Technical Report MPI-SWS-2021-001

Paul Francis

Feb. 02, 2021

Specification of Diffix Dogwood

This technical report contains a description of Diffix Dogwood data anonymization and the known attacks against which Diffix Dogwood defends. This description is taken as-is from the online documentation used for the 2020 bounty program at https://www.gda-score.org/diffix-dogwood-specification-and-attacks/.

The rules for the bounty program are in MPI-SWS tech report MPI-SWS-2021-002.

The customer documentation associated with Diffix Dogwood is in MPI-SWS tech report MPI-SWS-2021-003.

Specification for Diffix Dogwood

The anonymization algorithm underlying Aircloak Insights is Diffix. The current version of Diffix is Dogwood. This section describes the operation of Diffix Dogwood. It explains how and why Diffix Dogwood protects against known attacks, thus providing strong anonymity.

Table Of Contents

        * Add protection against divide-by-zero attacks
        * Add protection against square root of negative numbers
    - Handle DB answer
        * Noise Layers
        * Determine seeds
        * Low count suppression
        * Value flattening and noise addition
            · Operation of sum
            · Operation of avg
            · Operation of min and max
        * Reporting suppression
            · Bucket merging
        * Reporting noise
        * Suppress aggregate values
- Aggregation function count distinct
- Aggregation function stddev

Overview

Key concepts

Anonymization in Diffix Dogwood has two main aspects, **SQL constraints** and **answer perturbation**. The goal is to allow as much SQL as possible, especially commonly used SQL, and to minimize answer perturbation while maintaining strong anonymity. Strong anonymity is achieved when an attacker's guess as to the attributes of individual users either has low confidence with high probability, or high confidence with low probability.

In order to safely allow as much SQL as possible, Diffix Dogwood constrains SQL in a fine-grained fashion. While some entire SQL features are prevented (for instance window functions), more often an SQL feature is allowed but constrained in terms of what other features it may be used in combination with, how frequently it may be used, or even what constant values it may be used with.

Perturbation takes the form of suppressing answers and distorting answers. The latter occurs through both adjusting extreme column values (flattening) and adding noise. Key mechanisms include:

- **Answer suppression:** Answers that pertain to too few distinct users are suppressed.
- **Sticky layered noise:** Diffix Dogwood adds noise taken from a Gaussian distribution. The noise is sticky in that identical query conditions generate identical noise. The noise is layered in that each query condition contributes a separate noise value (which are summed together to produce the final noise value). There are in fact two types of noise layers, *UID-noise* and *static-noise*. Static-noise is sticky in the sense that the same filter condition,

for instance `age = 20`, will typically generate the same noise sample. UID-noise is sticky in the sense that the same filter condition combined with the same set of selected users will typically generate the same noise sample.

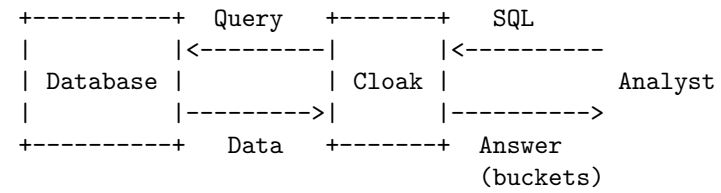
- **Extreme value flattening:** When a few individual users contribute an extreme amount to an answer (relative to other users), then the values contributed by those users are reduced (or increased if negative) to be comparable with values contributed by other users.

Diffix Dogwood can report how much answer suppression has taken place and how much noise was added to an answer. Diffix Dogwood cannot, however, report how much extreme value flattening has taken place.

Deployment

Diffix Dogwood is deployed as a function or device that sits between an analyst (or application) and an un-anonymized database. In the Aircloak system, this device is referred to as Insights Cloak. In this document, for readability, we refer to it simply as the cloak.

An SQL interface is exposed to the analyst. The database may be an SQL database, or may be some other kind of data store. The cloak translates the analyst SQL into the appropriate query language.



If the SQL query requests an aggregate (for instance `SELECT age, count(*) FROM table GROUP BY age`), then each row returned in the answer is referred to as a *bucket* in this document. In this example, each tuple `(age, count)` would be a separate bucket.

Main Pipeline

The cloak operates query-by-query: it accepts SQL queries from the analyst (or application), parses the SQL and checks to ensure that it is allowed, potentially modifies the analyst SQL to adhere to SQL constraints, requests the appropriate data from the database (SQL' in the figure), determines if buckets need to be suppressed, and adds noise to unsuppressed buckets before returning them to the analyst.

The cloak also maintains some internal state about data in the database. This state, produced by querying the database, is used to determine when certain constraints may be relaxed. The analysis queries producing the state are run when the system is initially setup, and are periodically repeated.

Prior to query: - Initialization of internal state (SQL constraints)

At query time: - Handle incoming SQL (SQL constraints) - Generate DB query (Answer perturbation prep) - Handle DB answer (Answer perturbation)

Database configuration

The database consists of tables. Tables have *columns* and *rows*.

Each table must be configured as being *personal* or *non-personal*. Personal tables contain user data that must be anonymized, whereas non-personal tables contain other data. (Note that in the Aircloak Insights product, the term "datasource" is used to refer generically to different types of databases.)

When queries include one or more personal tables, the cloak must be able to associate every row of data with one distinct user. This can happen in one of two ways: 1. The personal table is configured with one and only one `user_id` column (often referred to as the `uid` in this document). This uniquely identifies the entity that must be protected (typically but not necessarily a natural person). 2. The personal table can be linked to a personal table with a `user_id` through one or more `JOIN` operations. The columns that may be used for this linking, known as `key` columns, must be explicitly configured.

The cloak allows `JOIN (...) ON` operations using the configured `key` or `user_id` columns only (see Illegal JOINs).

Version Differences

This section outlines the major differences between versions.

From Cedar to Dogwood

- The functions `floor`, `ceil`, and `cast to int` were defined as implicit ranges, with the effect that clear restrictions are applied to them.

From Birch to Cedar

- The method for computing noise and flattening was changed from a UID-based approach to a stats-based approach (see Value flattening and noise addition).
- Several mechanisms to defend against side-channel attacks were added (JOIN timing, divide-by-zero, square root of negative numbers, overflow/underflow).
- Added internal state (shadow table, isolating columns, column min and max) and related restrictions.
- Numerous restrictions were added.

Initialization of internal state

The cloak establishes cached internal state that it uses to determine which SQL operations are allowed on which columns. They are:

Shadow table

Holds a list of the X values that occur most frequently in the column (X=200), so long as the value has at least 10 distinct users. These constants are evaluated as exact (not noisy) counts. (Note it would be slightly better to use noisy counts here, but this is not an important issue.) Values in this list may be used in negands (negative AND conditions) and posors (positive OR conditions) without limitations. aircloak/aircloak#2486 aircloak/aircloak#2972 aircloak/aircloak#3322

This table is refreshed every 30 days. aircloak/aircloak#3357

Isolating column label

The cloak labels columns as isolating if 80% or more of the values in the column are associated with only a single user. aircloak/aircloak#2485 aircloak/aircloak#2738 aircloak/aircloak#3396

This table is refreshed every 60 days. aircloak/aircloak#3357

Safe math functions

There are a variety of functions that can throw an error in some databases, for instance divide-by-zero, numeric overflow, datetime overflow, and taking the square root of a negative number. In such databases, the error manifests itself as a error message transmitted to the analyst, which can be exploited by an attacker (see Error generation attacks).

If the database allows user-defined exception handlers, then these are installed in the database, either by the cloak when it connects, through configuration of the database prior to cloak connection. If not, then the cloak modifies SQL to prevent exceptions. In both cases, errors are prevented from being transmitted to the analyst. See Determine if safe math functions are needed.

Per column min and max values

The safe math functions unfortunately slow down query execution. To mitigate this, the cloak conservatively estimates when a math function *might* result in an exception, and only executes the safe functions in these cases.

In order to make this estimate for numeric and date/time/datetime overflow exceptions, the cloak records an anonymized minimum and maximum value for each numeric and date/time/datetime column. These recorded values are not the true minimum and maximum values, because an attacker could then detect

these values through a series of queries that detect when a safe function was executed through a timing attack. aircloak/aircloak#3780

With high (but not 100%) probability, the approximated min and max exceed the true min and max. For numeric columns they are computed as follows:

1. Take the top/bottom 1000 values from a given column (note that a given user can have multiple values)
2. Discard all but the biggest/smallest value for each user out of those
3. Randomly select a cutoff with configured mean (default 20) and stddev (default 5)

- The cutoff won't be lower than a configured min (default 10)
- The PRNG for the cutoff is seeded with the table and column names and salt, giving the same result on every run

4. Ignore a number of the top/bottom values from step 2 equal to the selected cutoff
5. Take the top/bottom value after that and find the closest snapped value above/below it (see Snapped Ranges)
6. Given both a min and max computed this way create an extended bound

- If min is negative while max is positive, multiply both by 10
- If both are positive, multiply max by 10 and divide min by 10
- If both are negative, multiply min by 10 and divide max by 10

7. If there are not enough values to compute either min or max, set the bounds to :unknown and always use safe math functions

The procedure for `date`, `time`, and `datetime` columns is similar but differs in the following two ways: 1. In step 5, the closest snapped value is computed from '1900-01-01' (rather than zero, as is the case with numeric columns). 2. In step 6, the range is expanded by adding 50 years (rather than multiplying by 10). aircloak/aircloak#3794

Handle incoming SQL

Supported SQL

The following shows what SQL is supported. Any received SQL that has syntax outside of this specification is rejected.

{% include "sql/syntax.md" %}

Rejecting queries

Even if the SQL satisfies the above syntax, there are a number of specific uses that can be rejected.

Inequalities

While the cloak allows inequality operators (`> | >= | < | <=`), there are restrictions on how they may be used. In general, the cloak requires that both sides of an inequality be specified. This is necessary to prevent difference attacks using range creep with averaging.

This document refers to all inequalities that are bounded on both sides as *ranges*.

In addition to the ranges that can be specified with `BETWEEN` and inequality operators, some functions implicitly define a range. For instance, the function `hour` defines a range of width one hour. The definition of range in this document includes implicit ranges. These are: `hour`, `minute`, `second`, `year`, `quarter`, `month`, `day`, `weekday`, `date_trunc`, `floor`, `ceil`, `cast to int`, `round`, `trunc`, and `bucket`.

Normally the lower bound of a range is inclusive, while the upper bound is exclusive. The exception to this is when the upper bound corresponds to the maximum value allowed by the column type. aircloak/aircloak#4462

Inequalities that are not ranges (not bounded on both sides) are possible in the special case of Conditions with two columns.

Clear conditions (IN, range, negative conditions)

The cloak may require that certain conditions are *clear*. The primary purpose of clear conditions is so that the cloak can seed noise layers through SQL inspection rather than by floating the column value. The following operators must be clear:

- negative conditions (`col <> val`) including `NOT IN` and `IS NOT NULL`.
- `IN` (`col IN (val1, val2)`), though note that the column is floated for the purpose of seeding the static noise layer (not the per-element UID-noise layers).
- range (`col BETWEEN val1 and val2`), including implicit ranges.
- expressions within aggregation functions (for instance `sum(col + 1)`).

Negative conditions must be clear because it would not be efficient to float a value that is being excluded by the condition.

The column in a range can be floated. However, by generating the seed materials through inspection of the constants `val1` and `val2`, the intent of the condition is captured, and a given condition will always produce the same static seed.

The term "clear" implies that it is clear from SQL inspection alone what the semantics of the conditions are, and therefore how to seed the corresponding noise layers.

Clear conditions also have the effect of reducing the attack surface since it gives an attacker fewer mechanisms to work with. For isolating columns, the cloak forces clear conditions for all operators. aircloak/aircloak#4091

In earlier versions of Diffix, clear conditions were limited to only columns on the left-hand-side: no column functions were allowed (for instance `col = val`). This restriction was subsequently relaxed to allow operators that are on one hand frequently useful to analysts, but on the other hand do not give analysts an adequately large attack surface (for instance `lower(col) = val`). aircloak/aircloak#2982

`IN` and range are limited to only constants on the right-hand-side. Negative conditions may additionally have columns on the right-hand-side.

The left-hand-side can have either the native column (no transformations), or the native column with a single instance of the following functions:

For text columns, or columns that have been cast as text: `lower`, `upper`, `trim`, `ltrim`, `btrim`, and `substring`. aircloak/aircloak#2039 aircloak/aircloak#2091

For date, datetime, and time columns: `hour`, `minute`, `second`, `year`, `quarter`, `month`, `day`, `weekday`, and `dow`. aircloak/aircloak#2881 aircloak/aircloak#2982

A column in a clear condition cannot have undergone transformations prior to the condition. For instance, in the following query, the `IN` condition must be clear, but since there is a prior transformation (`numeric + 1` in the sub-query), the condition is unclear and the query rejected.

```
SELECT COUNT(*)
FROM (
    SELECT numeric + 1 AS number
    FROM table) x
WHERE number IN (1, 2, 3)
```

Too much math

In general In order to prevent SQL backdoor attacks, the cloak limits the complexity of certain math in queries. We are particulary concerned with discontinuous functions combined with constants because they can be coerced into acting as discrete logic. However, there are many ways of obtaining discontinuous functions, for instance string manipulations (for instance `left`, `ltrim`) followed by casting to numbers, or datetime functions (for instance `year` or `hour`). In addition, functions can be coerced into constants, as for instance `pow(col1, col2-col2)` is 1.

We take a conservative approach and limit the number of expressions containing a restricted function and a constant, or more than one restricted function to a total of 5.

The restricted functions and operators include `+`, `-`, `/`, `*`, `^`, `%`, `abs`, `ceil`, `floor`, `length`, `round`, `trunc`, `btrim`, `left`, `ltrim`, `right`, `rtrim`, `substring`, `year`, `quarter`, `month`, `day`, `weekday`, `hour`, `minute`, `second`, and `date_trunc`.

With isolating columns In order to prevent Linear program reconstruction attacks, the cloak places even more restrictions on expressions associated with isolating columns. An isolating column is one where 80% or more of the values in the column are associated with only a single user. These columns would otherwise be susceptible to linear program reconstruction attacks because they may be used to select groups of individual users.

The operators `IN` (with more than one element) and `<>` are disallowed for isolating columns. This prevents easily isolating individual users, or controling which individual users comprise an answer.

For the remaining operators, the cloak requires that *all conditions* that operate on isolating columns to be clear.

String functions

In order to generally reduce the attack surface available with string functions (`lower`, `upper`, `trim`, `ltrim`, `btrim`, and `substring`), the following limitations apply:

1. Columns which have undergone a string function cannot be combined with other transformations.
2. String functions cannot be applied to columns that have undergone multiple casts.
3. Results of string functions can only be compared with constants or with other columns.

Datetime intervals

The cloak allows `date`, `time`, and `datetime` math using intervals (for instance, `datetime_col + interval 'PT1H2M3S'`). In order to limit the number of cases that need to be checked for datetime overflow, the cloak limits overflow math to `datetime_col + interval` and `datetime_col - interval`. Math operations involving multiple intervals (`interval + interval`) or intervals and constants `integer * real` are prohibited. aircloak/aircloak#3794

LIKE and NOT LIKE

`[NOT] [I]LIKE` have similar restrictions as string functions. The first two restrictions that apply to String functions apply to `[NOT] [I]LIKE` as well (cannot be combined with other transformations, cannot be used with columns that have mutiple casts).

For isolating columns, the `%_` wildcards associated with `[NOT] [I]LIKE` is limited to using only the `%` wildcard (`_` is not allowed), and only in the first or last position of the string. For example, `col LIKE '%stuff%'` is allowed, but not `col LIKE 'stu%ff'`. Non-isolating columns have no limitation on the `[NOT] [I]LIKE` wildcards. aircloak/aircloak#2973

Limitations due to shadow table

To mitigate noise exploitation attacks through chaff conditions, negative conditions and `IN` right-hand-side elements are disallowed if the corresponding constant does not appear in the shadow table. aircloak/aircloak#2273 aircloak/aircloak#3557

Note that this mechanism does not prevent isolation in all cases. The shadow table is based on the complete column. Any given query, however, may have conditions that cover only part of the column. A constant may therefore appear in the shadow table and still not have any matching rows in the context of a given query. For instance, the condition `gender <> 'F'` may not be chaff in the `pro_football_players` table, but may be chaff when combined with `tournament = 'fifa world cup'`. However, executing an effective attack by exploiting this mechanism is essentially not possible (see Shadow table exploitation attack).

Conditions with two columns

The cloak allows conditions with two columns. However, each column may appear only once in the condition. Conditions with more than two columns are not allowed.

Two-column conditions with inequalities do not require that the condition be a range (have both a lower and upper boundary). For instance, the condition `WHERE col1 < col2` is allowed.

The condition of the form `date_constant BETWEEN date_col1 AND date_col2` is allowed, but the constant must be aligned to the month. aircloak/aircloak#4487

Illegal JOINs

The cloak rejects any query that has `JOIN (...) ON` conditions that are not explicitly allowed (see Database configuration). All `JOIN (...) ON` conditions must be simple `column1 = column2` expressions. This prevents attacks that would otherwise try to exploit known relationships between columns to generate extra noise samples. aircloak/aircloak#898 aircloak/aircloak#2704

Modifying queries

Normally queries with SQL that does not conform to the requirements of the cloak are rejected. Cases where the query is modified rather than rejected are described here.

Snapped Ranges

In order to prevent difference attacks using range creep with averaging, all ranges must conform to pre-designated widths and offsets. aircloak/aircloak#4486

In the case of numeric data types, the widths must fall within the following infinite sequence of widths: `[..., 0.1, 0.2, 0.5, 1, 2, 5, 10, 20, 50, ...]`. The offset must fall on an even multiple of the width, or an even multiple plus 1/2 of the width. The following are allowed:

- `col BETWEEN 10 and 15`: width 5, offset 10
- `col BETWEEN 7.5 and 12.5`: width 5, offset 7.5 (shifted by 2.5)
- `col BETWEEN -0.001 and -0.002`: width 0.001, offset -0.002

The following are not allowed: * `col BETWEEN 10 and 13`: width 3 (disallowed width) * `col BETWEEN 8 and 13`: width 5, offset 8 (not multiple of 5 or 2.5)

In the case of datetime types, the widths must correspond to natural datetime boundaries (year, month, week, day, hour, minute, second). Within each such boundary, the following additional widths are allowed:

- `[1, 2, 3, 6, 9, 12, ...]` months
- `[1, 2, 5, 10, 15, 20]` days
- `[1, 2, 6, 12]` hours
- `[1, 2, 5, 15, 30]` minutes
- `[1, 2, 5, 15, 30]` seconds

In addition, the cloak has a number of current date/time functions, including `current_date`, `current_time`, `current_datetime`, `now` and `current_timestamp`. These can be used in inequalities with columns, for instance `now() < col` and `now() between col1 and col2`. Current date/time functions are snapped by truncating to the current day (time `00:00:00`). aircloak/aircloak#3474

Aggregation functions sum, min, max, count

Different anonymization mechanisms are used for different aggregation functions. In this section, we describe anonymization for `sum(col)`, `count(*)`, `count(col)`, `count(DISTINCT uid)`, `min(col)`, and `max(col)`. A key requirement here is to minimize the amount of data that is transferred from the database to the cloak. The implementation of these functions limits the transfer to one row of data per bucket.

Generate DB query

As a running example for this section, we use the following query submitted by an analyst to the cloak:

```
SELECT frequency, sum(amount) AS amount_sum
FROM transactions
WHERE abs(acct_district_id) = 1
GROUP BY 1
```

The query subsequently generated by the cloak for the database is in essence the following (simplified for readability):

```
1  SELECT frequency,
2         SUM(amount_sum) AS amount_sum,
3         COUNT(uid) AS count_duid,
4         MIN(uid) AS min_uid,
5         MAX(uid) AS max_uid,
6         COUNT(amount_sum) AS amount_count,
7         MIN(amount_sum) AS amount_min,
8         MAX(amount_sum) AS amount_max,
9         STDDEV(amount_sum) AS amount_stddev,
10        MIN(acct_district_id_min) AS acct_district_id_min,
11        MAX(acct_district_id_max) AS acct_district_id_max,
12        SUM(acct_district_id_cnt) AS acct_district_id_cnt
13 FROM
14   (SELECT uid, frequency, sum(amount) AS amount_sum,
15           MIN(acct_district_id) AS acct_district_id_min,
16           MAX(acct_district_id) AS acct_district_id_max,
17           COUNT(*) AS acct_district_id_cnt
18    FROM transactions
19    WHERE ABS(acct_district_id) = 1
20    GROUP BY uid, frequency) t1
21 GROUP BY frequency
```

Gather bucket statistics

In order to compute the noise for aggregate answers and the noisy threshold for suppression, the cloak needs to have certain statistics about how much users contribute to each aggregate, especially users that contribute the most. The statistics are:

- `avg(contribution)`
- `min(contribution)`
- `max(contribution)`
- `stddev(contribution)`

This information is obtained by:

1. Adding the `uid` column to the selected columns (line 14) and `GROUP BY` (line 20),
2. Computing the contribution per user (`amount_sum` in line 14, since the aggregate in this example is `sum(amount)`,
3. Computing the four statistics (`amount_min` line 7, `amount_max` line 8, `amount_stddev` line 9, and as part of computing the average, `amount_count` line 6).

On line 14, `sum(amount)` would be replaced with `count(*)`, `count(amount)`, or

`1` if the analyst aggregate is `count(*)`, `count(amount)`, and `count(DISTINCT uid)` respectively.

Gather seed materials

Diffix adds noise samples (layers) per query filter condition. There are two types of noise layers, UID-noise and static-noise. Static-noise is "sticky" in the sense that the same filter condition, for instance `age = 20`, will typically generate the same noise sample. UID-noise is sticky in the sense that the same filter condition combined with the same set of selected users will typically generate the same noise sample.

The mechanism for generating the same noise sample is to generate the same seed for the PRNG (Pseudo-Random Number Generator). Each seed is composed of several components which we call the *seed materials* (see Determine seeds).

The seed material for both the UID-noise and the static-noise contain the values that are filtered for (in the case of positive conditions) or filtered against (in the case of negative conditions) by the condition. In some cases, it is possible to determine that value through inspection of the SQL itself. For instance, the selected value of `age = 20` is 20. In other cases, however, it is not clear from inspection, and so the selected value or values are requested from the database. We refer to this as "floating" the column.

In our running example, there are two filter conditions. One is in the `WHERE` clause (`ABS(acct_distrit_id) = 1`, line 19), and one is the selected column `frequency` (line 1). The selected column `frequency` is a filter condition because each resulting bucket has only a single `frequency` value. The cloak does not interpret the filtered value for the `WHERE` condition by inspection, but instead floats the column `acct_district_id`.

The static-noise is seeded by the minimum and maximum column values, and so these must be floated. In the running example, lines 15 and 16 select the per-UID per-bucket min and max for column `acct_district_id`, and lines 10 and 11 select the per-bucket min and max. aircloak/aircloak#3274

The UID-noise is seeded by (among other things): 1. the minimum UID (line 4), 2. the maximum UID (line 5), 3. the distinct number of UIDs (line 3), and 4. the number of rows (lines 12 and 17).

Determine if safe math functions are needed

A conservative analysis is run to determine if a given math expression in the query might result in an exception. Examples would be determining if a column in a divisor could be zero, or determining whether a `pow()` function on a column could lead to overflow. The analysis makes the assumption that the column value from the database does not exceed the approximated min and max (see Per column min and max values). aircloak/aircloak#3689

If a math function might lead to an exception, then the function in the query is replaced with a safe version (see Safe math functions).

If the analysis determines that a math function cannot lead to an exception, then the possible values for the column are bounded in the query so that the min and max assumptions are correct. An example of how this is done is as follows:

```
CASE WHEN amount < 0 THEN 0
     WHEN amount > 1000000 THEN 1000000
     ELSE amount
END
```

Here the assumed min and max for the `amount` column are 0 and 1000000. The `amount` column is replaced with this `CASE` statement, ensuring that any value from the `amount` column does not exceed these bounds.

For simplicity, these substitutions are not shown in the example database query.

Add protection against JOIN timing attack

As described in the JOIN timing attack, an analyst can strongly influence query execution time in the database by including `JOIN` expressions that may return an empty table. aircloak/aircloak#3691 aircloak/aircloak#4083 aircloak/aircloak#4164

To prevent this, the cloak modifies all but the last JOIN expression so that at least one row is always returned. This is done with a `UNION` operation:

```
original_expression UNION inverse_original_expression
```

The second expression of the `UNION` returns a single row if and only if the first expression, which is the original expression, returns no rows. Following is an example:

```
SELECT uid
FROM accounts
WHERE lastname = 'Zamora'
  AND birthdate = '1996-11-16'
UNION ALL
SELECT *
FROM (
   SELECT uid
   FROM accounts
   LIMIT 1
   OFFSET 0
   ) t
WHERE NOT EXISTS
(
   SELECT uid
   FROM accounts
```

```
   WHERE lastname = 'Zamora'
     AND birthdate = '1996-11-16'
)
```

The original `JOIN` expression is the part above the `UNION`. It is repeated within the `WHERE NOT EXISTS` expression. If the original `JOIN` expression returns nothing, the the `UNION` expression returns a single row.

Add protection against divide-by-zero attacks

To defend against Divide by zero attacks, the cloak modifies any expression having a column in the denominator of a divide operation so that NULL is generated instead of an exception.

This is done using a `CASE` statement that detects when the denominator is zero (or nearly so), and substitutes a NULL. For example, in the condition:

```
WHERE 1/column = 0.1
```

the expression `1/column` is substituted with:

```
CASE WHEN column < 1.0e-100 THEN NULL
     ELSE 1/column
END
```

Add protection against square root of negative numbers

To defend against Square root of a negative number attacks, the cloak modifies any expression having a column in a square root opeation that NULL is generated instead of an exception.

This is done using a `CASE` statement that detects when the expression in the square root operation is negative, and substitutes a NULL. For example, in the condition:

```
WHERE sqrt(column) = constant
```

the expression `column` is substituted with:

```
CASE WHEN column < 0 then NULL
     ELSE sqrt(column)
END
```

Handle DB answer

Noise Layers

A noise layer is an individual Gaussian noise sample. The total noise added to an aggregate value in each bucket of a query answer is the sum of the individual noise samples. Although noise layers are taken from a Gaussian distribution, their values are not random. Rather, the PRNG used to generate each noise

sample is seeded in such a way that identical query or answer conditions generate the same noise sample. We refer to this property as being *sticky*.

There are two types of noise layers, *static* and *uid*. The difference between them is based on what is used to seed each layer. Static noise layers typically (though not always) depend only on the semantics of the individual SQL query filter condition. In other words, the `WHERE` condition `age = 10` will always produce the same static noise layer independent of what UIDs comprise the answer. By contract, UID-noise layers always depend on which UIDs comprise the answer.

Each filter condition has a static noise layer, and most filter conditions additionally have a UID-noise layer. The following are considered to be filter conditions:

- WHERE clauses
- HAVING clauses in subqueries
- GROUP BY clauses
- All columns in the top-level SELECT clause
- Range WHERE clauses, for instance a pair of `a >= x` and `a < y`, is considered one filter instead of two
- Each element of `IN (a, b, c)` is treated as one distinct filter condition, while the entire expression taken together is treated as another filter condition

In addition, there is a generic noise layer for queries that otherwise have no noise layer (because no filter conditions). The query `SELECT count(*) FROM table` is such a query.

The aggregation function `count(col)` is given an additional UID-noise layer. The purpose of this noise layer is to defend against the Difference attack with counting NULL.

Ranges, as well as the implicit ranges `bucket()`, `round()`, and `trunc()`, do not have a UID-noise layer, only a static noise layer. This is to defend against noise exploitation attacks through chaff conditions. aircloak/aircloak#3931 aircloak/aircloak#4110

Conditions comparing two columns (`col1 <>|<|<=|>|>= col2`) have no noise layers. `JOIN (...) ON` conditions have also no noise layers. In both cases, the conditions are constrained to the point where there are no known attacks that can effectively exploit these conditions. aircloak/aircloak#3930 aircloak/aircloak#2704

The `IN` operator (with more than one element) has a single static noise layer for the entire expression, and a UID-noise layer for each element.

Determine seeds

Upon receiving the database answer, for each one-column condition, the cloak knows the following information: aircloak/aircloak#4018

1. column name
2. table name

For each bucket, the cloak additionally knows:

3. min column value
4. max column value
5. min UID
6. max UID
7. count of distinct UIDs
8. count of rows

The min and max column values are obtained either by floating (see Gather seed materials), or by inspection of the condition itself in the query. The following cases use inspection:

- If `col = val`, `col <> val`, or `clear_function(col) <> val`, then both the min and max are set to `val`.
- If `col BETWEEN val1 AND val2`, then min is set to `val1`, and max to `val2`.
- If `col IN (val1, ..., valN)`, then for each of the N per-element UID-noise layers, both min and max are set to `valN`. (Note that for the static noise layer, floating is used.)
- If a selected column (`SELECT col FROM ...`), then min and max are set to the value returned by the database.
- If an implicit range, then min is set to the left edge of the range, and max is set to the right edge of the range.
- In case of text datatypes, `val` is converted to lower case (`lower(val)`) for the purpose of seeding.

Most noise layers are seeded with at least the following:

- A canonical name of the column in the form {“table”, “column”}
- A secret salt (random value) that is established when the cloak is initialized and not changed afterwards
- The min and max column value

The negative conditions `<>`, `NOT LIKE`, and `NOT ILIKE` add the symbol `:<>` to the seed material. Note that `NOT IN` conditions are converted to their equivalent `<>` forms.

UID-noise layers additionally have the minimum and maximum UID values in the seed material.

The additional UID-noise layer assigned to `count(col)` is seeded with the secret salt, the canonical column name, and the minimum and maximum UID values.

The generic noise layer is seeded with the `count(DISTINCT uid)` value and the secret salt.

Low count suppression

The cloak obtains the number of distinct UIDs that contribute to each bucket (see Gather seed materials). Buckets that have too few distinct UIDs are suppressed (not returned to the analyst in the query answer).

A noisy threshold is used to determine if a bucket should be suppressed. The threshold value is determined by seeding a Gaussian random number with:

- Minimum UID
- Maximum UID
- Number of distinct UIDs

using the paramters `mean = 4` and `stddev = 0.5`. In the example of section Generate DB query, these three aggregates are gathered in lines 3-5.

If the number of distinct UIDs is less than the threshold value, then the bucket is suppressed.

Value flattening and noise addition

The cloak perturbs aggregate function values. This section describes how the `sum(col)`, `count(*)`, `count(col)`, `count(DISTINCT uid)`, `min(col)`, and `max(col)` aggregates are perturbed. aircloak/aircloak#2561 aircloak/aircloak#2807

The magnitude of perturbation is related to how much individual users contribute to the aggregate. The magnitude depends on two types of users, *heavy contributors*, and *extreme contributors*.

Extreme contributors are defined as the 1 or 2 users with the highest contribution. Heavy contributors are defined as the 3 to 5 users with the next highest contribution.

Conceptually the perturbation should do the following. First, it should modify the contributions of the extreme contributors so that they are similar to those of the heavy contributors. Second, it should add noise proportional to the magnitude of the heavy contributions.

For example, suppose a column contains user salary. Suppose that the average salary is 100K, but there are four or five users with salaries between 450K and 550K, and only one user with a salary higher than 550K. That user's salary is 2M. The group of four or five users would be regarded as heavy contributors, and the user with 2M salary would be regarded as an extreme contributor.

We want to adjust the value of the extreme contribution (2M) so that it is more like those of the heavy contributors. If the aggregate is `sum()`, this effectively means reducing the reported sum by around 1.5M. We call this adjustment "flattening". We want to then add noise proportional to that of the heavy contributors, for instance proportional to around 500K.

The rational for these perturbations is as follows. Assume that an analyst is able to forumlate two queries, one whose answer does not include the victim (the left query), and one that may include the victim (the right query). The analyst wants to compare answers to determine if the victim is present in the right query or not. If the right answer is greater than the left answer, for instance, the analyst may assume that the victim is present in the right answer. This is the basic Difference attack.

There needs to be enough noise that the analyst's confidence in a guess as to the presence or absence of the victim in the right query is low.

Alternatively, we can assume a worst-case where an analyst knows every value in a column except that of a single user (the victim), and the analyst wants to determine if the victim is included in the database or not. If a query taking the sum of the column is greater than the sum of all known values, then again the analyst can assume that the victim is in the database, and again there needs to be enough noise to keep the analyst's confidence low.

Since the victim may be a heavy contributor, there needs to be enough noise to obscure the presence or absence of a heavy contributor. The analyst can, however, in many cases estimate the magnitude of noise that was added. Indeed, the cloak explicitly reports this to the analyst for the purpose of better understanding the cloak's answer. If there is an extreme contributor, then merely the magnitude of noise can reveal with high confidence the presence or absence of the user.

To prevent this, the cloak flattens the extreme user or users values to hide the extreme user by blending it in with the heavy users. The noise is then based on the average contributions of the heavy users. Unfortunately, the cloak cannot report how much flattening took place, since that would defeat the purpose. As a result, aggregate values can on occasion be very inaccurate, and the analyst needs to be aware of this possibility.

In Diffix Birch, the perturbations for extreme and heavy contributors were done explicitly. That is, the cloak would explicitly determine the contributions of each user, identify the extreme and heavy contributors, and compute the flattening and noise accordingly. This process would sometimes incur a heavy performance penalty, as the database would have to convey per-user information to the cloak.

To improve performance, Diffix Dogwood uses per-bucket rather than per-user information. In most cases, this substantially reduces the amount of data transferred from the database to the cloak. Unfortunately, this means that the cloak can no longer explicitly determine the extreme and heavy contributors. Instead, the cloak uses aggregate statistics about the users in each bucket to *estimate* the extreme and heavy contributors.

The statistics are these (line numbers taken from example query):

- `col_avg`: average contribution of all users (line 6 divided by line 3)
- `col_std`: standard deviation of users' contributions (line 9)
- `col_min`: min contribution of all users (line 7)

- `col_max`: max contribution of all users (line 8)
- `col_cnt`: the number of distinct UIDs of the data (line 3)

These user statistics are taken for each aggregate function in the query.

For each noise layer, a Gaussian sample is generated with `mean=0` and standard deviation `sd=1` (as seeded in Determine seeds). These are summed together to produce the `base_noise`.

Operation of sum The following algorithm describes the algorithm for computing perturbation for `sum(col)`, `count(*)`, `count(col)`, and `count(DISTINCT uid)`. The same algorithm works for all four aggregate functions. `count(*)` is the sum where each row has value 1. `count(col)` is the sum where each non-NULL row has value 1 and NULL rows have value 0. `count(DISTINCT uid)` is the sum of users where each user contributes value 1.

The perturbation for sum is:

```
perturbation_sum = (base_noise * sum_sd) - flatten
```

where `sum_sd` is a factor applied to the `base_noise` and is approximately proportional to the average heavy contribution, and `flatten` is the adjustment due to an approximation of the difference between the extreme contributions and the average heavy contribution.

Note that for `count(DISTINCT uid)`, all users contribute exactly the same, so `flatten=0`, and `sum_sd=1`.

The sum advertised to the analyst is:

```
noisy_sum = true_sum + perturbation_sum
```

where `true_sum` is the exact (non-noisy) sum from the database (line 2).

If we assume for the moment that all user contributions are positive, then the intuition behind the algorithm follows from these two observations: 1. The larger the standard deviation (`col_std`), the larger the average heavy contribution. 2. The larger the max value relative to the average plus several standard deviations, the greater the amount of flattening.

This is perhaps explained through several examples. Let's suppose that these examples are for `sum(salary)`.

First, consider a case where `col_min=col_max=col_avg=100K` and `col_std=0`. In this case, all users have the same salary of 100K, we want a noise factor of `sum_sd=100K` (equal to the contribution of any user), and there is no flattening.

Now suppose that `col_min=95K`, `col_avg=100K`, `col_std=3K`, and `col_max=10M`. The relatively low standard deviation `col_std=3k` implies that the majority of salaries are clustered tightly around the average of `col_avg=100K`. In contrast to this, however, the maximum salary `col_max=10M` is far away from this cluster, suggesting that a great deal of flattening is needed.

Now suppose `col_min`, `col_avg`, and `col_max` stay the same as in the previous example, but `col_std=200K`. This implies that the salaries are no longer so tightly clustered around the average, but rather there are some relatively high salaries that can be regarded as heavy hitters. As the standard deviation increases, all other things being equal, the noise factor `sum_sd` would increase, and the amount of flattening would decrease.

The following algorithm was generated through a process of trial and error, with a goal towards minimizing perturbation while keeping attacker confidence low.

The algorithm has three constants, set as follows:

- `factor=4`
- `avg_scale=1`
- `top_scale=0.5`

The algorithm operates over positive values, negative values, or a combination. The algorithm actually makes two computations, one for values above the average, and one for values below the average. `sum_sd` is the largest of the two `sum_sd` values computed above and below. By contrast, `flatten` is a combination of the flattening for above and below. (For a distribution with positive values, almost all of the flattening comes from the computation above. For distributions with both positive and negative values, however, flattening above and below can in fact cancel each other out!)

Of course, the `col_std` computed by the cloak is comprised of values both above and below the `col_avg`. So we use the `col_max` and `col_min` to estimate how much of the `col_std` comes from above and how much comes from below, as follows:

```
std_above = col_std * ((col_max - col_avg)/(col_max - col_min))
std_below = col_std * ((col_avg - col_min)/(col_max - col_min))
```

Next we try to estimate the rough values for heavy contributors above and below. These are computed using the estimated standard deviations and `factor=4` constant:

```
heavy_above = col_avg + (factor * std_above)
heavy_below = col_avg - (factor * std_below)
```

```
flatten_above = col_max - heavy_above
flatten_below = col_min - heavy_below
```

Note that either flatten value may be positive or negative.

The final `flatten` value is then:

```
flatten = flatten_above + flatten_below
```

Flattening mimics the effect of modifying values in the data. This in turn should impact the computed average. To reflect this impact, we adjust `col_avg` as follows:

```
if (flatten > 0) {
  col_avg = col_avg - (flatten / col_cnt)
}
```

Finally, `sum_sd` is computed as:

```
sum_sd = max(
  abs(avg_scale * col_avg),
  abs(top_scale * heavy_above),
  abs(top_scale * heavy_below)
)
```

Remembering that `avg_scale=1` and `top_scale=0.5`, the above computation was found to reduce overall noise while keeping attacker confidence low. aircloak/aircloak#3013

Operation of avg The average aggregator is computed as the `sum(col)`/`count(col)`.

Operation of min and max The aggregators `min(col)` and `max(col)` are set to the `edge_below` and `edge_above` values from Operation of Sum, with the exception that `min(col)` may never be greater than `avg(col)`, `max(col)` may never be less than `avg(col)`, and `min(col)` may never be greater than `max(col)`. aircloak/aircloak#3642 aircloak/aircloak#3643 aircloak/aircloak#3392

Reporting suppression

In order to help analysts understand query results, the cloak provides information about suppressed buckets. Data from suppressed buckets are merged to generate a new bucket, referred to as a *star* bucket. Star buckets are subject to the same bucket suppression and noise adding mechanisms as any bucket. In other words, a star bucket itself may be suppressed. As such, absence of a star bucket does not necessarily imply that no suppression has taken place.

The following simplistic and idealized example illustrates how suppression is reported. Here `count` is a count of distinct UIDs. For simplicity, we assume here that buckets with 4 or fewer distinct UIDs are suppressed. In reality, this decision is based on a noisy threshold (Low count suppression). This example also assumes that any given UID appears only once. As a result, the count resulting from merging two buckets is the sum of the two individual counts.

Suppose that the data transferred from the database to the cloak is as follows:

x	y	count
a	1	10
a	2	2
a	3	3
b	2	7
b	4	8

x	y	count
b	1	4
b	7	3
b	9	4
b	5	4
c	1	3
d	2	3

Two columns, `x` and `y` have been selected.

The suppressed buckets are these:

x	y	count
a	2	2
a	3	3
b	1	4
b	7	3
b	9	4
b	5	4
c	1	3
d	1	3

Suppression reporting attempts to expose as many column values as possible. It gives priority to reporting column values from left to right as they appear in the `GROUP BY`. Assuming that `x` is to the left of `y` in the `GROUP BY`, the suppressed buckets are merged as follows:

x	y	count
a	*	5
b	*	15
c	*	3
d	*	3

The `x` values are preserved, and the `y` values are replaced with a star (`*`) symbol. The first two buckets will be reported to the analyst, but the second two buckets must themselves be suppressed, and so are extracted:

x	y	count
c	*	3
d	*	3

These buckets are merged into the following bucket, losing the `x` column values in the process:

x	y	count
*	*	6

This bucket is not suppressed, and so is reported to the analyst.

The final set of buckets is therefore:

x	y	count
a	1	10
a	*	5
b	2	7
b	4	8
b	*	15
	*	6

These buckets are reported to the analyst with noisy counts.

Three of the seven 'y' values have been suppressed, and two of the four x values. Had the query placed the `y` column on the left, then the suppression process would have produced this:

y	x	count
1	a	10
1	*	7
2	b	7
2	*	5
4	b	8
	*	14

Note that somewhat more detail regarding the `y` column values is revealed.

Although this example used the star symbol for both the text and integer column, in practice it would be illegal to return a star symbol, which is text, in a numeric or datetime column. In these cases, the cloak would report a `NULL` value.

Bucket merging Merged buckets are processed like any other bucket: they are checked for suppression and aggregate answers are perturbed. As such, the information needed for this processing must be preserved when buckets are merged (lines 2-9 in the example query). aircloak/aircloak#3368

It is not possible to preserve this information perfectly. Merging is essentially an `OR` operation, and as such any users that happen to be in both buckets cannot be accounted for because the cloak does not have per-user information. Therefore the merge is designed to produce approximate values.

The information that needs to be generated is: `col_sum, count_duid, min_uid, max_uid, col_count, col_min, col_max, col_stddev`. The cloak also records the set of UIDs (`uid_set`) generated in previous merges during the processing of an answer. Assuming the merging of two buckets with subscripts 1 and 2, the values for the merged bucket are computed as follows:

- `merged_uid_set = union(uid_set1, uid_set2)`
- `merged_col_sum = col_sum1 + col_sum2`
- `merged_col_min = min(col_min1, col_min2)`
- `merged_col_max = max(col_max1, col_max2)`
- If the two uid ranges do not overlap:
  - `merged_col_count = col_count1 + col_count2`
  - `merged_count_duid = count_duid1 + count_duid2`
- If the two uid ranges touch (minimum of one set equals the maximum of the other set):
  - `merged_col_count = col_count1 + col_count2 - 1`
  - `merged_count_duid = count_duid1 + count_duid2 - 1`
- Otherwise, merged count is estimated as the maximum count plus a quarter of the minimum count (because some collisions could occur):
  - `merged_col_count = max(col_count1, col_count2) + min(col_count1, col_count2) / 4`
  - `merged_count_duid = max(count_duid1, count_duid2) + min(count_duid1, count_duid2) / 4`
- `merged_avg = merged_col_sum / merged_col_count`
- For the standard deviation, we use the formula: `sd(v) = sqrt(sum(v^2) / count - avg(v)^2)` We first extract the sums of squared values: `sum_sqrs1 = (col_stddev1^2 + avg1^2) * col_count1` and `sum_sqrs2 = (col_stddev2^2 + avg2^2) * col_count2`, we then add them together to get the merged sum of squared values, resulting in: `merged_sd = sqrt(merged_sum_sqrs / merged_col_count - merged_avg^2)`

Reporting noise

For each of the aggregation functions `count()`, `sum()`, `avg()`, and `stddev()`, the cloak provides corresponding `aggr_noise()` functions (`count_noise()` etc.). The ideal here would be to report the exact standard deviation of the actual noise added by the cloak. Doing so, however, opens the cloak to a Noise signal attack. This is because the `min` and `max` values from the set of aggregated values have a substantial influence on the noise standard deviation. In the aggregate value itself, this influence is somewhat hidden by the presence of the other values and by flattening.

To hide the influence in the reported noise, the cloak reports a value close to but different from the true noise standard deviation. The larger the contribution of the extreme user, the larger the distortion. (Note that in any event, the larger the contribution of the extreme user, the greater the amount of flattening, and so the reported noise is not a good representation of the total distortion.)

The section Value flattening and noise addition describes how the standard deviation for noise is computed from the measured values of `col_avg`, `col_std`, `col_min`, `col_max`, and `col_cnt`. When computing a standard deviation for reporting noise, the following computation is used instead:

```
1  col_sum = col_avg * ct
2  new_avg = (col_sum - (col_max - col_avg) - (col_min - col_avg)) / ct
3  sum_of_sqr_diff = col_std^2 * (ct - 1)
4  new_sum_of_sqr_diff = sum_of_sqr_diff - (col_max - col_avg)^2 - (col_min - col_avg)^2
5  new_var = new_sum_of_sqr_diff / (ct - 1)
6  new_std = new_var^0.5
```

The reported noise is then:

```
reported_noise = max(
                     (new_avg + (factor * new_std)) * top_scale,
                     new_avg * avg_scale
                    )
```

Suppress aggregate values

When the number of distinct users is low, the aggregates `min()`, `max()`, `sum()`, `avg()` and `stddev()` can potentially reveal information about the underlying values. In addition, the accuracy can be quite poor. To mitigate these factors, the cloak reports `NULL` when the number of distinct users is less than a noisy threshold with mean 10 and a standard deviation of `(0.5 * L)`, where `L` is the number of noise layers. Note that the bucket itself is still reported: only the aggregate value itself is set to `NULL`.

Aggregation function count distinct

When computing for instance `count(column)`, any non-NULL column value will contribute to the count. Noise must be added for instance to hide the fact that a single user may be included or excluded.

By contrast, when computing `count(DISTINCT column)`, it is not always necessary to add noise. This is because including or excluding a single user may not change the count at all. To take a simple example, suppose that the column is `gender` with possible values 'M' and 'F', and that both genders are well-represented by many users. The answer to `count(DISTINCT gender)` is `2`, and including or excluding any given user does not change that answer. In cases like this we want to return the exact answer.

Suppose on the other hand the column is `purchase_time`, that there are many thousands of different purchase times, and that many of the purchase times are unique to a single user. In this case, the presence or absence of a single user can easily change the true answer, and so noise is needed to hide such users.

To deal with this, the cloak generates a query that first determines, for each UID, the count of column values `count_distinct_vals_per_uid` that are unique to that UID. The cloak then computes the `min`, `max`, `avg`, and `stddev` statistics across all `count_distinct_vals_per_uid`, and uses the same algorithm as with `sum()` to compute noise and flattening. If there are zero column values that have a single UID associated with them, then no noise is added. aircloak/aircloak#3563 aircloak/aircloak#4080

The following example shows how this is done.

Suppose the analyst query is:

```
SELECT column1, COUNT(DISTINCT column2) FROM table GROUP BY 1
```

The cloak would generate a query similar to the following simplified example:

```
 1  SELECT
 2    group_0,
 3    MAX(CAST((uid IS NOT NULL) AS integer) * count_distinct) AS cnt_max,
 4    MIN(CAST((uid IS NOT NULL) AS integer) * count_distinct) AS cnt_min,
 5    AVG(CAST((uid IS NOT NULL) AS integer) * count_distinct) AS cnt_avg,
 6    STDDEV(CAST((uid IS NOT NULL) AS integer) * count_distinct) AS cnt_stddev,
 7    SUM(count_distinct) AS count_distinct
 8  FROM (
 9    SELECT
10      uid,
11      group_0,
12      COUNT(target) AS count_distinct
13    FROM (
14      SELECT
15        column1 AS group_0,
16        column2 AS target,
17        CASE WHEN (COUNT(DISTINCT uid) < 2) THEN MIN(uid) ELSE NULL END AS uid
18        FROM table
19        GROUP BY 1, 2
20    ) AS distinct_values
21    GROUP BY 1, 2
22  ) AS uid_grouping
23  GROUP BY 1
```

The inner-most `SELECT` (lines 14 - 19) does a `GROUP BY` based on distinct values of the selected columns. In line 17, the UID of values that have only one user is recorded.

In the next `SELECT` (lines 9 - 21), at line 12 we count the number of times each

UID appears, as well as the number of times no UID appears (which gives us the number of distinct column values that have more than one user).

In the outer-most `SELECT`, the maximum, minimum, average, and stddev statistics for how often each UID is unique for a `column1` value are computed (lines 3-6), along with the number of distinct values (line 7). Noise and flattening is computed from these statistics as with the algorithm of section Operation of sum.

Low-count suppression is done as with `sum()` (Low count suppression).

Aggregation function stddev

Currently `stddev(col)` is computed with per-UID information as done in Diffix Birch. See prior documentation for more information (https://arxiv.org/pdf/1806.02075.pdf).

List of Known Defended Attacks

This section describes a variety of attacks against anonymization mechanisms. Some of the attacks are general in nature (can be used against multiple mechanisms). Many are specific to Diffix. This list is current to Diffix Dogwood. In what follows, the *cloak* is the device that implements Diffix.

Table Of Contents

Attack criteria

In the following attacks, our criteria for what determines a successful attack are the same as those used by the EU Article 29 Data Protection Working Party Opinion 05/2014 on Anonymisation Techniques, namely *singling-out*, *linkability*, and *inference*. We regard an attack as more effective when the attacker is able to make statements of the following sort with high confidence:

- *Singling-out:* "There is a single user with the following attributes."
- *Inference:* "All rows with attributes A, B, and C also have attribute D"
- *Linkability:* "The users with these attributes in the table protected by Diffix are also in a table not protected by Diffix."

A description of how these criteria are used can be found in the Diffix Birch paper

Trackers

A powerful class of attack, called trackers, was discovered in the late 1970s. This attack targets anonymization mechanisms where answers to queries that pertain to fewer than K individuals or more than all-but-K individuals are suppressed, but otherwise exact answers are given.

A tracker attack requires two things: 1. A pseudo-identifier: A column value or set of column values that uniquely identifies an individual. 2. A tracker: a column value that includes more than K individuals.

A pseudo-identifier can be an expression with multiple terms, like `(zip = 12345 AND bday = '1975-03-11' AND univ = 'MIT')`. The attacker does not need to know in advance that a given expression is a pseudo-identifier. This can be validated in the attack.

An example of a tracker would be `gender = 'M'` (because there are more than K males in the dataset). The victim of the attack doesn't have to be male. The tracker `gender = 'F'` would work just as well.

In the following example, the pseudo-identifier for the victim is `(zip = 12345 AND bday = '1975-03-11' AND univ = 'MIT')` and the tracker is `gender = 'M'`.

First, the attacker verifies that the pseudo-identifier indeed identifies a single individual with four queries. The first two are used to compute the total number of users in the database (here column `uid` contains a distinct value for each individual):

Query 1: compute *N1* as:

```
SELECT count(DISTINCT uid)
FROM database
WHERE gender = 'M'
```

Query 2: compute *N2* as:

```
SELECT count(DISTINCT uid)
FROM database
WHERE gender <> 'M'
```

The number of users N is $N = (N1 + N2)$

Query 3: Compute *N3*:

```
SELECT count(DISTINCT uid)
FROM database
WHERE gender = 'M' OR
      (zip = 12345 AND bday = '1975-03-11' AND univ = 'MIT')
```

Query 4: Compute *N4*:

```
SELECT count(DISTINCT uid)
FROM database
WHERE gender <> 'M' OR
      (zip = 12345 AND bday = '1975-03-11' AND univ = 'MIT')
```

Both counts *N3* and *N4* include all users with the pseudo-identifier. *N3* also includes all males, and *N4* includes everyone else. The sum *N3* + *N4* therefore counts all users with the pseudo-identifier twice, and counts everybody else once. As such, if $(N3 + N4) - N = 1$, then this validates for the attacker that there is only one user with the pseudo-identifier.

Knowing this, the attacker can learn virtually anything about the victim. For instance, the attacker can learn the victim's salary with four queries (here assuming that each user is represented only once in the database):

Query 1: compute *S1* as:

```
SELECT sum(salary)
FROM database
WHERE gender = 'M'
```

Query 2: compute *S2* as:

```
SELECT sum(salary)
FROM database
WHERE gender <> 'M'
```

The sum of all salaries S is $S = (S1 + S2)$.

Query 3: Compute *S3*:

```
SELECT sum(salary)
FROM database
WHERE gender = 'M' OR
      (zip = 12345 AND bday = '1975-03-11' AND univ = 'MIT')
```

Query 4: Compute *S4*:

```
SELECT sum(salary)
FROM database
WHERE gender <> 'M' OR
      (zip = 12345 AND bday = '1975-03-11' AND univ = 'MIT')
```

Again, *S3* and *S4* include the victim, while *S3* also includes males, and *S4* also includes all other users. The salary if the victim is counted twice in *S3 + S4*, and so the victim's salary is *(S3 + S4) - S*.

Diffix defends against tracker attacks by disallowing `OR` and it's De Morgan equivalent (`NOT (A and B)` is the same as `NOT A or NOT B`). It does allow a limited form of union logic with `IN`, but this is not enough to execute a tracker attack.

`OR` can be emulated with multiple queries and corresponding arithmetic, as in `A OR B` is equivalent to `A + B - A AND B`. This would not work with the cloak both because queries with only the pseudo-identifier would be suppressed because of a low count, as well as because of the noise.

Attribute value inspection attacks

Perhaps the most direct attack is to simply list column values. If the analyst can list a column value or set of column values that apply to a single user, then anonymity according to our goal would be violated.

A simple query for this attack would be:

```
SELECT ssn
FROM table
```

or for multiple columns,

```
SELECT birthdate, zip, gender
FROM table
```

If the first query listed any social security numbers, then the privacy of the users with those numbers would be violated. Likewise if the second query listed users with a unique combination of birthdate, zipcode, and gender, then those users privacy would be violated according to our goal.

This attack is prevented through the use of *low-count suppression*. Any column values that pertain to too few distinct users are suppressed by not being output.

Suppression signal attacks

An attacker may be able to learn about individual users from whether a given bucket was suppressed or not. For instance, if a constant threshold were used for the suppress decision (i.e. suppress when 2 or fewer distinct users), then if the attacker happened to know that there is either 2 or 3 users with a certain set of attribute values, then if the bucket is not suppressed then the attacker

knows that there must be three users, and potentially learns something about the third user.

To prevent this, Diffix uses a noisy threshold using sticky layered noise. Because the threshold itself can vary, in the previous example the attacker would be uncertain as to whether there were 2 or 3 users.

Noise signal attack

Diffix can report the amount of noise added to the aggregates `count()`, `sum()`, `avg()`, and `stddev()`. This is reported as the standard deviation of the noise. In Diffix Dogwood, the amount of noise is influenced by the `min` and `max` values. If a user has an extreme value (say 2x or 3x greater than the next highest value), then the presence or absence of that user in a bucket may have a significant effect on the amount of noise added.

This opens an attack whereby the attacker can determine which bucket the extreme user is in by noting which bucket reports the most noise (for instance, using `sum_noise()`).

For instance, if one user (the victimm) has a salary that far exceeds the second highest salary, then `gender` could be learned with the following query:

```
SELECT gender, sum(salary), sum_noise(salary)
FROM table
GROUP BY 1
```

Whichever gender bucket reports the highest `sum_noise(salary)` is the bucket that contains the victim.

To defend against this, the cloak does not precisely report the standard deviation of the noise, but rather an approximation based on a formulation that eliminates the effect of the `min` and `max` values.

Noise averaging attacks

Averaging attacks attempt to remove noise through multiple queries that somehow generate different noise samples (overcome the noise stickiness) and average away the noise.

Naive averaging

The simplest averaging attack repeats the same query. This doesn't work because of sticky noise generates the same noise value.

Different syntax but same semantics, with floating

Different noise samples are generated by queries that execute identical query semantics but with different syntax in the hope that the cloak does not recognize the same syntax and so produces a different noise sample.

A simple example would be two queries with the following `WHERE` clauses:

```
Q1: WHERE age = 10
Q2: WHERE age + 1 = 11
```

In principle, the cloak could interpret the math and determine that the above two expressions are identical. This can get extremely difficult, however, as the math gets more complex. Therefore for any expression that uses math, the cloak *floats* the column value by re-writing the SQL so that the column in question (here `age`) is brought up to the outer `SELECT` so that the actual selected values can be examined.

However, other examples exist:

```
Q1: WHERE left(column,5)
Q2: WHERE substring(column FROM 1 FOR 5)
```

Not so many different semantically identical conditions can be made with string manipulation as with math manipulation, but nevertheless the above conditions would be floated to ensure that the seeds are identical.

Different syntax but same semantics, without floating

The cloak doesn't always float. It does not float `col BETWEEN val1 AND val2` for instance. Rather, it deduces the seed material from SQL inspection. There are cases where same semantic but different syntax conditions can be generated from non-floated conditions. In a few cases, the cloak does not correctly account for the different syntax and uses different seeding material.

One such case is `BETWEEN` versus `bucket()`. For instance the range `100-200` can be generated with both of the following queries:

```
Q1: SELECT bucket(column BY 100) ....
Q2: WHERE column BETWEEN 100 and 200
```

The cloak, however, does not detect that these semantics are identical and generates different seeds as a result. Through this trick the analyst can get two noise samples for the layers associated with these conditions, and therefore reduce the overall noise somewhat.

In many other cases, however, the cloak does seed properly.

Text datatypes for instance are all converted to lower case for the purpose of seeding, so that the `lower()` and `upper()` functions can't be used to derive additional noise samples.

Different syntax and semantics, but same result

In Diffix Dogwood, an analyst can obtain multiple samples for `<>` text conditions using `substring()`. For instance, suppose that a text column contains two

values, 'ABCDE' and 'FGHIJ'. Each of the following conditions would generate a different seed even though the underlying answers are the same:

```
WHERE substring(col for 1) <> 'A'
WHERE substring(col for 2) <> 'AB'
WHERE substring(col for 3) <> 'ABC'
...
```

While this is certainly unfortunate, there are no known attacks that can successfully exploit this weakness.

Split averaging attack

This attack creates pairs of queries where the sum of the queries in each pair have the same underlying value (i.e. before noise), but where each pair uses semantically different conditions from all other pairs, thus producing different noise values that can be averaged. This can be done with `WHERE` clauses of the following sort:

```
Pair 1:
Q1: WHERE column = 'X'
Q2: WHERE column <> 'X'
Pair 2:
Q1: WHERE column = 'Y'
Q2: WHERE column <> 'Y'
```

From pair 1, the first query Q1 matches all rows where `column = 'X'`, and Q2 matches all other columns (not equal to 'X'). The sum of Q1 and Q1 therefore includes all rows (that match other conditions, not shown here). The same holds for the two queries from pair 2. However, all of these conditions are semantically different and so produce different noise samples.

The cloak defends against this through *layered noise*. Each condition generates its own noise layers, which are summed together to produce the final noise value. The layers from the above `WHERE` clauses would indeed be averaged away, but the noise from other conditions in the queries would not average out. As a simple example, the following two queries are from one pair where the attacker goal is to learn the exact number of women:

Query 1:

```
SELECT count(DISTINCT uid)
FROM table
WHERE age = 20 AND gender = 'F'
```

Query 2:

```
SELECT count(DISTINCT uid)
FROM table
WHERE age <> 20 AND gender = 'F'
```

While the noise layers for `age` would indeed average out, the static noise layer for `gender = 'F'` would be the same across all queries in the attack and would result in a noisy final count.

Linear program reconstruction

In this attack, the analyst generates queries where each query selects a different set of users, but where any given user is selected by a substantial number of queries. Repeatedly selecting among a pseudo-random subset of users would have this effect. The analyst then generates a set of equations and solves for the presence or absence of each user in that set of equations.

This attack was successfully run against Diffix Birch using the following WHERE clause:

```
WHERE floor(5 * pow((client_id * 2),0.5) + 0.5) = floor(5 * pow((client_id * 2), 0.5))
```

where different constants were used to effectly select different users.

The current defense is twofold: to force `clear` conditions (no math) on columns that have a substantial fraction of user-unique values, and to force `clear` conditions on all implicit ranges (for instance `round()`, `date_trunc()`).

JOINs with non-personal tables

The cloak does not anonymize non-personal tables. Therefore, analysts are allowed to see the complete tables. The cloak allows JOIN, but with strict limitations. Without these limitations, an analyst could potentially exploit knowledge of non-personal tables to generate multiple samples and average away noise.

A typical non-personal table might contain a numeric identifier and a text field. For instance, a `product_names` table might have columns `product_id` and `name`. This table is non-personal because it contains no user information. It may be linked to for instance a `purchases` table with the `product_id` as key.

Suppose an analyst wished to remove noise associated with the condition `age = 20`, and there are multiple non-personal tables of various sorts. The analyst could then build a set of queries with different `JOIN` conditions as follows:

```
JOIN (...) ON users.age = product_names.id - 28788 WHERE product_names.value = 'pears'
JOIN (...) ON users.age = error_codes.id - 232 WHERE error_codes.value = 'Bad Input'
JOIN (...) ON users.age = cc_types.id + 16 WHERE cc_types.value = 'VISA'
etc.
```

The ID of the non-personal table, after arithmetic, matches the target age (20). Because the table name is included in the seed material for the condition, each of the noise layers associated with the `ON` condition could be averaged away.

The fix is to limit `ON` conditions to simple `col1 = col2` expressions, and limit which columns can be keys through configuration. With these limitations,

any meaningful attack is so unlikely that no noise layers for `ON` conditions are necessary.

Difference attacks

In difference attacks, the attacker creates pairs of queries where the underlying true answers are either identical or differ by one user. The attacker then tries to determine which is the case.

A simple example would be the following pair of queries:

Query 1:

```
SELECT count(*)
FROM table
WHERE salary BETWEEN 100000 AND 110000 AND
      dept = 'CS' AND
      gender = 'M'
```

Query 2:

```
SELECT count(*)
FROM table
WHERE salary BETWEEN 100000 AND 110000 AND
      dept = 'CS'
```

in the case where there is only one woman in the CS department. The victim in this attack is the woman, who is excluded in the first query and included in the second. The unknown attribute (the thing the attacker wants to learn) is whether or not the victim's salary is in the range 100000 to 110000.

The simplest approach for the attacker is to deduce that the victim is in query 2 so long as `count(*)` is greater in query 2. Because of the noise, however, such an approach would often produce the wrong deduction.

The greater the difference between the two queries, however, the more likely the victim is indeed in that salary range. When the difference is large, the probability that the difference is due purely to the Gaussian distribution of noise is much less than the probability that the difference is due to both the noise distribution and the difference in the underlying true answer.

It is relatively rare, however, that the noise value is large enough to give the attacker a high-confidence deduction (like one in 10000 attacks).

Note that a simpler way to exclude the victim would be to add a negative condition that pertains to only that victim, for instance:

```
SELECT count(*)
FROM table
WHERE salary BETWEEN 100000 AND 110000 AND
      ssn <> '539-54-9355'
```

Note however that isolating negative conditions like this are disallowed when the value is not present in the shadow table (which is the case for rare values).

First derivative difference attack

There is a form of difference attack whereby the analyst generates a histogram of bucket pairs under the condition that the victim is not in the first query of each pair, and is in one and only one bucket of the second query. An example is the following:

Query 1:

```
SELECT bucket(salary BY 10000), count(*)
FROM table
WHERE dept = 'CS' AND
      gender = 'M'
```

Query 2:

```
SELECT bucket(salary BY 10000), count(*)
FROM table
WHERE dept = 'CS'
```

If the noise layers were only static (based purely on the conditions themselves), then the difference in the noise between each pair of buckets would be the same except for the one containing the victim. This difference in the difference would identify the victim's salary.

The uid-based noise layers, however, add additional noise that is different between every pair of buckets.

Difference attack with counting NULL

The effect of inserting safe math functions is that column values may be NULL. This gives an attacker the opportunity to force a single user to have the only NULL value in the column. The attacker can determine if the NULL value is present by comparing the outputs of `count(*)` and `count(col)`. The former counts the NULL rows, the latter does not.

For example, the following difference attack can be used to learn the value of the `cli_district_id` of the user that has a transaction amount of 24615.

```
SELECT cli_district_id, count(foo)
FROM (
    SELECT uid, cli_district_id, 1/(amount - 24615) AS foo
    FROM transactions ) t
GROUP BY 1
```

```
SELECT cli_district_id, count(*)
FROM (
    SELECT uid, cli_district_id, 1/(amount - 24615) AS foo
```

```
    FROM transactions ) t
GROUP BY 1
```

The cloak defends against this by adding an additional UID noise layer for `count(col)`.

Noise exploitation attacks

Through extreme user contribution The magnitude of the noise must be in some sense proportional to the amount that the most extreme users contribute to an answer. For instance, suppose that one knows that roughly the average salary in a database excluding the CEO. By querying for the average salary, the amount by which it differs from the known average reveals the CEO's salary. The noise needs to be large enough to mask this salary. This is called *proportional noise*.

However, with a difference attack, the amount of noise itself can reveal the presence or absence of the CEO in a query. If the CEO has a substantially higher salary than anyone else, and if the query that may include the CEO has a lot of noise, then one can conclude that the CEO is present in the query.

To prevent this, the cloak removes a small number of the most extreme values in a given query (based on a noisy threshold), and then bases the amount of noise on the average value of a small group of users with the next most extreme values (also based on a noisy threshold). This is called *flattening*.

Through chaff conditions In this difference attack, the attacker tries to exploit the uid-based noise by intentionally adding conditions that have no impact on the true underlying answer, but increase the cumulative amount of noise. The bucket pair that differs the most is most likely the one containing the user.

A simple way to do this would be to add conditions like `age <> 1000`, `age <> 1001` and so on. These are called chaff conditions. In pairs where the underlying set of users is the same, the uid-based noise values are the same for each query of the pair. In pairs where the underlying set of users is different, the uid-based noise value is different.

Alternatively, a range condition such as `age BETWEEN 0 and 1000` can also be a chaff condition. Likewise the implicit ranges `round()` and `trunc()` can be chaff conditions. This is because these functions can span the entire number range of a column with one bucket.

The chaff conditions can either all be added to the same query, or added one at a time to multiple pairs, and then summing the results across the first queries of each pair and separately the second queries.

The attack using range conditions is prevented by not using uid-based noise layers for range conditions.

The attack using negative conditions is prevented by disallowing negative conditions for rare values (those not found in the shadow table). Specifically, any value in the shadow table has at least 10 users associated with it. If such a value is used in a negative condition, then the selected users will cause a difference in multiple bucket pairs, and the attacker cannot be sure which are associated with the victim.

Range creep with averaging

One way to do a difference attack is to grow a range (`BETWEEN`) so that one additional user is included in the modified range. The noise layers associated with the range defeat a simple version of this attack that uses two queries. However, if an attacker could incrementally modify a range so that each change does not change the underlying answer, then the different noise values could be averaged. If the attacker averaged away the noise in this fashion for both the smaller and larger ranges, then the noise could no longer defend against the attack.

To defend against this, the cloak forces ranges to fall within preset range offsets and sizes. This makes it very unlikely that an attacker could get enough samples both excluding and including the victim.

Multiple isolating negands

If an analyst could make multiple different negative ANDed conditions (negands) isolating the same individual user, then they could be averaged out in the context of a difference attack. For instance, `account_number <> 12345`, `social_security_number <> 123-45-6789`, `login_name <> 'alex433`, `email_address <> 'alex433@gmail.com` are all values that can isolate a single user. If each were used as the single negand, then the negands could be averaged away.

This negands are rejected by virtue of not being represented in the shadow table.

Shadow table exploitation attack

For each column, the shadow table contains the up-to-200 column values assigned to the largest number of distinct users. Any value must have at least 10 distinct users to be included in the shadow table. Column values for negands (negative conditions such as `age <> 1000`) and `IN` expressions (`age NOT IN (1000,1001,1002)`) that are not in the shadow table are disallowed.

It is possible, however, for a negand from the shadow table to isolate a single user in a difference attack. This can happen when the other query conditions reduce the number of users sufficiently so that all other users with the same value are excluded from both queries.

For instance, suppose that around 30 users have a zip code of 12345. The negand `zip <> 12345` would then exclude those users. Suppose in addition that

some other attribute value, say `profession = 'stylist'`, applies to a relatively small fraction of the population, for instance 1 in 500 people, but nevertheless is in the shadow table. Finally, suppose that the attacker knows that the victim is a stylist and lives in zip code 12345. Chances are then high that two queries with the following two conditions:

```
Q1: profession = 'stylist'
Q1: profession = 'stylist' AND `zip <> 12345`
```

would differ by only one user, the victim. This is because it is statistically unlikely that another one of the 15 users in zip 12345 is a stylist.

A first derivative difference attack using only the above sets of conditions would fail because of the added noise. However, if the attacker could come up with 4 or 5 different pairs of conditions that have the above isolating characteristics, then the noise due to each of the negands could be averaged away.

However, it is quite rare that the necessary criteria exists even once for a given user, and virtually impossible that the criteria exists 4 or 5 times (and even less likely that the attacker would know the appropriate information about the victim). Therefore, while theortically possible, this attack is in practice essentially impossible.

SQL backdoor attacks

A backdoor attack is where the attacker avoids defense mechanisms by encoding conditions indirectly through math. For instance, the following query:

```
SELECT count(*)
FROM table
WHERE age = 30 OR age = 40
```

Can be encoded in the following query without using a `WHERE` clause:

```
SELECT count(*), age30or40 FROM (
    SELECT uid, (age30 + age40) % 2 as age30or40
    FROM (
        SELECT uid,
            floor((agegreater29 + ageless31) / 2) AS age30,
            floor((agegreater39 + ageless41) / 2) AS age40
        FROM (
            SELECT uid,
                ceil((age - 29) / 100) AS agegreater29,
                ceil(0 - (age - 31) / 100) AS ageless31,
                ceil((age - 39) / 100) AS agegreater39,
                ceil(0 - (age - 41) / 100) AS ageless41
            FROM table
        ) x
    ) y
```

```
) z
GROUP BY age30or40;
```

To prevent this, the cloak limits the amount of math, particularly non-continuous functions like `ceil` and `floor`, that can appear in a query.

Side Channel attacks

Error generation attacks

Divide by zero Say that the victim's birthdate is known to be '1957-14-12', and zipcode is 60036. The following query would trigger a divide-by-zero if the victim has a salary of 100000.

```
SELECT count(*) FROM
  (SELECT uid, z/(6003614.195712-z+d+y+m) FROM
     (SELECT uid, zipcode * 100 AS z,
             year(bday) / 10000 AS y,
             month(bday) / 1000000 AS m,
             day(bday) as d
      FROM user_info
      WHERE salary = 100000
     ) t1
  ) t2
```

Some databases throw an exception when divide-by-zero occurs. In these cases, the exception itself signals the salary of the victim. This attack is prevented by adding SQL that catches the exception or prevents it from occuring (depending on the backend database).

Overflow In some database, a numeric overflow throws an exception. This can be exploited, at least in Postgres, with for instance with the following attack.

```
SELECT count(*)
FROM accounts
WHERE lastname = 'Zamora' AND
      birthdate = '1996-11-17' AND
      2^(10000.01 * cli_district_id) = 123.12
```

The third condition in the `WHERE` clause causes an overflow if it is executed. Assuming that there is only a single user with the lastname 'Zamora', if Zamora does not have the indicated birthdate, then the third condition won't be executed and a suppressed output is given. If Zamora does have that birthdate, then the third condition throws an exception. The exception is transmitted to the analyst as an execution error.

A similar attack against `date`, `time`, and `datetime` column types is also possible.

The cloak defends against this by installing and executing "safe" math routines in the database. The safe routines capture exceptions and returns NULL rather

than throwing an exception. As a result there is no error signal transmitted to the analyst, and therefore the analyst doesn't know if an exception took place or not.

Unfortunately the safe math routines slow down query execution. To minimize this performance hit, the cloak also makes a conservative estimate as to whether or not a given math expression *might* result in an exception. If not, then the safe math routine is not executed.

Square root of a negative number It may be possible to execute an error generating attack by forcing the square root of a negative number. The cloak defends against this by checking if the operand of the square root function is negative.

NULL producing safe function attacks

IS NOT NULL The defense against Error generation attacks is to install or use a safe math routine that inserts a `NULL` value rather than throw and exception. This mechanism leads to attacks that exploits the insertion of a `NULL` value. ghi ghi4091

An example of this attack is the following two queries:

```
SELECT cli_district_id, sum(acct_district_id)
FROM transactions
WHERE 1/(amount - 24615) IS NOT NULL
GROUP BY 1
```

```
SELECT cli_district_id, sum(acct_district_id)
FROM transactions
WHERE 1/(amount - 24615.3333) IS NOT NULL
GROUP BY 1
```

The victim in the attack is a user that has a unique value for the `amount` column of 24615. No user has an `amount` value of 24615.3333. The goal of the attack is to learn the `cli_district_id` value of the user.

In the first query, the term `1/(amount - 24615)` has a divide-by-zero error for the victim, which gets converted to `NULL` and is therefore filtered by the `WHERE` condition. The `WHERE` clause of the second query doesn't filter any rows. At the same time, the noise layers for the two `WHERE` clauses are with high probability identical, because the seed would be based on the floated max and min `amount`, which are likely the same.

The victim's `cli_district_id` then is recorded as that where the sums from the first and second queries differ.

The defense against this attack is to force `IS NOT NULL` expressions to be `clear`.

NULL within aggregation The following query attacks the same victim as the previous example.

```
SELECT cli_district_id, count(amount), count(1/(amount - 24615))
FROM transactions
GROUP BY 1
```

The first `count()` counts all rows (assuming no `NULL` values in the `amount` column). Because the victim's row has a divide-by-zero which becomes `NULL`, the second `count()` counts all rows except that of the victim. There is no noise layer associated with the expression within the aggregation, and so the victim's `cli_district_id` is simply that where the two counts differ.

The defense is similar: to force expressions within aggregation functions to be clear.

Timing attacks

JOIN timing attack Because of optimizations in some database implementations, a query such as the following can be used in a timing attack:

```
SELECT count(*)
FROM (
  SELECT uid
  FROM accounts
  WHERE lastname = 'Zamora' AND birthdate = '1996-11-17' AND
        salary = 100000
) t1 JOIN (
  SELECT distinct uid
  FROM transactions
) t2 on t1.uid = t2.uid
```

The optimization here is that, if the left `JOIN` expression (`t1`) returns zero rows, then the right `JOIN` expression is either not executed or terminated before completion. Otherwise the right `JOIN` expression is executed to completion.

If the right `JOIN` expression takes a long time to compute, as this one does, then the analyst can determine whether or not the left `JOIN` expression has zero rows or not.

In the example above, the analyst may have know that there is only one user with name 'Zamora' and birthdate '1996-11-17'. If Zamora also has this salary, then the query execution time is shorter. If not, then the query execution time is longer. ghi3691