PrivJail: Enforcing Differential Privacy in Pythonic Data Processing

Shumpei Shiina, Sho Nakatani
Toyota Motor Corporation, Japan
{shumpei_shiina,sho_nakatani}@mail.toyota.co.jp

Takumi Hiraoka, Kenjiro Taura
The University of Tokyo, Japan
{hiraoka,tau}@eidos.ic.i.u-tokyo.ac.jp

1 INTRODUCTION

Due to the risk of privacy violations, data curators cannot freely share the data they collect from individuals. As a result, they often remove useful information for anonymization or aggregate data into statistical summaries before providing it to data analysts. However, ideally, data analysts should be able to perform flexible data processing directly on raw data by writing familiar Python scripts.

To address this situation, we propose PrivJail, a Python library that enforces differential privacy (DP) for data analysts. PrivJail provides a strong privacy guarantee by prohibiting any outputs of non-differentially private values, even to data analysts. At the same time, PrivJail allows data analysts to write Python scripts using Pandas-like dataframe operations [19] to process raw data flexibly.

Figure 1 provides an overview of PrivJail's architecture. In PrivJail, non-differential private values are referred to as *prisoners*, which are derived from input private data. Analysts can spend ϵ to release prisoners by applying DP mechanisms. They may continue releasing prisoners until the predetermined privacy budget limit is reached.

PrivJail enforces DP through a two-stage mechanism. These two stages ensure that:

(1) As long as only explicitly provided library calls are used, the output is always guaranteed to satisfy DP.
(2) There is no direct access to prisoners through any means other than the explicitly provided library calls.

In stage (1), each library call ensures that prisoner values are never directly output. Instead, it guarantees that only noisy results that satisfy ϵ-DP are released. To calibrate noise that satisfy ϵ-DP, the library needs to determine the *sensitivity* for each operation. In Python-style data processing, noise is often added after applying multiple data transformations to the input data. In such cases, the library must automatically derive the overall sensitivity of the combined operations based on the sensitivities of individual operations. This process is referred to as *sensitivity tracking*. PrivJail dynamically tracks sensitivity, as proposed in previous research [2, 3, 10, 17].

What is novel about PrivJail is its support for dynamic sensitivity tracking for major Pandas [19] dataframe operations. A dataframe is a tabular, two-dimensional data structure consisting of rows and columns. Unlike relational databases, both rows and columns are ordered in a dataframe. Existing sensitivity tracking methods do not account for the order of records, making them incompatible with many dataframe operations. We discuss sensitivity tracking rules for dataframe operations in section 5.

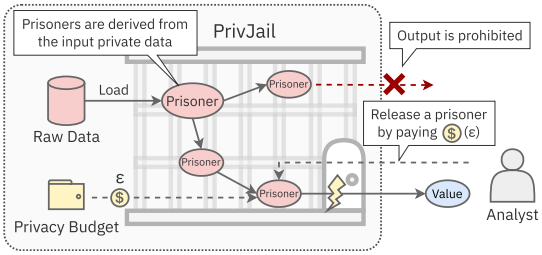

Figure 1: Concept of PrivJail.

In stage (2), only the operations on prisoners assumed in stage (2) are permitted, while any unauthorized access paths to prisoners are strictly prohibited. For example, in Python, it is difficult to restrict access to private class members, making it easy to extract and output prisoner values. One approach to blocking such loopholes is to use Python sandboxes [15], but as discussed in section 2, it is a less secure approach as it depends on Python's language specifications and implementation details.

In this paper, we propose a system design that isolates prisoner values at the process level, separating them from user-provided script execution. Only permitted function calls are accepted via remote procedure calls (RPCs) between processes (section 6). This design ensures that operations on prisoners are allowed only through language-independent RPCs, enabling strong privacy protection.

We demonstrate that PrivJail is expressive enough to implement a DP decision tree algorithm [8] in section 7.

2 RELATED WORK

Static type systems, such as Fuzz [11, 21], DFuzz [9], Duet [18], Solo [1], and Spar [14], enforce DP by automatically deriving sensitivity from low-level programming primitives. In contrast, dynamic sensitivity tracking is used by Python libraries such as DDUO [2], OpenDP [10], and Tumult Analytics [3]. PrivJail follows this dynamic approach for sensitivity tracking. We move a step further by introducing computation rules to support major Pandas-like dataframe operations, as discussed in section 5. This is not straightforward because dataframes offer row ordering unlike relational databases, and operations between dataframes are prevalent.

There exists a spectrum of trust levels granted to data analysts. First, when query inputs are restricted to SQL ([4, 12, 13, 22]) or domain-specific languages (such as Fuzz [11, 21]), it is easier to design a system that does not trust data analysts. On the other hand, libraries built on general-purpose languages [1, 2, 14, 17] often assume a certain level of trust in data analysts, aiming to prevent unintended mistakes in DP

implementations, following the *honest-but-fallible* model [1, 2]. In such systems, loopholes for direct access to sensitive values are not necessarily eliminated.

Antigranular [15] is a Python library designed to block the aforementioned loopholes. To prevent unintended operations, Antigranular executes user-provided scripts within a Python sandbox. However, this approach heavily relies on the correctness of the sandbox implementation, making it less secure. In fact, several vulnerabilities have already been reported in RestrictedPython library [7]. Since sandboxes rely heavily on language specifications and runtime implementations, they have a large attack surface and struggle to keep up with evolving language features. As discussed in section 6, PrivJail's approach to protecting sensitive data is language-independent and therefore considered more secure.

3 PRELIMINARIES

The mechanism M satisfies *ϵ-differential privacy (ϵ-DP)* [5] if, for any *neighboring databases* $D \sim D'$ and any output set S, the following condition holds:

$$P[M(D) \in S] \leq \exp(\epsilon) \cdot P[M(D') \in S]$$

Neighboring databases differ by exactly one row.

For simplicity, we focus on a function $f : \mathbb{D} \to \mathbb{R}$ that takes a database as input and outputs a real number. The *global sensitivity* of such functions is defined as follows:

$$\Delta f = \max_{D \sim D'} |f(D) - f(D')|$$

Sensitivity is used to determine the magnitude of noise in differential privacy. For example, the *Laplace mechanism* satisfies ϵ-DP by adding noise drawn from the Laplace distribution $Lap(\Delta f/\epsilon)$.

4 OVERVIEW AND USAGE OF PRIVJAIL

PrivJail assumes that each row in the input table corresponds to a single individual, enforcing event-level DP. In other words, PrivJail does not currently support user-level DP [6]. The current implementation supports only pure DP and does not yet incorporate advanced composition techniques.

To use PrivJail in Python, users can import PrivJail and its Pandas interface as follows:

```
import privjail as pj
from privjail import pandas as pd
```

To load a CSV file:

```
>>> df = pd.read_csv("filename.csv")
```

The returned dataframe is an instance of the `PrivDataFrame` class, which prohibits direct output of its content.

```
>>> print(df)
Prisoner(<class 'pandas.core.frame.DataFrame'>, distance=1)
```

In PrivJail, such values are referred to as *prisoners*, distinguishing them from normal *public values*. The `PrivDataFrame` class is a subclass of the abstract `Prisoner` class.

To retrieve the row count from a private dataframe:

```
>>> df.shape[0]
Prisoner(<class 'int'>, distance=1)
```

The `df.shape` property returns a tuple of the row count and column count. The row count is a prisoner (`SensitiveInt` class), while the column count is a public value. To release a prisoner, users must apply a DP mechanism.

To observe the row count using the Laplace mechanism:

```
>>> pj.laplace_mechanism(df.shape[0], eps=0.1)
32561.65454862431
```

This output is a normal floating-point number (public value).

To compute the mean of a column:

```
>>> df["age"].mean(eps=0.1)
privjail.util.DPError: The domain is unbounded. Use clip().
```

However, this results in an error because the input domain is unbounded.

Users can set a bounded domain for a column using `clip()`:

```
>>> df["age"].clip(0, 120).mean(eps=0.1)
38.52147065037341
```

PrivJail's private dataframes manage domains for each column, which are accessible via `df.domains`. Besides clipping, users can specify column domains using a schema file, which can be loaded alongside the data source. A schema file (e.g., json) may contain data types, ranges, and potential categorical values for each column.

Users can check the currently consumed privacy budget for each data source by:

```
>>> pj.consumed_privacy_budget()
{'filename.csv': 0.3}
```

Data curators can set an upper limit on privacy budget, preventing further analysis once the limit is reached.

5 DYNAMIC SENSITIVITY TRACKING

To count the number of people over 40 years old:

```
>>> pj.laplace_mechanism(df[df["age"] > 40].shape[0], eps=0.1)
13445.401235025374
```

This follows the typical Pandas-style filtering notation, but automatically determining that the sensitivity of this operation is 1 is nontrivial. This difficulty arises because the above filtering operation consists of multiple underlying primitives:

```
s1  = df["age"] # Select a column (series) by column name
s2  = s1 > 40   # Element-wise comparison with a scalar value
df1 = df[s2]    # Filtering df by a series of boolean elements
df1.shape[0]    # Retrieve a row count of df
```

A unified rule is necessary to automatically derive the sensitivity of any combination of such primitives.

To address this, we adopt the generalized definition of function sensitivity introduced by Reed and Pierce [21], which accommodates functions that take arbitrary input and output types. According to their definition, a function $f : \tau_1 \to \tau_2$ is *c-sensitive* if, for any input x and x', the following condition holds:

$$d_{\tau_2}(f(x), f(x')) \leq c \cdot d_{\tau_1}(x, x')$$

where $d_{\tau_1}(x, x')$ and $d_{\tau_2}(x, x')$ are the distance functions in the domains of types τ_1 and τ_2. This definition enables defining sensitivity for primitives including intermediate data transformations.

The goal of sensitivity tracking is to determine the sensitivity of a composite function $F : \mathbb{D} \to \mathbb{R}$ from its constituent primitive functions $f : \tau_1 \to \tau_2$. This can be achieved through either static analysis [1, 9, 14, 18, 21] or dynamic analysis [2, 3, 10, 17]. Similar to DDUO [2], PrivJail adopts a dynamic approach in Python.

PrivJail dynamically tracks sensitivity by assigning each prisoner a *distance*, representing the maximum possible difference between its values when computed over neighboring databases. The original prisoner is initialized with a distance of 1, as neighboring databases differ by a single row. New prisoners are assigned distances according to *prisoner computation rules* defined for each function. For any c-sensitive function, the rule is expressed as follows:

$$\langle v_1 \rangle_d : \langle \tau_1 \rangle \ \to \ \langle v_2 \rangle_{c \cdot d} : \langle \tau_2 \rangle$$

where $\langle v \rangle_d : \langle \tau \rangle$ represents a prisoner holding a true value $v : \tau$ with distance d assigned. This rule states that the output prisoner's distance is obtained by multiplying the input distance by c.

The prisoner computation rule can also be defined for functions with multiple prisoner inputs. For example, the rule for adding two prisoners holding real numbers can be expressed as follows:

$$\langle v \rangle_d : \langle \mathbb{R} \rangle, \ \langle v' \rangle_{d'} : \langle \mathbb{R} \rangle \ \to \ \langle v + v' \rangle_{d+d'} : \langle \mathbb{R} \rangle$$

This assumes that the two input prisoners are originated from the same data source.

If the prisoner computation rules are correctly applied, it follows that the composite function $F : \mathbb{D} \to \mathbb{R}$ producing a prisoner $\langle v \rangle_d : \langle \mathbb{R} \rangle$ is d-sensitive. Therefore, the distance assigned to a prisoner can be treated as a sensitivity parameter when applying noise. For example, the computation rule for the Laplace mechanism can be written as follows:

$$\langle v \rangle_d : \langle \mathbb{R} \rangle, \ \epsilon : \mathbb{R}_{>0} \ \rightsquigarrow \ v + Lap(\Delta/\epsilon) : \mathbb{R}$$

where " $\rightsquigarrow$ " represents probabilistic computation.

One of our contributions is the definition of a set of prisoner computation rules for major dataframe operations.

5.1 Distance Function of DataFrames

In order to define prisoner computation rules for dataframe operations, we first need to define a distance function for private dataframes. In previous work, such as Fuzz [21], databases are represented as multisets, in which rows are not ordered. However, dataframes generally offer ordering on rows [20]. For example, the methods `df.head()` and `df.tail()` rely on row order.

Hence, we use *edit distance* as the distance between dataframes. Here, the edit distance replaces character matching with row matching in a dataframe. Specifically, we use the *Longest Common Subsequence (LCS) distance*, where adding or deleting a single row increases the distance by 1. Using the LCS distance, we define the distance between two dataframes $df_1, df_2 \in \mathcal{DF}$ as follows:

$$d_{\mathcal{DF}}(df_1, df_2) = |df_1| + |df_2| - 2 \cdot |LCS(df_1, df_2)|$$

where $|df|$ represents the number of rows in the DataFrame, and $LCS(df_1, df_2)$ denotes the LCS between df_1 and df_2.

By adopting the above definition, PrivJail can support a query such as "the average age of the top 100 highest earners" can be expressed as follows.

```
df.sort_values("income").tail(100)["age"].mean(eps=0.1)
```

It is easy to see that the sensitivity of stable sort is 1.

5.2 Row-Wise Operations between DataFrames

Pandas supports many row-wise operations between dataframes, such as addition (`df1 + df2`) and row filtering with a boolean series (`df[df["age"] > 40]`). However, the presence or absence of a single row in one of the dataframes can change the row correspondence, potentially causing all values in the resulting dataframe to differ in the worst case. As a result, the output distance cannot be bounded for arbitrary pairs of input dataframes.

To address this, we restrict row-wise operations to dataframes that have the same row correspondence. A dataframe operation is *row-preserving* if

- it is a row-wise operation that the computation for the i-th row in the input dataframe(s) results in the i-th row in the output dataframe(s), and
- the computation for the i-th row does not depend on values from other rows or row indices (i).

It is guaranteed that dataframes derived from the same source dataframe only through row-preserving operations have the same row correspondence.

In the implementation, a private dataframe is assigned a *row tag* in addition to a distance. For example, the prisoner computation rule for row filtering is expressed as follows:

$$\langle df \rangle_d^p : \langle \mathcal{DF} \rangle, \langle s \rangle_d^p : \langle \mathcal{S} \rangle \ \to \ \langle df[s] \rangle_d^{p^{fresh}} : \langle \mathcal{DF} \rangle$$

where p is the row tag of the inputs and p^{fresh} is a newly generated row tag in this operation. This rule indicates that the input dataframe and series must have the same row tag, and that this operation is not row-preserving, as a new row tag is assigned to the output dataframe.

5.3 Exclusive Partitioning of DataFrames

The method `df.groupby()` partitions the dataframe exclusively based on values of a specified column. A simplified rule for this operation could be expressed as follows:

$$\langle df \rangle_d^p : \langle \mathcal{DF} \rangle, \ c : str \ \to \ \langle [df_i] \rangle_d : \langle [\mathcal{DF}] \rangle$$

where the output is a list of partitioned dataframes $[df_i]$, and its distance is defined as the L1 norm. To retrieve the

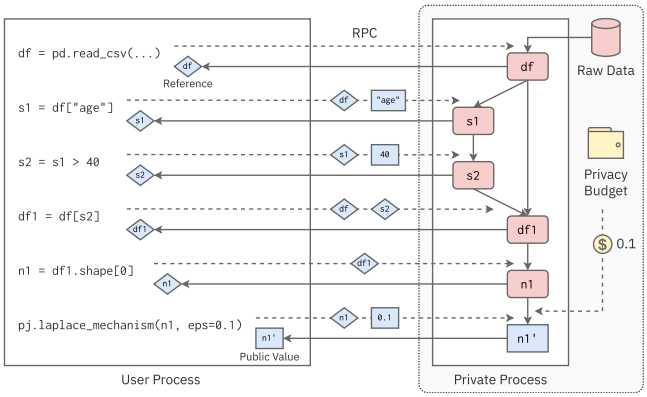

Figure 2: Process isolation.

k-th dataframe from a list of dataframes:

$$\langle [df_i] \rangle_d : \langle [\mathcal{DF}] \rangle,\ k : \mathbb{N}_0 \ \rightarrow\ \langle df_k \rangle_d^{p^{fresh}} : \langle \mathcal{DF} \rangle$$

However, if we retrieve all dataframes from the list individually and sum their row counts, the resulting distance becomes $d \cdot N$, where N is the number of dataframes in the list. Ideally, the result should be equal to the original dataframe's row count with distance d.

We address this distance amplification issue by introducing *distance variables* and *distance constraints*, inspired by their use in Spar [14] for static analysis. In PrivJail, we express a distance as $d = (\eta, C)$, where η is a distance expression and C is a set of distance constraints. A refined rule for the groupby operation can be expressed as follows:

$$\langle df \rangle_d^p : \langle \mathcal{DF} \rangle,\ c : str \ \rightarrow\ [\langle df_j \rangle_{d_j}^{p_j^{fresh}}] : [\langle \mathcal{DF} \rangle]$$

where $d = (\eta, C),\ d_j = (\eta_j, C \cup \{\sum_j \eta_j = \eta\}),\ \eta_j \leftarrow$ fresh var

This outputs a list of prisoners, rather than a prisoner of a list. Each partitioned dataframe is assigned a new distance variable η_j, under the distance constraint $\sum_j \eta_j = \eta$. Consequently, summing the row counts of all partitioned dataframes yields a result distance of $\sum_j \eta_j$, which is equal to the original dataframe's distance η. When applying noise, the sensitivity parameter is calculated as the maximum value of the distance expression η under the constraints C.

6 PROCESS ISOLATION

To prevent the output of prisoners without applying a DP mechanism, we propose a system design that isolates prisoner values from user-provided script execution at the process level. We illustrate its architecture in Figure 2. The user-provided Python script is executed in the *user process*, while prisoner values reside exclusively in the *private process*. When a private dataframe is loaded, a reference to it is returned from the private process to the user process, while its actual content remains in the private process. Subsequent computations involving prisoners are internally forwarded to the private process via remote procedure calls (RPCs), with prisoner references passed as arguments. Once a DP mechanism is applied, the resulting values (public values) can be directly returned to the user process.

Due to process isolation, unintended operations in user-provided scripts or vulnerabilities in the Python runtime do not directly affect prisoners in the private process. The only way to access prisoners is to make proper RPC requests, which are language-independent and predefined in PrivJail.

```python
def DiffPID3(df, attrs, class_attr, d, eps):
    # df         : Input PrivDataFrame (categorical elements)
    # attrs      : Set of explanatory attributes (column names)
    # class_attr : Target attribute (column name)
    # d          : Maximum tree depth
    # eps        : Epsilon consumed in this function
    eps_each = eps / (2*(d+1))
    return build_DiffPID3(df, attrs, class_attr, d, eps_each)

def build_DiffPID3(df, attrs, class_attr, d, eps):
    t = max([len(df.domains[a].categories) for a in attrs])
    C = len(df.domains[class_attr].categories)
    N = noisy_count(df, eps)

    if len(attrs) == 0 or d == 0 or N/(t*C) < (2**0.5)/eps:
        # Record the mode of class_attr values in a leaf node
        class_counts = {c: noisy_count(df_c, eps) \
                        for c, df_c in df.groupby(class_attr)}
        best_class = max(class_counts, key=class_counts.get)
        return LeafNode(best_class)

    # Choose best_attr that gives the best split
    qs = {a: quality_fn(df, a, class_attr) for a in attrs}
    best_attr = pj.exponential_mechanism(qs, eps=eps)

    # Partition df based on best_attr values and recurse
    node = InnerNode(best_attr)
    for category, df_child in df.groupby(best_attr):
        child_node = build_DiffPID3(df_child, \
                attrs - {best_attr}, class_attr, d - 1, eps)
        node.add_child(category, child_node)
    return node

# Return a noisy row count (>= 0)
def noisy_count(df, eps):
    return max(0, pj.laplace_mechanism(df.shape[0], eps=eps))

# Implement "Max operator" as a quality function
def quality_fn(df, split_attr, class_attr):
    s = 0
    for _, df_c in df.groupby(split_attr):
        s += df_c[class_attr].value_counts(sort=False).max()
    return s
```

Figure 3: DiffPID3 algorithm [8] written in PrivJail.

7 CASE STUDY: DP DECISION TREE

To demonstrate the expressiveness of PrivJail, we faithfully implemented a DP decision tree algorithm called DiffPID3 [8], as shown in Figure 3. To summarize the key points of this program:

- It involves recursion with conditional branching based on input data. Since static analysis and tracing are difficult in this case, dynamic analysis is particularly effective.
- Dynamic sensitivity tracking (section 5) is effective to automatically calibrate noise to satisfy ϵ-DP. For example, in `quality_fn()`, multiple computations on prisoners are combined and passed to the exponential mechanism [16].
- `df.groupby()` exclusively partitions the dataframe, and each partitioned dataframe is then recursively processed. To properly account for the total privacy budget, it is necessary to consider the parallel composition theorem (not covered in this abstract).

PrivJail is available at https://github.com/privjail/privjail.

REFERENCES

[1] Chiké Abuah, David Darais, and Joseph P. Near. 2022. Solo: a lightweight static analysis for differential privacy. *Proceedings of the ACM on Programming Languages* 6, OOPSLA2 (Oct. 2022), 699–728.

[2] Chike Abuah, Alex Silence, David Darais, and Joseph P. Near. 2021. DDUO: General-Purpose Dynamic Analysis for Differential Privacy. In *Proceedings of the 2021 IEEE 34th Computer Security Foundations Symposium* (Dubrovnik, Croatia) *(CSF '21)*. 1–15.

[3] Skye Berghel, Philip Bohannon, Damien Desfontaines, Charles Estes, Sam Haney, Luke Hartman, Michael Hay, Ashwin Machanavajjhala, Tom Magerlein, Gerome Miklau, Amritha Pai, William Sexton, and Ruchit Shrestha. 2022. Tumult Analytics: a robust, easy-to-use, scalable, and expressive framework for differential privacy. https://arxiv.org/abs/2212.04133.

[4] Microsoft Corporation and Harvard University. 2020. SmartNoise. https://smartnoise.org. Last accessed on 2024-12-19.

[5] Cynthia Dwork. 2006. Differential Privacy. In *Proceedings of the 33rd International Colloquium on Automata, Languages, and Programming* (Venice, Italy) *(ICALP '06)*. 1–12.

[6] Cynthia Dwork, Moni Naor, Toniann Pitassi, Guy N. Rothblum, and Sergey Yekhanin. 2010. Pan-Private Streaming Algorithms. In *Proceedings of the 1st Symposium on Innovations in Computer Science* (Beijing, China) *(ICS '10)*. 66–80.

[7] Zope Foundation. 2024. RestrictedPython. https://github.com/zopefoundation/RestrictedPython. Last accessed on 2024-12-27.

[8] Arik Friedman and Assaf Schuster. 2010. Data mining with differential privacy. In *Proceedings of the 16th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining* (Washington, DC, USA) *(KDD '10)*. 493–502.

[9] Marco Gaboardi, Andreas Haeberlen, Justin Hsu, Arjun Narayan, and Benjamin C. Pierce. 2013. Linear dependent types for differential privacy. In *Proceedings of the 40th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (Rome, Italy) *(POPL '13)*. 357–370.

[10] Marco Gaboardi, Michael Hay, and Salil Vadhan. 2020. A Programming Framework for OpenDP. https://projects.iq.harvard.edu/files/opendp/files/opendp_programming_framework_11may2020_1_01.pdf. Last accessed on 2024-12-18.

[11] Andreas Haeberlen, Benjamin C. Pierce, and Arjun Narayan. 2011. Differential Privacy Under Fire. In *Proceedings of the 20th USENIX Security Symposium* (San Francisco, CA, USA) *(USENIX Security '11)*. 1–15.

[12] Noah Johnson, Joseph P. Near, and Dawn Song. 2018. Towards practical differential privacy for SQL queries. *Proceedings of the VLDB Endowment* 11, 5 (Jan. 2018), 526–539.

[13] Ios Kotsogiannis, Yuchao Tao, Xi He, Maryam Fanaeepour, Ashwin Machanavajjhala, Michael Hay, and Gerome Miklau. 2019. PrivateSQL: a differentially private SQL query engine. *Proceedings of the VLDB Endowment* 12, 11 (July 2019), 1371–1384.

[14] Elisabet Lobo-Vesga, Alejandro Russo, Marco Gaboardi, and Carlos Tomé Cortiñas. 2024. Sensitivity by Parametricity. *Proceedings of the ACM on Programming Languages* 8, OOPSLA2 (Oct. 2024), 415–441.

[15] Oblivious Software Ltd. 2023. Antigranular. https://www.antigranular.com. Last accessed on 2024-12-09.

[16] Frank McSherry and Kunal Talwar. 2007. Mechanism Design via Differential Privacy. In *Proceedings of the 48th Annual IEEE Symposium on Foundations of Computer Science* (Providence, RI, USA) *(FOCS '07)*. 94–103.

[17] Frank D. McSherry. 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* (Providence, Rhode Island, USA) *(SIGMOD '09)*. 19–30.

[18] Joseph P. Near, David Darais, Chike Abuah, Tim Stevens, Pranav Gaddamadugu, Lun Wang, Neel Somani, Mu Zhang, Nikhil Sharma, Alex Shan, and Dawn Song. 2019. Duet: an expressive higher-order language and linear type system for statically enforcing differential privacy. *Proceedings of the ACM on Programming Languages* 3, OOPSLA (Oct. 2019), 172:1–172:30.

[19] The pandas development team. 2024. pandas-dev/pandas: Pandas (v2.2.3). https://doi.org/10.5281/zenodo.13819579.

[20] Devin Petersohn, Stephen Macke, Doris Xin, William Ma, Doris Lee, Xiangxi Mo, Joseph E. Gonzalez, Joseph M. Hellerstein, Anthony D. Joseph, and Aditya Parameswaran. 2020. Towards scalable dataframe systems. *Proceedings of the VLDB Endowment* 13, 12 (July 2020), 2033–2046.

[21] Jason Reed and Benjamin C. Pierce. 2010. Distance makes the types grow stronger: a calculus for differential privacy. In *Proceedings of the 15th ACM SIGPLAN International Conference on Functional Programming* (Baltimore, Maryland, USA) *(ICFP '10)*. 157–168.

[22] Royce J. Wilson, Celia Yuxin Zhang, William Lam, Damien Desfontaines, Daniel Simmons-Marengo, and Bryant Gipson. 2020. Differentially Private SQL with Bounded User Contribution. In *Proceedings on Privacy Enhancing Technologies* (Virtual Event) *(PETS '20)*. 230–250.