregated by the *Qrlew* rewriter as a composed mechanism. The overall privacy loss is aggregated in a RDP accountant (Mironov 2017).

6 Comparison to other systems

There are a few existing open-source libraries for differential privacy.

Some libraries focus on deep learning and *DP-SGD* (Abadi et al. 2016), such as: *Opacus* (Yousefpour et al. 2021), *Tensorflow Privacy* (Google 2023c) or *Optax's DP-SGD* (DeepMind et al. 2020). *Qrlew* has a very different goal: analytics and SQL.

GoogleDP (Google 2023a) is a library implementing many differentially private mechanisms in various languages (C++, Go and Java). *IBM's diffprivlib* (Holohan et al. 2019) is also a rich library implementing a wide variety of DP primitives in python and in particular many DP versions of classical machine learning algorithms. These libraries provide the bricks for experts to build DP algorithms. *Qrlew* has a very different approach, it is a high level tool designed to take queries written in SQL by a data practitioner with no expertise in privacy and to rewrite them into DP equivalent able to run on any SQL-enabled data store. *Qrlew* implemented very few DP mechanisms to date, but automated the whole process of rewriting a query, while these library offer a rich variety of DP mechanism, and give full control to the user to use them as they wish.

Google built several higher-level tools on top of (Google 2023a). *PrivacyOnBeam* (Google 2023b) is a framework to run DP jobs written in Apache Beam with its Go SDK. *PipelineDP* (Google 2022) is a framework that let analysts write Beam-like or Spark-like programs and have them run on Apache Spark or Apache Beam as back-end. It focuses on the Beam and Spark ecosystem, while *Qrlew* tries to provide an SQL interface to the analyst and runs on SQL-enabled back-ends (including Spark, a variety of data warehouses, and more traditional databases). (OpenMined and Google 2023), gives the user a way to write SQL-like queries and have them executed on tables using GoogleDB custom code, so it is not compatible with any SQL data store and support relatively simple queries only.

OpenDP (The OpenDP Team 2023) is a powerful Rust library with a python bindings. It offers many possibilities of building complex DP computations by composing basic elements. Nonetheless, it require both expertise in privacy and to learn a new API to describe a query. Also, the computations are handled by the Rust core, so it does not integrate easily with existing data stores and may not scale well either.

Tumult Analytics (Berghel et al. 2022) shares many of the nice composable design of OpenDP, but runs on Apache Spark, making it a scalable alternative to OpenDP. Still, it require the learning of a specific API (close to that of Spark) and cannot leverage any SQL back-end.

SmartNoise SQL is a library that share some of the design choices of *Qrlew*. An analyst can write SQL queries, but the scope of possible queries is relatively limited: no `JOIN`s, no sub-queries, no CTEs (`WITH`) that *Qrlew* supports. Also, it does not run the full computation in the DB so the integration with existing systems may not be straightforward.

Other systems such as *PINQ* (McSherry 2009) and *Chorus* (Johnson et al. 2020) are prototypes that do not seem to be actively maintained. *Chorus* shares many of the design goals of *Qrlew*, but requires post-processing outside of the DB, which can make the integration more complex on the data-owner side (as the computation happens in two distinct places).

Beyond that, *Qrlew* brings unique functionalities, such as:

- advanced automated range propagation;
- the possibility to automatically blend in synthetic data;
- advanced privacy unit definition capabilities across many related tables;
- the possibility for the non-expert to simply write standard SQL, but for the DP aware analyst to improve its utility by adding `WHERE x < b` or `WHERE x IN (1,2,3)` to give hints to the *Qrlew*;
- all the compute happens in the DB.

This last point comes with some limitations (see section 7), but opens new possibilities like the delegation of the rewriting to a trusted third party. The data practitioner could simply write his desired query in SQL, send it to the rewriter that would keep track of the privacy losses and use *Qrlew* to rewrite the query, sign it, and send it back to the data practitioner that can then send the data-owner, who will check the signature certifying the DP properties of the rewritten query[2].

7 Known Limitations

Qrlew still implements a limited number of DP mechanisms, it is still lacking basic functionalities such as: quantile estimation, exponential mechanisms.

Qrlew relies on the random number generator of the SQL engine used. It is usually not a cryptographic secure random number generator.

Qrlew uses the floating-point numbers of the host SQL engine, therefore it is liable to the vulnerabilities described in (Casacuberta et al. 2022).

8 Conclusion

Qrlew is a novel way of bringing DP to analytics. It brings both a unique set of features, an extended coverage of standard SQL, and full execution in the SQL engine, which opens up new ways to integrate a privacy layer in a data practitioner – data owner relationship. The code is available on github: https://github.com/Qrlew/qrlew with a Python bindings: https://github.com/Qrlew/pyqrlew, a short description: https://qrlew.github.io/
and an interactive demo: https://qrlew.github.io/dp.

[2]A proof of concept is available at: https://github.com/Qrlew/server

References

Abadi, M.; Chu, A.; Goodfellow, I.; McMahan, H. B.; Mironov, I.; Talwar, K.; and Zhang, L. 2016. Deep learning with differential privacy. In *Proceedings of the 2016 ACM SIGSAC conference on computer and communications security*, 308–318.

Archie, M.; Gershon, S.; Katcoff, A.; and Zeng, A. 2018. Who's watching? de-anonymization of netflix reviews using amazon reviews.

Berghel, S.; Bohannon, P.; Desfontaines, D.; Estes, C.; Haney, S.; Hartman, L.; Hay, M.; Machanavajjhala, A.; Magerlein, T.; Miklau, G.; et al. 2022. Tumult Analytics: a robust, easy-to-use, scalable, and expressive framework for differential privacy. *arXiv preprint arXiv:2212.04133*.

Bowen, C. M.; and Snoke, J. 2019. Comparative study of differentially private synthetic data algorithms from the NIST PSCR differential privacy synthetic data challenge. *arXiv preprint arXiv:1911.12704*.

Canale, L.; Grislain, N.; Lothe, G.; and Leduc, J. 2022. Generative Modeling of Complex Data. *arXiv preprint arXiv:2202.02145*.

Casacuberta, S.; Shoemate, M.; Vadhan, S.; and Wagaman, C. 2022. Widespread Underestimation of Sensitivity in Differentially Private Libraries and How to Fix It. arXiv:2207.10635.

Castellon, R.; Gopal, A.; Bloniarz, B.; and Rosenberg, D. 2023. DP-TBART: A Transformer-based Autoregressive Model for Differentially Private Tabular Data Generation. *arXiv preprint arXiv:2307.10430*.

DeepMind; Babuschkin, I.; Baumli, K.; Bell, A.; Bhupatiraju, S.; Bruce, J.; Buchlovsky, P.; Budden, D.; Cai, T.; Clark, A.; Danihelka, I.; Dedieu, A.; Fantacci, C.; Godwin, J.; Jones, C.; Hemsley, R.; Hennigan, T.; Hessel, M.; Hou, S.; Kapturowski, S.; Keck, T.; Kemaev, I.; King, M.; Kunesch, M.; Martens, L.; Merzic, H.; Mikulik, V.; Norman, T.; Papamakarios, G.; Quan, J.; Ring, R.; Ruiz, F.; Sanchez, A.; Sartran, L.; Schneider, R.; Sezener, E.; Spencer, S.; Srinivasan, S.; Stanojević, M.; Stokowiec, W.; Wang, L.; Zhou, G.; and Viola, F. 2020. The DeepMind JAX Ecosystem.

Dwork, C.; McSherry, F.; Nissim, K.; and Smith, A. 2006. Calibrating noise to sensitivity in private data analysis. In *Theory of Cryptography: Third Theory of Cryptography Conference, TCC 2006, New York, NY, USA, March 4-7, 2006. Proceedings 3*, 265–284. Springer.

Dwork, C.; Roth, A.; et al. 2014. The algorithmic foundations of differential privacy. *Foundations and Trends® in Theoretical Computer Science*, 9(3–4): 211–407.

Dwork, C.; Smith, A.; Steinke, T.; and Ullman, J. 2017. Exposed! a survey of attacks on private data. *Annual Review of Statistics and Its Application*, 4: 61–84.

Google. 2022. PipelineDP Library.

Google. 2023a. Google's differential privacy libraries.

Google. 2023b. Privacy On Beam.

Google. 2023c. TensorFlow Privacy.

Grislain, N.; de Sainte Agathe, V.; Cuko, A.; and Sarus Technologies. 2023. Qrlew.

Holohan, N.; Braghin, S.; Mac Aonghusa, P.; and Levacher, K. 2019. Diffprivlib: the IBM differential privacy library. *ArXiv e-prints*, 1907.02444 [cs.CR].

IBM. 2023. Cost of a Data Breach Report 2023.

Johnson, N.; Near, J. P.; Hellerstein, J. M.; and Song, D. 2020. Chorus: a programming framework for building scalable differential privacy mechanisms. In *2020 IEEE European Symposium on Security and Privacy (EuroS&P)*, 535–551. IEEE.

Korolova, A.; Kenthapadi, K.; Mishra, N.; and Ntoulas, A. 2009. Releasing search queries and clicks privately. In *Proceedings of the 18th international conference on World wide web*, 171–180.

Kotsogiannis, I.; Tao, Y.; He, X.; Fanaeepour, M.; Machanavajjhala, A.; Hay, M.; and Miklau, G. 2019. PrivateSQL: A Differentially Private SQL Query Engine. *Proc. VLDB Endow.*, 12(11): 1371–1384.

Liu, Y.; Suresh, A. T.; Yu, F. X. X.; Kumar, S.; and Riley, M. 2020. Learning discrete distributions: user vs item-level privacy. *Advances in Neural Information Processing Systems*, 33: 20965–20976.

McKenna, R.; Miklau, G.; and Sheldon, D. 2021. Winning the NIST Contest: A scalable and general approach to differentially private synthetic data. *arXiv preprint arXiv:2108.04978*.

McSherry, F. D. 2009. Privacy integrated queries: an extensible platform for privacy-preserving data analysis. In *Proceedings of the 2009 ACM SIGMOD International Conference on Management of data*, 19–30.

Mironov, I. 2017. Rényi differential privacy. In *2017 IEEE 30th computer security foundations symposium (CSF)*, 263–275. IEEE.

Narayanan, A.; and Shmatikov, V. 2008. Robust de-anonymization of large sparse datasets. In *2008 IEEE Symposium on Security and Privacy (sp 2008)*, 111–125. IEEE.

Near, J. 2020. Threat Models for Differential Privacy.

OpenMined and Google. 2023. ZetaSQL Library.

Sablayrolles, A.; Wang, Y.; and Karrer, B. 2023. Privately generating tabular data using language models. *arXiv preprint arXiv:2306.04803*.

Sweeney, L.; Abu, A.; and Winn, J. 2013. Identifying participants in the personal genome project by name (a re-identification experiment). *arXiv preprint arXiv:1304.7605*.

The OpenDP Team. 2023. OpenDP Library.

Wilson, R. J.; Zhang, C. Y.; Lam, W.; Desfontaines, D.; Simmons-Marengo, D.; and Gipson, B. 2019. Differentially private SQL with bounded user contribution. *arXiv preprint arXiv:1909.01917*.

Wood, A.; Altman, M.; Bembenek, A.; Bun, M.; Gaboardi, M.; Honaker, J.; Nissim, K.; O'Brien, D. R.; Steinke, T.; and Vadhan, S. 2018. Differential privacy: A primer for a non-technical audience. *Vand. J. Ent. & Tech. L.*, 21: 209.

Yousefpour, A.; Shilov, I.; Sablayrolles, A.; Testuggine, D.; Prasad, K.; Malek, M.; Nguyen, J.; Ghosh, S.; Bharadwaj, A.; Zhao, J.; et al. 2021. Opacus: User-friendly differential privacy library in PyTorch. *arXiv preprint arXiv:2109.12298*.

Appendix

Clipping value used to limit the contribution per user within the aggregations

In our algorithm, the clipping value c is given by:

$$c = k \max(|\min \mathbf{x}|, |\max \mathbf{x}|), \tag{1}$$

where $\min \mathbf{x}$ and $\max \mathbf{x}$ are the known bounds of $\mathbf{x}$ and k is some parameter of the engine that can be used to trade some lower noise for some extra bias.

DP evaluation

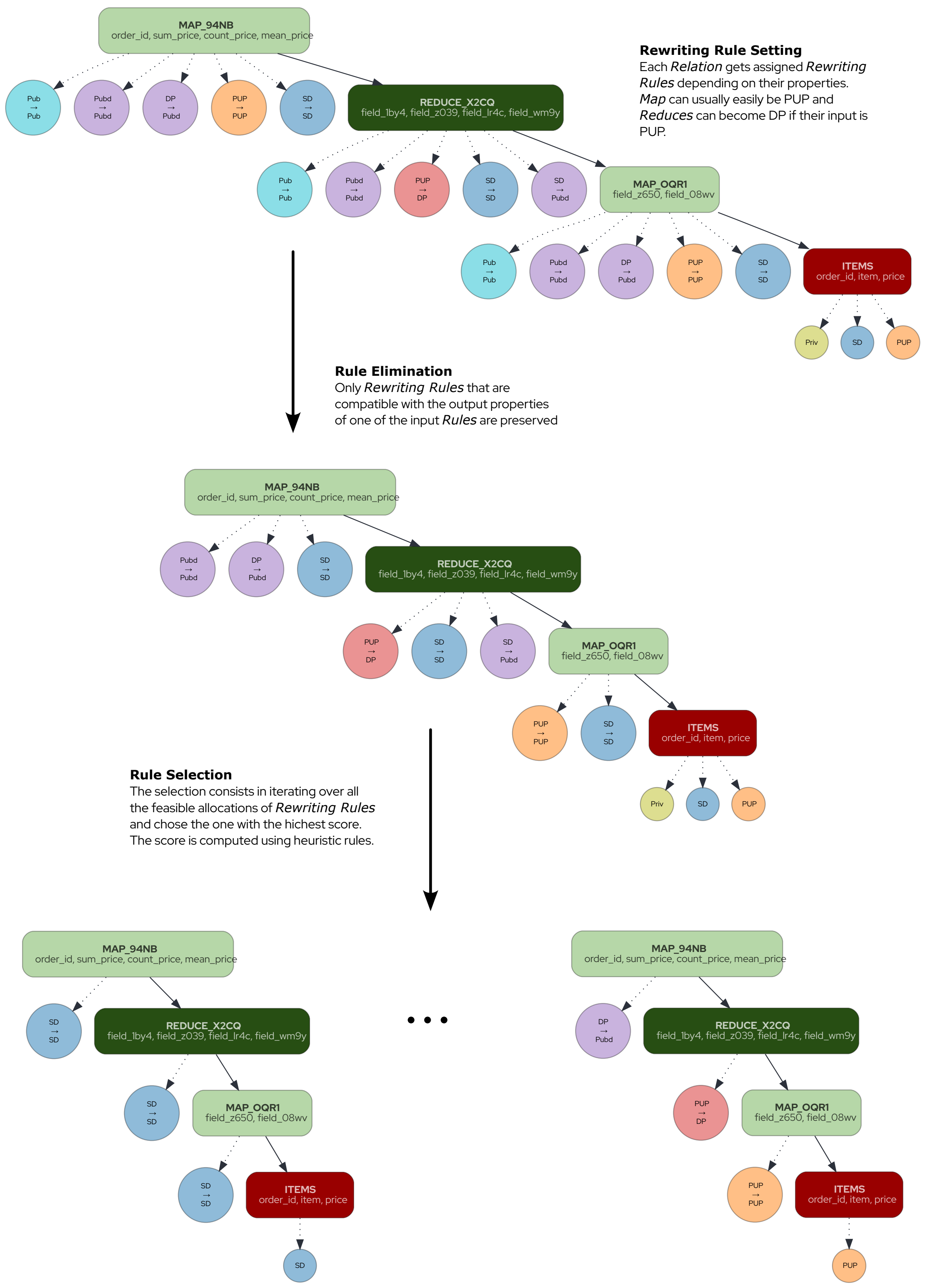
We've assessed the differential privacy of our code following the outlined procedure in Wilson et al.. Our tables were constructed using columns of Halton sequences. Two scenarii were tested:

- In the first scenario, each user possessed exactly one row.
- In the second scenario, a user could have several rows, with the number of rows owned by one user following a normal distribution centered at half the size of the table.

Adjacent databases were created by removing one user compared to the reference database containing all users. Privacy profiles of the underlying distributions were computed using the formula:

$$\delta(e^{\varepsilon}) = \sup_{k} \sup_{x \in \mathbb{R}} f_D(x) - e^{\varepsilon} f_{D_k}(x), \tag{2}$$

In this context, D represents the distribution of results when the query is executed on the entire dataset, and D_k corresponds to the distribution when the query is run on the dataset excluding the data owned by the user k. The algorithm inputting the (ε, δ) parameters is indeed (ε, δ)-differentially private if $\delta(e^{\varepsilon})$ in smaller than δ. We have verified this property holds true for various queries.



Rewriting Rule Setting
Each *Relation* gets assigned *Rewriting Rules* depending on their properties. *Map* can usually easily be PUP and *Reduces* can become DP if their input is PUP.

Rule Elimination
Only *Rewriting Rules* that are compatible with the output properties of one of the input *Rules* are preserved

Rule Selection
The selection consists in iterating over all the feasible allocations of *Rewriting Rules* and chose the one with the hichest score. The score is computed using heuristic rules.

Figure 4: The rewriting happens in three steps: *Rule Setting* when we assign the set of potential rewriting rules to each *Relation* in a computation graph; *Rule Elimination*, when only feasible rewriting rules are preserved; and *Rule Selection*, when an actual allocation is selected.