Technical Report MPI-SWS-2021-003

Paul Francis

Feb. 2, 2021

Customer Documentation for Aircloak's Diffix Dogwood

This technical report contains selected portions of the Aircloak customer documentation for Diffix Dogwood. The documentation is taken as-is from the online documentation provided by Aircloak.

Whereas the Diffix Dogwood description at MPI-SWS-2021-001 describes how the anonymization works, this document is useful for understanding the features and limitations of Diffix Dogwood, as well as the necessary steps in configuration.

Individual components

An Aircloak installation consists of a set of individual components. A subset of these are provided by Aircloak. The remaining ones are provided by the customer. Below follows a description of the individual components.

Components provided by Aircloak

Insights Air

Insights Air is the component that analysts and system administrators interact with directly. It provides a web interface for managing users, user privileges, and datasources, as well as for running queries. Additionally Insights Air provides an **HTTP API** for connecting external tools to an Aircloak installation as well as an endpoint for tools and plugins that support the **PostgreSQL message protocol**.

Insights Air never handles sensitive user data. It can therefore safely be deployed in a **DMZ** or made available to less privileged users.

Insights Cloak

Insights Cloak analyses and anonymizes sensitive data as requested by Insights Air. It operates on raw and sensitive data and should therefore run in an environment that is well protected. Insights Cloak does not require the ability to receive inbound connections. Upon booting it will establish a connection to the Insights Air instance it has been statically configured to trust. Multiple Insights Cloak instances can all be connected to the same Insights Air instance, and furthermore do not communicate with each other.

It is highly recommended that Insights Cloak is hosted within a restricted and well protected network as it has access to and operates on sensitive and restricted data.

Only anonymized and aggregated data is sent from Insights Cloak to Insights Air.

Insights Datasource Connector

Insights Datasource Connector knows how to provide the data required by Insights Cloak to run a query. It connects to the database server hosting the datasource being queried and transfers the required data to Insights Cloak. Insights Datasource Connector, like Insights Cloak, does not permanently store any sensitive data.

Insights Datasource Connector is deployed and configured as part of Insights Cloak. An Insights Datasource Connector instance therefore only serves a single Insights Cloak. An Insights Cloak instance on the other hand may make use of multiple distinct Insights Datasource Connectors in order to serve data from distinct datasources.

Because Insights Datasource Connector needs access to the database server hosting the data to be analysed, so does the Insights Cloak that it is a part of.

The Insights Datasource Connector has the ability to emulate database features beyond what is supported natively by the database server itself. Examples of this include the ability to work on encrypted fields and columns, converting data to types not natively supported, and performing table joins where no such support exists.

For more information about the supported datastores and what query features are emulated, please have a look at the datastore page.

Diffix Explorer

Diffix Explorer is an *optional* open source component backed by the Max Planck Institute for Software Systems, which integrates with Insights Air to provide actionable information to analysts without the need to manually write queries. Insights Air provides a built-in UI integration to manage and display these results, but Diffix Explorer can also be used through it's API to derive useful information for other interfaces.

Components provided by the customer

PostgreSQL database

Insights Air requires access to a PostgreSQL database. This database is used to store analyst accounts, system settings, as well as audit logs. No sensitive user data nor access credentials to the datasources being queried are stored in this database.

Datasource

The data being queried is made available through a database system of some kind. In nearly all cases this system already exists and can directly be used by the Aircloak Insights platform.

A list of supported datasources can be found here.

Logging infrastructure

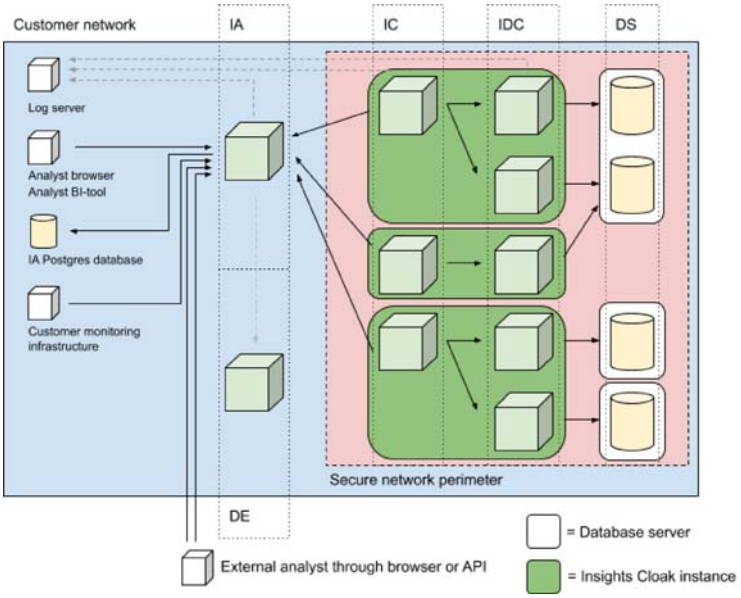
Insights Air, and Insights Cloak (and through Insights Cloak also Insights Datasource Connector) produce logs that can be used to determine if the system is behaving as expected. These logs are automatically collected by the infrastructure on which the individual components run, but can optionally be forwarded to a centralized log storage and processing facility operated by the customer.

Monitoring

Insights Air provides a monitoring API endpoint. This API endpoint can be tied into existing monitoring infrastructure, to improve the visibility into the workings of the Aircloak Insights platform.

How the components interact

The following diagram shows the possible ways in which the components interact. For common configurations, see the deployment guide.



In the diagram above the arrows point to the component being connected to. For example it shows that Insights Cloak connects to Insights Air, rather than the other way around.

Of special interest is that:

- An Insights Air instance can serve multiple Insights Cloak instances
- An Insights Cloak instance can connect to multiple datasources through multiple Insights Datasource Connectors. These datasources might be hosted in a single or in distinct database servers.
- The Insights Cloak and Insights Datasource Connectors should be hosted in a protected environment, whereas it is generally safe to allow analysts outside the customer organisation to access Insights Air

Query Language

To write queries you use SQL. Aircloak supports a subset of standard SQL, implemented in a way that prevents leakage of sensitive data.

Exploring the database

You can discover database tables and their structure using the `SHOW` statement. To list the tables in the database, you can use the `SHOW TABLES` statement. If you want to see the columns of a particular table, you can invoke `SHOW COLUMNS FROM table_name`.

Querying the database

The `SELECT` statement can be used to obtain anonymized data from tables. See Understanding query results for an explanation of the effects of anonymization on the results.

The syntax conforms to the standard SQL syntax (with some exceptions), but only a subset of features is supported. The general shape of the query looks like:

filename

Describing the query plan

To inspect a query without running it, you can prefix the uppermost `SELECT` statement with `EXPLAIN`. The `EXPLAIN` query will return an overview with information about the query and its subqueries.

The following example shows the result of describing an anonymizing restricted query with an inner non-anonymizing restricted subquery:

```sql
EXPLAIN SELECT age, count(*) as individuals
FROM (
  SELECT uid, t1.age
  FROM table1 t1
  INNER JOIN table2 t2 ON t1.uid = t2.uid
) t
GROUP BY age
```

```
query (anonymized, statistics, 2 noise layers)
  --> regular_stats (Aircloak generated, restricted)
    --> uid_grouping (Aircloak generated, restricted)
      --> t (restricted)
        --> t1 (personal table)
        --> t2 (personal table)
```

Considerations

- The `*` argument can only be provided to the `COUNT` and `COUNT_NOISE` aggregators and it specifies counting rows instead of otherwise counting only non-`NULL` values. `NULL` values are ignored by all other aggregators.
- The operator `OR` is not supported.
- The operator `NOT` can only be used in the cases mentioned above (`IS NOT NULL` , `NOT IN` , `NOT LIKE` , `NOT ILIKE` , and `NOT boolean_column_expression`).
- You can restrict the range of returned rows by a query using the `LIMIT` and/or `OFFSET` clauses, but you need to provide the `ORDER BY` clause to ensure a stable order for the rows.
- Conditions in the `HAVING` clause must not refer to non-aggregated fields.
- Aliases can be used in the `WHERE` , `GROUP BY` , `ORDER BY` and `HAVING` clauses, as long as the alias doesn't conflict with a column name in one of the selected tables.
- If an integer is specified in the `GROUP BY` or `ORDER BY` clause, it represents a 1-based position in the select list. The corresponding expression from the select list is used as the grouping or ordering expression.
- Values of type `datetime with timezone` are not supported. The timezone information will be dropped and the value will be exposed as a simple `datetime` in the UTC format.
- The order of rows in subqueries is not preserved in the outer query. Add an `ORDER BY` clause in the outer query if you want a specific order.
- When `NULL` handling is not specified in an `ORDER BY` in a subquery (either `NULLS FIRST` or `NULLS LAST`) the default handling for the underlying datasource will be used. For PostgreSQL that means that `NULL` values will be treated as larger than all other values. For MySQL and SQL Server they will be treated as smaller than all other values. The top-level query always defaults to treating `NULL` values as larger than other values.
- Using a `column_expression` in place of a `filter_expression` will implicitly compare the value of that `column_expression` to `TRUE` . In other words: `WHERE active` is equivalent to `WHERE active = TRUE` .

Query and subquery types

Aircloak Insights supports both queries over `personal` data and queries over `non-personal` data. In this context `personal` data is data about a single person (or entity, depending on what it is you want to protect through anonymization), and `non-personal` data facts from a fact or lookup table (not relating to any one individual) or the result of an anonymizing subquery over personal data.

Queries that process `personal` data are subject to various **restrictions**, and are called *restricted queries*. Restricted queries can be arbitrarily nested. The top-most restricted query anonymizes the data. The anonymization produces a result that is about groups of users rather than individuals, and filters out values that could identify an individual. Such a top-most restricted query is called an *anonymizing query*.

An anonymizing query can itself be a subquery to another query. In such a case the data processed by the other query is already anonymous and hence `non-personal` . Such a query is called a *standard query*. A standard query can be used to further process an anonymized result set or, for example, to combine anonymized data with data from a `non-personal` table such as a fact or lookup table. Standard queries have the usual SQL validations applied to

them (such as type checking), but the restrictions that are enforced for queries processing personal data do not apply. Standard queries can only refer to `non-personal` tables or to other standard or anonymizing subqueries.

Distinguishing between query types

Being able to tell the query types apart helps you make sure you get the results you expect and want. It is not uncommon that an analyst new to Aircloak Insights ends up writing a subquery producing aggregates over multiple users, which becomes an anonymizing subquery, where they needed to write a subquery producing per-user aggregates, which becomes a non-anonymizing restricted subquery. This mistake can cause most of the data to get anonymized away, leading to results vastly different from what was expected. It can be hard to debug and understand the cause of a situation like this if you do not know of the different query types and how to distinguish between them.

You can use the **EXPLAIN statement** to identify the type of a query and its subqueries.

Non-anonymizing restricted subqueries

You can tell that a subquery produces a per-user aggregate (is a *non-anonymizing restricted subquery*) if *both of* the following are true:

- it processes data from one or more personal tables or other non-anonymizing restricted subqueries
- it explicitly selects or groups by the column that was specified as the user id

Note that the top-most query can never be a non-anonymizing restricted subquery. The reason for this is that all Aircloak queries have to produce an anonymized result. This is incompatible with the results of a non-anonymized restricted subquery which is always personal data.

Some examples

This query is a non-anonymizing restricted subquery as it is a subquery and explicitly selects the user id column from a table containing personal data:

```sql
... (
  SELECT uid, age
  FROM personal_table
) subquery
...
```

This query is a non-anonymizing restricted subquery as it is a subquery and explicitly creates a per-user aggregate (through selecting and grouping by the user id column) from a table containing personal data:

```sql
... (
  SELECT uid, count(*) as numTransactions
  FROM transactions
  GROUP BY uid
) subquery
...
```

This query is a non-anonymizing restricted subquery as it is a subquery and explicitly creates per-user aggregates (through selecting and grouping by the user id column) from a combination of tables containing personal data:

```sql
... (
  SELECT t.uid, count(*) as numTransactions, count(distinct a.id)
as numAccounts
  FROM transactions t INNER JOIN accounts a ON t.account_id =
a.id
  GROUP BY t.uid
) subquery
...
```

Anonymizing restricted queries

A query is an anonymizing query if it operates on personal data and *either of* the following is true:

- it is the top-most query (Aircloak will never return a non-anonymized result)
- it aggregates over multiple users (i.e. does not explicitly group by the user id column)

Some examples

This query is anonymizing as it aggregates the personal data of multiple users into a single aggregate. That the aggregate is over multiple users can be deduced from the query not grouping by the user id column. Grouping by the user id column would produce per-user aggregates.

Even if the query hadn't been an aggregate over the personal data of multiple users, it would still have been an anonymizing query. This is the case because it is the top-most query and processes personal data. Had it not been anonymizing, it would have leaked personal information.

```sql
SELECT count(*)
FROM personal_table
```

The following subquery is anonymizing, despite grouping by a column that, in this particular example, likely is mostly unique to a user. The intent might have been for the query to be a non-anonymizing restricted query producing a per-user aggregate. It isn't marked as such by Aircloak Insights as it does not select and group by the column specified as the user id column. This query is likely to produce no (or very little) results after anonymization as there will only be few cases where multiple distinct individuals share the same phone number.

```sql
... (
  SELECT phone_number, count(*) as numTransactions
  FROM transactions
  GROUP BY phone_number
) per_phone_transactions
```

The following subquery is anonymizing as it produces an aggregate over the data of multiple users. The resulting data can be further processed by a standard query.

```sql
... (
  SELECT city, count(*)
  FROM personal_table
  GROUP BY city
) city_distribution
```

Somewhat counterintuitively the following query is rejected despite being perfectly valid SQL. The reason is that it is a personal query due to creating a per-user aggregate. We can tell that this is the case by observing that the query selects (and groups by) the user id column. At the same time the query is the top-most query and as such Aircloak is forced to anonymize its output. Anonymizing the results of a query where each output row uniquely belongs to a single user (the uid column is unique to a user) cannot be done without having to filter away all the data. Aircloak therefore rejects the query.

```sql
SELECT uid, count(*)
FROM personal_table
GROUP BY uid
```

Standard queries

A standard query is a query that executes over anonymized and/or non-personal data. In fact any query which is not either an anonymizing query or a non-anonymizing restricted query is a standard query.

The following query combines all query types into a single composite query showing how they might relate:

```sql
-- Standard query (only processes anonymized (and hence non-
personal) data)
SELECT
  min(age), max(age),
  count(age), sum(individuals) as num_users
FROM (
  -- An anonymizing (and restricted) query as it produces an
aggregate
  -- that spans the data of multiple persons
  SELECT age, count(*) as individuals FROM (
    -- Restricted query as processes per-user data and explicitly
    -- selects the user id column (here named uid)
    SELECT uid, t1.age
    FROM table1 t1 INNER JOIN table2 t2
      ON t1.uid = t2.uid
  ) t
  GROUP BY age
) b
```

Restrictions

The Aircloak system imposes restrictions on the query language used in anonymizing queries that go beyond those imposed by the ANSI SQL standard. The restrictions, outlined below, do not apply to standard queries. For an overview over the difference between anonymizing and standard queries, consult the Query and subquery types section.

JOIN restrictions

To ensure that data can be reliably anonymized, some limitations exist in the `JOIN` part of the query.

Comparison operators `NOT LIKE`, `NOT ILIKE`, and `<>` are not allowed in join conditions (i.e. the `ON ...` part of a `JOIN` expression).

When analysing data across multiple tables, it is required that the data that is joined is all about the same individual. This can either be achieved by adding a `WHERE`-clause, or in the case of `INNER JOIN`'s and `OUTER JOIN`'s through a corresponding restriction in the `ON`-clause.

For example, assuming tables `t1` and `t2` both have a `user_id` columns called `uid`, you would write joins as follows:

- `SELECT c1, c2 FROM t1, t2 WHERE t1.uid = t2.uid`
- `SELECT c1, c2 FROM t1 CROSS JOIN t2 WHERE t1.uid = t2.uid`
- `SELECT c1, c2 FROM t1 INNER JOIN t2 ON t1.uid = t2.uid`

Note:

- `OUTER` is automatically implied when you use `LEFT`, `RIGHT` joins. Writing `LEFT OUTER JOIN` is therefore equivalent to writing `LEFT JOIN`
- `INNER` is automatically implied when you use `JOIN` without any other qualifiers. Writing `t1 JOIN t2` is therefore the same as writing `t1 INNER JOIN t2`

Subquery restrictions

A subquery expression with an aggregate must always select a `user_id` column as well. For example, assuming there exists a table `t1` with a `user_id` column called `uid`:

- **Valid**: `SELECT name FROM (SELECT name FROM t1) sq`
- **Valid**: `SELECT name FROM (SELECT uid, count(*) FROM t1 GROUP BY uid) sq`
- **Invalid**: `SELECT name FROM (SELECT count(*) FROM t1) sq`

When using `LIMIT` and `OFFSET` in a subquery:

- `LIMIT` is required if `OFFSET` is specified
- `LIMIT` will be adjusted to the closest number in the sequence `[10, 20, 50, 100, 200, 500, 1000, ...]` (i.e. `10e^n`, `20e^n`, `50e^n` for any natural number `n` larger than 0). For example: 1 or 14 become 10, etc

- `OFFSET` will automatically be adjusted to the nearest multiple of `LIMIT`. For example an `OFFSET` of 240 will be adjusted to 200 given a `LIMIT` of 100

Top-level HAVING clause

Any conditions specified in the `HAVING` clause of the top-level query (*not* a subquery) are "safe" in the sense that they will only ever be applied to data that has already been aggregated and anonymized. The clause will merely affect which of the anonymized data to display, not how that data is obtained. Because of this, many of the restrictions described in the following sections don't apply to the top-level `HAVING` clause.

CASE statements

`CASE` statements over personal data have multiple restrictions:

- They are only allowed in the `SELECT` or `GROUP BY` clauses of anonymizing queries;
- They can not be post-processed in any way, other than aggregation;
- The `WHEN` clauses can only consist of a single equality condition between a clear expression and a constant. The constant has to be from the list of frequent values in that column, unless the system administrator explicitly allows usage of any value. Check the **Insights Cloak configuration** section for information on how to enable it.
- The `THEN` / `ELSE` clauses can only return constant values; furthermore, when aggregated, they can only return the values 0, 1 or NULL.

A few examples:

```sql
-- Correct - conditional selection and grouping:
SELECT
  CASE
    WHEN column = 'aaa' THEN 1
    WHEN column = 'bbb' THEN 2
    ELSE NULL
  END,
  COUNT(*)
FROM table
GROUP BY 1

-- Correct - conditional aggregation:
SELECT SUM(CASE WHEN column = 'aaa' THEN 1 END) FROM table

-- Incorrect - multiple conditions are used in the same `WHEN`
clause:
SELECT CASE WHEN column = 'aaa' AND column = 'bbb' THEN TRUE END
FROM table

-- Incorrect - an unsupported condition is used in the `WHEN`
clause:
SELECT CASE WHEN column <> 'aaa' THEN TRUE END FROM table

-- Incorrect - the `THEN` clause doesn't return a constant value:
SELECT CASE WHEN column = 'aaa' THEN other_column END FROM table

-- Incorrect - the `ELSE` clause returns an unsupported constant
during aggregation:
SELECT AVG(CASE WHEN column = 'aaa' THEN 0 ELSE 1000 END) FROM
table

-- Incorrect - `CASE` statement is not used in the `SELECT` or
`GROUP BY` clauses:
SELECT COUNT(*) FROM table WHERE CASE WHEN column = 3 THEN TRUE
ELSE FALSE END
```

Math and function application restrictions

The usage of some functions is restricted. The scenarios when the restrictions come into effect are when a database column is transformed by more than 5 such functions and the expression on which the functions operate also contains a constant.

An expression containing two or more mathematical operators is considered to be the equivalent of a constant. The reason for this is that one can easily construct constants from pure database columns. For example `pow(colA, colB - colB)` equals the number 1.

The rules apply to the following functions:

- `abs` , `bucket` , `ceil` , `floor` , `length` , `round` , `trunc` , and `cast` 's.
- `+` , `-` , `*` , `/` , `^` , `%` , `pow` , `sqrt`
- `year` , `quarter` , `month` , `day` , `hour` , `minute` , `second` , `weekday` , `date_trunc`
- `btrim` , `ltrim` , `rtrim` , `left` , `right` , `substring`

Below is an example of the restrictions in action:

```sql
-- The following query contains more than 5 restricted functions
as well as a constant and
-- is therefore rejected.

SELECT
  -- This expression contains a total of 7 restricted functions:
  -- - 3 from value1
  -- - 3 from value2
  -- - 1 from the addition of value1 and value2
  value1 + value2
FROM (
  SELECT
    uid,
    -- contains 3 restricted functions, namely:
    -- - division with a constant
    -- - abs on an expression containing a constant
    -- - + where one of the arguments is an expression containing
a constant
    abs(age / 2) + height as value1
  FROM table1
) a INNER JOIN (
  SELECT
    uid,
    -- contains 3 restricted functions, namely:
    -- - addition with a constant
    -- - division with a constant
    -- - multiplication where one of the arguments is an
expression containing a constant
    (birth_year + 1) / 11 * height as value2
  FROM table
) b ON a.uid = b.uid
```

Below is an example of a query being rejected because multiple math operators have been interpreted as being a constant:

```sql
SELECT
  -- we have a total of 6 functions operating on an expression
containing a potential constant,
  -- as a result the query is rejected.
  floor(abs(sqrt(ceil(floor(sqrt(
    -- Aircloak considers two or more math operations to
potentially be a constant
    (age / age) / age
  , 2)), 2))))
FROM table
```

Functions that can cause database exceptions when a database column contains a certain value are prohibited. These functions include division and sqrt when the divisor and the parameter respectively are expressions containing a database column as well as a constant value.

Below is an example of the restrictions in action:

```sql
-- The following query is illegal as the divisor contains a
constant, in this case the number 1
SELECT age / (age + 1) FROM table
```

Constant values

In order to prevent overflow errors, the following restrictions on constant values are in place:

- Numeric values are limited to the range `[-10^18, 10^18]`.
- Date and datetime years are limited to the range `[1900, 9999]`.
- Intervals are limited to `100` years.

Clear expressions

A clear expression is a simple expression that:

- references exactly one database column,
- uses at most one `CAST`,
- only uses the following allowed functions:
  - string functions: `lower`, `upper`, `substring`, `trim`, `ltrim`, `rtrim`, `btrim`, `hex`, `left`, `right`;
  - date/time functions: `year`, `quarter`, `month`, `weekday`, `day`, `hour`, `minute`, `second`, `date_trunc`;
  - numerical functions: `trunc`, `floor`, `ceil`, `round`, `bucket`;
  - any aggregator (`MIN`, `MAX`, `COUNT`, `SUM`, `AVG`, `STDDEV`, `VARIANCE`).

Such expressions are considered to be safe in general and are exempt from many of the following restrictions.

Aggregated expressions

All aggregated expressions have to be clear.

```sql
-- Correct - aggregated expression is clear:
SELECT SUM(round(column)) FROM table

-- Incorrect - aggregated expression is not clear:
SELECT SUM(1 / column) FROM table
```

`IS [NOT] NULL` conditions

The subject of an `IS [NOT] NULL` condition has to be a clear expression.

```sql
-- Correct - subject is a clear expression:
SELECT COUNT(*) FROM table WHERE column IS NOT NULL

-- Incorrect - subject is not a clear expression:
SELECT COUNT(*) FROM table WHERE 1 / column IS NULL
```

Constant ranges

Whenever a comparison (`>`, `>=`, `<`, or `<=`) with a constant is used in a `WHERE`-, `JOIN`- or `HAVING`-clause, that clause needs to contain two comparisons. These should form a constant range on a single clear expression. That is, one `>=` comparison and one `<` comparison, limiting the expression from bottom and top.

The following special cases are excluded from this restriction:

- comparisons with clear expressions on both sides;
- date comparisons between a clear expression and the current date.
- date comparisons between a constant, month-aligned date and a range of clear expressions.

```sql
-- Correct - a constant range is used:
SELECT COUNT(*) FROM table WHERE column > 10 AND column < 20

-- Correct - comparison between clear expressions:
SELECT COUNT(*) FROM table WHERE column1 > column2
SELECT COUNT(*) FROM table WHERE column1 < round(column2)

-- Incorrect - only one side of the constant range provided:
SELECT COUNT(*) FROM table WHERE column > 10

-- Incorrect - the lower end of the constant range is bigger than
the upper end:
SELECT COUNT(*) FROM table WHERE column > 10 AND column < 0

-- Incorrect - the comparisons are over different expressions:
SELECT COUNT(*) FROM table WHERE column + 1 > 10 AND column - 1 <
20

-- Incorrect - multiple columns are referenced on one side of the
comparison:
SELECT COUNT(*) FROM table WHERE column1 - column1 < column2

-- Correct - comparison between a clear expression and the
current date:
SELECT COUNT(*) FROM table WHERE column <= current_date()

-- Correct - comparison between a month-aligned date and a range
of clear expressions:
SELECT COUNT(*) FROM table WHERE date '2020-01-01' BETWEEN
column1 AND column2
```

Note that a condition using the `BETWEEN` operator automatically forms a constant range:

```sql
-- These two queries are equivalent:
SELECT COUNT(*) FROM table WHERE column BETWEEN 10 AND 20
SELECT COUNT(*) FROM table WHERE column >= 10 AND column < 20
```

Constant range alignment

The system will adjust constant ranges provided in queries. The adjustment will "snap" the range to a fixed, predefined grid. It will always make sure that the specified range is included in the adjusted range. The range will also be modified to be closed on the left (`>=`) and open on the right (`<`).

If any such modifications take place an appropriate notice will be displayed in the web interface. When using the API the notice will be included under the `info` key of the result. When using the PostgreSQL interface, it will be sent across the wire as a `notice` message.

The grid sizes available depend on the type of the column that is being limited by the range:

- For numerical columns the grid sizes are `[..., 0.1, 0.2, 0.5, 1, 2, 5, 10, 20, 50, ...]`
- For date/time columns they are:
  - `[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.

The adjusted range will have the smallest size from the ones listed that can contain the full range provided in the query. Furthermore the starting point of the range will be changed so that it falls on a multiple of the adjusted range's size from a zero point. That zero point is the number 0 for numbers, midnight for times, and `1900-01-01 00:00:00` for dates and datetimes.

To better fit the range provided in the query the range might also be shifted by half its size, however this will not happen in the following cases:

- A range of 1 day on a date type - the underlying data type cannot represent such a shift
- A range of 1 second on a time or datetime type - the underlying data type cannot represent such a shift
- A range of 1 month - months have an irregular number of days and no clear "half-point"

For best results design your queries so that they take this adjustment into account and mostly use ranges that are already adjusted.

```sql
SELECT COUNT(*) FROM table WHERE column > 10 AND column < 20
-- Adjusted to 10 <= column < 20

SELECT COUNT(*) FROM table WHERE column >= 10 AND column < 19
-- Adjusted to 10 <= column < 20

SELECT COUNT(*) FROM table WHERE column >= 9 AND column < 19
-- Adjusted to 0 <= column < 20

SELECT COUNT(*) FROM table WHERE column >= 16 AND column < 24
-- Adjusted to 15 <= column < 25

SELECT COUNT(*) FROM table WHERE date >= '2016-01-01' AND date <
'2016-01-29'
-- Adjusted to a full month - 2016-01-01 <= date < 2016-02-01

SELECT COUNT(*) FROM table WHERE datetime >= '2016-01-01
12:27:00' AND date < '2016-01-01 12:31:00'
-- Adjusted to a grid size of 5 minutes - 2016-01-01 12:22:30 <=
datetime < 2016-01-01 12:37:30
-- The 5 minute intervals can start on a full five-minute mark or
a 2 minutes 30 seconds mark

SELECT COUNT(*) FROM table WHERE date >= '2017-01-10' AND date <
'2017-01-20'
-- Adjusted to 20 days - '2017-01-07' <= date < '2017-01-27'
-- The day-sized intervals can only start a multiple of their
size from 1900-01-01

SELECT COUNT(*) FROM table WHERE date >= '2017-01-07' AND date <
'2017-01-17'
-- Not adjusted -- see previous example
```

Implicit ranges

Some functions can be used to almost the same effect as a pair of inequalities. For example the following two queries are roughly equivalent:

```sql
SELECT COUNT(*) FROM table WHERE round(number) = 10
SELECT COUNT(*) FROM table WHERE number >= 9.5 AND number < 10.5
```

Because of this, usage of such functions must be restricted in a similar way to inequalities and the `BETWEEN` operator. The restrictions disallow the usage of most functions or mathematical operations before or after applying an implicit range function, if the expression is not clear. The operations that can be applied are a single `CAST`, any aggregator (`MIN`, `MAX`, `COUNT`, `SUM`, `AVG`, `STDDEV`, `VARIANCE`), and a sub-month date extraction function (`day`, `weekday`, `hour`, `minute`, `second`, `extract(day/weekday/hour/minute/second)`). The restrictions apply when an implicit range function is used in a `WHERE` or `JOIN` clause, selected in the top-level `SELECT` clause or used in a non-top-level `HAVING` clause - see Top-level HAVING clause.

The following functions are treated as implicit range functions: `round`, `trunc`, `date_trunc`, and sub-month date extraction functions (`day`, `weekday`, `hour`, `minute`, `second`, `extract(day/weekday/hour/minute/second)`).

```sql
-- Correct - no other function used
SELECT COUNT(*) FROM table WHERE round(number) = 10

-- Correct - an aggregate is used
SELECT COUNT(*) FROM table GROUP BY category WHERE
round(max(number)) = 10

-- Incorrect - another operation (/) is applied in the same
expression as round
SELECT COUNT(*) FROM table WHERE round(number / 2) = 10

-- Correct - used in the top-level HAVING, so restrictions don't
apply
SELECT COUNT(*) FROM table GROUP BY category HAVING
round(max(number) / 2) = 10

-- Incorrect - used in a non-top-level HAVING
SELECT COUNT(*) FROM (SELECT uid FROM table GROUP BY category
HAVING round(max(number) / 2) = 10) x

-- Incorrect - another operation is used in top-level SELECT
SELECT round(abs(number)) FROM table

-- Correct - math on month-aligned expressions is allowed
SELECT COUNT(*) FROM table WHERE year(birthday) * 12 +
month(birthday) = 2000 * 12 + 3

-- Incorrect - math on sub-month-aligned expressions is rejected
SELECT COUNT(*) FROM table WHERE month(birthday) * 30 +
day(birthday) = 100
```

Text operations

Certain operations on textual data can be used to almost the same effect as a pair of inequalities. For example the following two queries are roughly equivalent:

```sql
SELECT COUNT(*) FROM table WHERE number BETWEEN 10 AND 20

SELECT COUNT(*) FROM table WHERE
  LEFT(CAST(number AS text), 1) = '1' AND length(CAST(number AS
text)) = 2
```

Because of this, the usage of operations on textual data has to be restricted to prevent circumvention of measures that would normally limit what can be done with range conditions. The restrictions on expressions containing text manipulation functions are the same as ones described for implicit ranges. In addition, the result of text manipulation can only be compared to a clear expression.

The following functions are treated as text manipulation functions: `left`, `right`, `rtrim`, `ltrim`, `trim`, and `substring`.

```sql
-- Correct
SELECT COUNT(*) FROM table WHERE LEFT(name, 1) = 'A'

-- Incorrect - the results of a text operation are compared to a
complex expression
SELECT COUNT(*) FROM table WHERE LEFT(name, 1) = RIGHT(name ||
'a', 1)
```

Furthermore, the aggregators `min` and `max` cannot be used on data of type `text` in anonymizing queries and subqueries (see Query and subquery types for more about this distinction). This is due to the mode of operation of these aggregators and the fact that they require estimating the spread in subsets of the data. See Aggregates for more on this topic.

IN, NOT IN, NOT LIKE, and <>

Any conditions using `IN`, `NOT LIKE`, `NOT ILIKE`, or `<>` are subject to additional restrictions. Note that `NOT IN` is treated just as `<>`, because there is always an equivalent query using `<>` for every `NOT IN` query:

```sql
-- These queries are equivalent
SELECT COUNT(*) FROM table WHERE number NOT IN (1, 2, 3)
SELECT COUNT(*) FROM table WHERE number <> 1 AND number <> 2 AND
number <> 3
```

Allowed expressions

Conditions using `IN` or `<>` have to have a clear expression on the left-hand side. Only a single `COUNT`, `MIN` or `MAX` aggregator is allowed in such conditions.

All items on the right-hand side of the `IN` operator must be constants from the list of frequent values in that column, unless the system administrator explicitly allows usage of any value. Check the Insights Cloak configuration section for information on how to enable it.

The right-hand side of a `<>` condition has to be a clear expression or a constant from the list of frequent values in that column.

Conditions using `NOT LIKE` or `NOT ILIKE` cannot contain any functions except for aggregators. A single `CAST` is allowed.

The top-level `HAVING` clause is exempt from all these restrictions - see Top-level HAVING clause.

```sql
-- Correct - assuming 'alice' is a frequent value
SELECT COUNT(*) FROM table WHERE lower(name) <> 'alice'
SELECT COUNT(*) FROM table WHERE name IN ('alice', 'bob')

-- Incorrect - a disallowed operation was used
SELECT COUNT(*) FROM table WHERE length(name) <> 2
SELECT COUNT(*) FROM table WHERE length(name) IN (1, 2, 3)

-- Correct - top-level HAVING is exempt from restrictions
SELECT COUNT(*) FROM table GROUP BY name HAVING length(name) <> 2

-- Correct - comparing two clear expressions
SELECT COUNT(*) FROM table WHERE name <> surname
SELECT COUNT(*) FROM table WHERE round(column1) <> round(column2)

-- Incorrect - multiple columns are referenced on one side of the
inequality:
SELECT COUNT(*) FROM table WHERE column1 - column1 <> column2
```

Negative conditions over rare values

By default, negative conditions (`NOT IN` , `NOT LIKE` , `NOT ILIKE` , and `<>`) over rare personal values are forbidden. Conditions that match values appearing frequently in a given column are excluded from this limitation.

This limitation can be relaxed, on a per data source basis, by increasing the value of the `max_rare_negative_conditions` configuration setting. Check the Insights Cloak configuration section for information on how to modify it. Note that a `NOT IN` condition will be counted multiple times - once for each rare element on the right-hand side.

The examples below assume that the Insights Cloak is configured to allow one negative condition at the most, and that the names `Alice` and `Bob` appear frequently in the column `name` , while all other values appear only rarely.

```sql
-- Allowed - only one negative condition matches a rare value
SELECT COUNT(*) FROM table
WHERE name <> 'Alice' AND name <> 'Bob' AND name <> 'Charles'

-- Allowed - there is only one negative conditions on a rare
value
SELECT COUNT(*) FROM table
WHERE name <> 'Charles'

-- Disallowed - there are three negative conditions matching rare
values
SELECT COUNT(*) FROM table
WHERE name <> 'Charles' AND name <> 'Damien' AND name <> 'Ecbert'

-- Disallowed - equivalent to the previous query
SELECT COUNT(*) FROM table
WHERE name NOT IN ('Charles', 'Damien', 'Ecbert')

-- Allowed - all conditions match frequent values
SELECT COUNT(*) FROM table
WHERE name NOT LIKE 'A%' AND name NOT LIKE 'B%' AND upper(name)
<> 'BOB'
```

Isolating columns

Some columns in the underlying data source might identify users despite not being marked as a user id. For example a table might contain a `user_id` column and an `email` column. The emails are in all likelihood unique per user, and so can identify a user just as well as the `user_id` column. We call these columns "isolating" and apply some additional restrictions to expressions including them. Note that the `user_id` column is always isolating.

Only clear expressions are allowed on these columns. All other functions and mathematical operations are forbidden.

Furthermore, conditions using the `LIKE` operator are limited to simple patterns of the form `%foo`, `foo%`, or `%foo%`.

```sql
-- These examples assume that the 'email' and 'social_security'
columns are isolating

-- Correct
SELECT COUNT(*) FROM table WHERE trim(email) =
'alice@example.com'

-- Correct
SELECT COUNT(*) FROM table WHERE email <> 'alice@example.com'

-- Incorrect - a function from outside the allowed list is used
SELECT COUNT(*) FROM table WHERE length(email) = 20

-- Incorrect - a mathematical operation is used
SELECT COUNT(*) FROM table WHERE social_security / 10 = 10000000

-- Correct
SELECT COUNT(*) FROM table WHERE BUCKET(social_security BY 10) =
100000000

-- Incorrect - a disallowed LIKE pattern is used
SELECT COUNT(*) FROM table WHERE email LIKE 'alice@%.com'

-- Correct
SELECT COUNT(*) FROM table WHERE email LIKE 'alice@example.%'
```

Insights Cloak will automatically discover which columns isolate users. This computation might be very slow and resource-intensive for large data sources. See Manually classifying isolating columns for information on alternative means of classifying isolating columns.

Column bounds

Some mathematical operations can cause an overflow and result in an error when performed in the database. To avoid such cases, Insights Cloak analyzes the query and finds potentially problematic operations, making sure that either:

- The input columns to these operations have data distributed in such a way that there won't be a problem. For example, dividing by a column that is always positive will never result in a divide-by-zero error.
- The operations are performed using a potentially slower but safe method.

Insights Cloak needs the lower and upper bounds of the values in numeric columns in order to perform this analysis. Note that the actual bounds used by Insights Cloak will be based on the true bounds, but anonymized. Note also, that any data found outside of these anonymized bounds will be treated as if it had the maximum/minimum anonymized value instead. The bounds are computed with some "extra room", so this can most often happen in the case of a value being an extreme outlier.

So long as this analysis is not complete for a certain column, mathematical operations on that column need to be performed using the safe method, wich might be slower on some data sources. For certain data sources (Microsoft SQL Server) these operations are safe by default, the analysis does not need to be performed and it won't result in any slowdown. In Oracle DB these operations are emulated by default. However, the database administrator can enable Aircloak UDFs to avoid this emulation - **see here for details**.

Column analysis

In order to apply the restrictions described in **Number of conditions**, **Isolating columns**, and **Column bounds**, Insights Cloak needs to analyze the contents of the data source. This process might take some time, but the data source is available for querying while the analysis is under way. While the analysis is incomplete Insights Cloak needs to make conservative assumptions about the data. As a result, all columns are treated as if they were isolating and had no frequent values, until the analysis is completed for a particular column.

You can check the isolator status of a table by using the `SHOW COLUMNS` statement:

```sql
SHOW COLUMNS FROM users

| name       | type    | isolator? | comment |
| ---------- | ------- | --------- | ------- |
| uid        | integer | true      |         |
| first_name | text    | false     |         |
| last_name  | text    | true      |         |
| email      | text    | pending   |         |
```

In this case the columns `uid` and `last_name` are isolating, while the column `first_name` is not. The status of the `email` column is not yet known, so it will be treated as isolating until its analysis is complete.

Best practices

Because Aircloak Insights anonymizes query results, it must be queried in a slightly different way than one would query a normal database. This guide will explain some of the peculiarities and show best practices which allow you to gain the most value from the system.

In the following examples we will pretend we are querying a database containing the following raw data:

first_name	last_name	age	zip_code	gender
Alice	Anderson	10	10000	F
Amanda	Anderson	20	11000	F
Amy	Anderson	15	11000	F
Anna	Anderson	20	11000	F
Bob	Anderson	10	10000	M
Bob	Barlow	10	10000	M
Bob	Boyle	10	10000	M
Bob	Buckner	10	10000	M

How anonymization alters results

Aircloak Insights' anonymization does not change the values in the columns you select. Instead, it alters the aggregates resulting from your queries.

In other words, if you were to write the query:

```sql
SELECT zip_code, gender, count(*)
FROM table
GROUP BY zip_code, gender
```

you would be given the exact zip code and gender values as they appear in the database. The anonymized result might look something like this:

zip_code	gender	count
11000	F	3
10000	M	4

What is anonymized is the aggregate - in this case the count. Instead of getting a count of 4 for the zip code and gender pair 10000 and M you might see a count between 2 and 6.

Aircloak Insights also filters out values that do not appear frequently enough. We call this low count filtering. In the case of the query above we see that the only female living in zip code 10000 is Alice. Instead of giving us an altered count, the system will withold this information altogether.

More details about how the aggregates get altered can be found in the understanding query results chapter of these user guides.

Column selection and its effect on anonymization

When querying a traditional database system it is common to run a query such as `SELECT *` or otherwise select a wide range of columns. This does not work well with the approach Aircloak Insights takes to anonymization.

Aircloak Insights filters out rows from the result set where the combination of values in the selected columns do not appear for enough distinct individuals in the dataset. Aircloak Insights could give you information about males living in zip code 10000 (there are 4 of them), but not about the sole female living in the same zip code.

Running the query `SELECT * FROM table` is, in the context of the dataset above, the same as running the query:

```sql
SELECT first_name, last_name, age, zip_code, gender
FROM table
```

If we look at the example data, we see that the combination of these attributes is always unique. For example while there are multiple Bob's, they all have distinct last names, and while there are quite a number of individuals with the last name Anderson, none of them share the same first name. Because of these unique combinations of columns, effectively no results can be shown, and, as a general rule, the more columns you select in a query, the lower the probability that there is data from sufficiently many distinct users that share the same attribute values and thereby won't get anonymized away.

Aircloak Insights will attempt to provide some value even in the cases where you select many columns while there is not enough unique individuals to provide values for the full set of column combinations. It will do so by selectively replacing one column at a time with a `*` (indicating that the value was anonymized away). This process takes place from the rightmost to the leftmost selected column. For the following examples we will assume the minimum threshold for the number of individuals needed to pass the low count filter is 3. The reality is somewhat more complex, but it's enough for the purposes of this example. In the example tables below we will add a metadata column as the first column, showing how many unique users share a set of column values. The initial value of 1 for each row indicates that each combination of column values only exists once:

# individuals	first_name	last_name	age	zip_code	gender
1	Alice	Anderson	10	10000	F
1	Amanda	Anderson	20	11000	F
1	Amy	Anderson	15	11000	F
1	Anna	Anderson	20	11000	F
1	Bob	Anderson	10	10000	M
1	Bob	Barlow	10	10000	M
1	Bob	Boyle	10	10000	M
1	Bob	Buckner	10	10000	M

The rightmost column is dropped first. In this case this is the gender column since it was the rightmost selected column in the query:

# individuals	first_name	last_name	age	zip_code	gender

# individuals	first_name	last_name	age	zip_code	gender
1	Alice	Anderson	10	10000	*
1	Amanda	Anderson	20	11000	*
1	Amy	Anderson	15	11000	*
1	Anna	Anderson	20	11000	*
1	Bob	Anderson	10	10000	*
1	Bob	Barlow	10	10000	*
1	Bob	Boyle	10	10000	*
1	Bob	Buckner	10	10000	*

Each set of column value combinations is still uniquely identifying a user, so the next rightmost column gets replaced. In this case that is the zip code:

# individuals	first_name	last_name	age	zip_code	gender
1	Alice	Anderson	10	*	*
1	Amanda	Anderson	20	*	*
1	Amy	Anderson	15	*	*
1	Anna	Anderson	20	*	*
1	Bob	Anderson	10	*	*
1	Bob	Barlow	10	*	*
1	Bob	Boyle	10	*	*
1	Bob	Buckner	10	*	*

The zip code column is then followed by the age column:

# individuals	first_name	last_name	age	zip_code	gender
1	Alice	Anderson	*	*	*
1	Amanda	Anderson	*	*	*
1	Amy	Anderson	*	*	*
1	Anna	Anderson	*	*	*
1	Bob	Anderson	*	*	*
1	Bob	Barlow	*	*	*
1	Bob	Boyle	*	*	*
1	Bob	Buckner	*	*	*

Even replacing the age column does not produce rows with column values that wouldn't be uniquely identifying. The next row to be taken away is the last name column:

# individuals	first_name	last_name	age	zip_code	gender
1	Alice	*	*	*	*
1	Amanda	*	*	*	*
1	Amy	*	*	*	*
1	Anna	*	*	*	*
4	Bob	*	*	*	*

When taking away the last name column, we see that there are sufficiently many individuals named Bob that it can be reported. For the remaining rows Aircloak Insights will then also try to take away the first name column, to at least give an indication of how many rows had to be filtered away. That leaves us with the final table:

# individuals	first_name	last_name	age	zip_code	gender
4	*	*	*	*	*
4	Bob	*	*	*	*

We can take away a number of things from this example:

1. The higher the number of distinct columns you select in your query, the lower the likelihood that the set of individual attributes occur frequently enough to pass the low count threshold.
2. Queries such as `SELECT * ...` will usually not yield useful information in the context of Aircloak Insights.
3. As an analyst you can directly influence the order in which Aircloak Insights will drop columns. The process takes place from right to left. Therefore you should order the columns from most to least important.
4. Selecting a column that has a unique value per user, or close to it, as one of your first columns will automatically lead to all the columns right of it being anonymized away. You are therefore likely better off selecting columns with a high number of distinct values as one of your later columns, as chances are they will get dropped.

Let us now revisit the same query again, but looking at what the result would have been if we changed the order in which we selected the columns. Let's say the query we ran was:

```sql
SELECT gender, zip_code, first_name, age, last_name
FROM table
```

Just like last time none of the rows occur frequently enough to pass the anonymizer, and Aircloak Insights will drop the rightmost column, in this case last name:

# individuals	gender	zip_code	first_name	age	last_name
1	F	10000	Alice	10	*
1	F	11000	Amanda	20	*
1	F	11000	Amy	15	*
1	F	11000	Anna	20	*
4	M	10000	Bob	10	*

We already have a value that can be reported, namely that there are 4 male Bob's living in zip code 10000 aged 10.

There are still a set of other rows that need further refining. Aircloak Insights attempts to drop the next rightmost column, namely age:

# individuals	gender	zip_code	first_name	age	last_name
1	F	10000	Alice	*	*
1	F	11000	Amanda	*	*
1	F	11000	Amy	*	*

# individuals	gender	zip_code	first_name	age	last_name
1	F	11000	Anna	*	*
4	M	10000	Bob	10	*

Dropping the age column did not make any of the other rows reach the low count threshold, so the next rightmost column is dropped: first name.

# individuals	gender	zip_code	first_name	age	last_name
1	F	10000	*	*	*
3	F	11000	*	*	*
4	M	10000	Bob	10	*

As a result of dropping the first name we now see that we can additionally report that there are 3 females living in zip code 11000.

After removing the first name column there are too few individuals remaining to produce further reportable rows, so the process is finished with the following result:

# individuals	gender	zip_code	first_name	age	last_name
3	F	11000	*	*	*
4	M	10000	Bob	10	*

Which is quite a bit more informative than what we got previously:

# individuals	first_name	last_name	age	zip_code	gender
4	*	*	*	*	*
4	Bob	*	*	*	*

Grouping values

The problem shown above of there needing to be enough distinct users with a given set of column values gets exacerbated when one of the columns contains values that are prone to have a large spread like integer values, or that are likely to be unique like floating point values or high resolution date and time values.

As an analyst you have more knowledge about what your desired outcome is than Aircloak Insights does. You might, for example, know that age groups spanning 10 years would be perfectly fine, that you only care about the first digit of a zip code or a salary amount rounded to the nearest thousand.

These types of explicit groupings of numerical values make it much more likely that values will get past the low count filter, and can be achieved by using the `bucket` function. The bucket function is described in more detail in the Supported functions chapter, but suffice to say that `bucket(age by 10)` would group the age values to the nearest 10. By default values are rounded down, such that 11, 15, as well as 19 all become 10, while 20, 21, 25 all become 20, etc.

In order to achieve our objective of getting ages by the nearest 10, and zip codes by the first digit, we could run the query:

```sql
SELECT
  bucket(age by 10) as age_bucket,
  bucket(zip_code by 10000) as zip_code_bucket
FROM table
```

Which would result in the following table of values:

# individuals	age	zip_code
6	10	10000
2	20	10000

which after low count filtering, would become:

# individuals	age	zip_code
6	10	10000

where the age group 20 to 30 got removed because there were not enough individuals within the given zip code group.

You can play similar tricks with times, dates, and datetimes by using the functions `year`, `month`, `day`, `hour`, `minute`, and `second` to extract components from the date or time, or alternatively use `date_trunc(part, dateTimeColumn)` which truncates everything going beyond a certain level of accuracy. For example `date_trunc('hour', '12:22:44.004200')` would turn the time into one at hour resolution: `12:00:00.000000`. This value is more likely to pass the low count filter than the high resolution time value would be.

Using grouping sets

A way to extract a good amount of data from Aircloak is to try different groupings of columns with grouping sets. A quick way is to use the `CUBE` clause.

```sql
SELECT first_name, last_name, gender, count(*), sum(age)
FROM table
GROUP BY CUBE(first_name, last_name, gender)
```

The query above would, after low count filtering, produce the following table:

first_name	last_name	gender	count	sum(age)
Bob	null	M	4	40
Bob	null	null	4	40
null	Anderson	F	4	65
null	Anderson	null	5	75
null	null	F	4	65
null	null	M	4	40
null	null	null	8	105

Of course Aircloak would adjust the values of the aggregates, but it still is an effective technique to get a wide view of the underlying data.

Another useful technique is by increasing resolution of timestamps:

```sql
SELECT YEAR(date), MONTH(date), DAY(date), sum(price)
FROM sales
GROUP BY ROLLUP(1,2,3)
```

would neatly show breakdowns down to days on days where there is sufficient data, but at least shows coarser aggregates for where data is insufficient.

Also Aircloak provides the `grouping_id` function that can help you identify which grouping took place.

Null values and counts of 2

In most dialects of SQL all but the `count` aggregate may produce a `null` value. The `count` aggregate would, lacking data to produce a count, return 0 rather than `null`. Aircloak Insights behaves similarly. When there is insufficient data to produce a properly anonymized aggregate but sufficient data that the set of column values passed the low count filter, then `null` will be returned for all aggregates but the `count`. As tools expect a non-`null` value for `count`s, Aircloak Insights will return a hardcoded lower bound value of 2 instead.

Say we ran the query:

```sql
SELECT last_name, avg(age)
FROM table
GROUP BY last_name
```

last_name	avg
Anderson	null
*	null

The low count filter would filter out all last names other than Anderson. In the case of the 5 Andersons we have enough users to inform you as an analyst that users with the last name of Anderson exist in the dataset, but not enough to make a properly anomymized average age. The `avg(age)` would therefore be returned as `null`.

In the case of `count` we might have enough distinct users to produce a count for the number of Anderson's, but not enough other users to generate a count for the `*` row (the anonymized row). Unlike for `avg` Aircloak Insights cannot report a `null` value as that would be incompatible with most existing tools and would return 2 instead. The presence of 2 in a `count` should therefore be considered as information about the fact that there are users with the given properties in the dataset, but not enough to produce a proper count. To validate that this is what is going on, you could also request the matching `count_noise()` value, which in this case would be `null`.

If we ran the query:

```sql
SELECT last_name, count(*), count_noise(*)
FROM table
GROUP BY last_name
```

We can validate that the count of 2 for the `*` row is due to there being insufficiently many users. The `count_noise` being `null` confirms this.

last_name	count	count_noise
Anderson	5	1.4
*	2	null

If you want to hide these `2` values from your result set you can add a `HAVING` clause to your query like this:

```sql
SELECT last_name, count(*)
FROM players
GROUP BY last_name
HAVING count(*) > 2
```

last_name	count
Anderson	5

Implicit aggregate count

If you write a query that does not contain an explicit aggregate, for example:

```sql
SELECT last_name
FROM table
```

then Aircloak Insights will internally run this query as if you had included a `count(*)`, namely the query:

```sql
SELECT last_name, count(*)
FROM table
GROUP BY last_name
```

Each occurrence of `last_name` would then be repeated `count(*)` number of times. For example the results of this query given the example dataset we have been working with in this guide would be:

last_name	count
Anderson	5
*	3

which then would result in the following table being returned to you as an analyst:

last_name
Anderson
Anderson
Anderson
Anderson
Anderson

last_name
*
*
*

When running these types of queries you might end up seeing a lot of rows appearing exactly two times. This is an artifact of what was described above, namely that the implicit `count(*)` added by Aircloak Insights returns 2 as a placeholder for the value `null` as a means of indicating that the value did not appear frequently enough to produce a properly anonymized aggregate.

Understanding query results

All anonymization systems necessarily distort query results. While the amount of distortion in Aircloak is remarkably small, the analyst must nevertheless understand how and when distortion occurs to properly interpret query results.

This section describes how and when distortion occurs, and suggests strategies for minimizing the impact of distortion.

Aircloak distorts data in the following ways:

- Adds zero mean noise to anonymizing aggregation function outputs
- May modify the values of *outlying* data (the highest or lowest values in a column)
- May suppress certain results when too few users are represented (low-count filtering)

Pro Tips

- Use the noise reporting functions (e.g. `count_noise()`) to determine how much noise is added.
- Remember that outlying values are *flattened*: sums, row counts, and maxes may be quite inaccurate when users with extreme outlying values are in the data.
- Look for the `*` output row to gauge how much data is being hidden.
- Output rows with `NULL` probably mean that there are not enough distinct users to compute an anonymized aggregate.
- In the Aircloak web interface, output rows in italics have more relative distortion.
- Queries with fewer and simpler conditions have less noise.

Zero-mean noise

Aircloak adds zero-mean Gaussian noise to the outputs of `count`, `sum`, `avg`, `stddev`, and `variance`. The *amount* (standard deviation, or *sigma*) of the noise may vary. As a rule, the noise is roughly proportional to the influence that the most influential users have on the output. For example:

- If the sum of salaries is being computed, and the highest salaries are around $1,000,000, then the sigma will be proportional to $1,000,000.
- If the count of distinct users is being computed, then the sigma will be proportional to 1 (the maximum amount that any user contributes to the count).

The reason for this is to hide the effect of the highest contributing users and thereby protect their privacy.

Aircloak increases the noise with an increase in the number of certain query *conditions* (for instance those found in the `WHERE` and `HAVING` clauses). Specifically, most conditions contribute a *baseline* of two noise samples, and some conditions contribute additional samples. These noise samples are summed together. We refer to the noise samples as *noise layers*. The following table gives the noise layers produced by each condition:

Condition	Noise Layers

Condition	Noise Layers
equality (`=` or `<>`)	Baseline (two noise layers)
Any `SELECT` 'ed column	Baseline
`concat()` in equality	Baseline
range (`>=` and `<` , or `BETWEEN`)	Baseline
`IN`	One layer plus one layer per `IN` element
`NOT IN`	Two layers per `NOT IN` element
`[I]LIKE` and `NOT [I]LIKE`	One layer plus one layer per wildcard
`right` , `left` , `ltrim` , `rtrim` , `btrim` , `trim` , or `substring`	Baseline plus one layer
`upper` , `lower` with `<>`	Baseline plus one layer
`col1 <> col2` (special case of `<>`)	No noise layer
None	One noise layer

Aircloak provides functions that report the sigma of the zero-mean noise for `count()` , `sum()` , `avg()` , `stddev()` , and `variance()` . They are `count_noise()` , `sum_noise()` , `avg_noise()` , `stddev_noise()` , and `variance_noise()` respectively. Note that the reported sigma are themselves rounded, but are generally within 5% of the true value.

Examples

Example 1

The answer to the following query indicates that noise with `sigma = 2` was added to the count:

```sql
SELECT count(*), count_noise(*)
FROM accounts
```

count	count_noise
5368	2

This is because the `accounts` table has only one row per user, and therefore the amount contributed by the most influential user is just 1.

Example 2

By contrast, for the following query, noise with sigma of roughly 340 was added:

```sql
SELECT count(*), count_noise(*)
FROM transactions
```

count	count_noise
1262167	320

The reason is that the number of transactions per user varies substantially in this table (the reported max is nearly 14000, the reported min is 5).

Example 3

The following query has noise with sigma of roughly 2:

```sql
SELECT count(*), count_noise(*)
FROM accounts
WHERE frequency = 'POPLATEK MESICNE' AND
      disp_type = 'OWNER'
```

count	count_noise
4167	4

This query has more noise than the query of example 1 above because each of the two conditions adds two noise layers. Each layer has `sigma = 2`, so the resulting cumulative sigma is `sqrt(4) * 2 = 4`.

Example 4

The following query produces answer rows with `sigma = 4`. This represents 16 noise layers: 2 for `acct_date`, 6 for the `LIKE` condition, and 8 for the `IN` condition.

```sql
SELECT acct_date, count(*), count_noise(*)
FROM accounts
WHERE frequency LIKE 'P_PL_TE_ ME_I_NE' AND
      acct_district_id IN (30,31,33,35,37,39,41)
GROUP BY acct_date
```

Low-count filtering

The Aircloak anonymizing aggregator computes outputs from the data of multiple users. If the number of distinct users contributing to a single output row is too small, the row is suppressed (not reported). This suppression is called *low-count filtering*.

The threshold for the low-count filter is itself a noisy value with an average of 4. If there are 4 distinct users that comprise an output row, then there is a 50% chance the row will be suppressed. Fewer users increases the chance of suppression, and more users decreases the chance of suppression. Any reported output row always has at least 2 distinct users.

For instance, suppose that a query counts the users with each given first name, and that the names in the `users` table (before anonymization) are distributed as follows:

Name	Number of distinct users
Alice	100
Bob	2
John	150
Mary	1
Tom	2

Since there is only one Mary, she definitely won't appear in the output. Since there are only two Bobs and Toms, their names probably won't appear in the output. Therefore, the anonymized result returned by Aircloak may be something like:

Name	Number of distinct users
Alice	102
John	147
*	7

The `*` row provides the analyst with an indication that some names have been suppressed because of low-count filtering. This indication is particularly important in cases where a large number of values are low-count filtered: the analyst can learn that a substantial amount of data is being hidden. Note that the `*` row is itself anonymized: the anonymized aggregate associated with it has noise, and it itself is low-count filtered. In other words, lack of a `*` row does not mean that no data was suppressed, only that very little data was suppressed.

When a large number of non-aggregated columns is selected in a query, the chances of having lots of rows with very few users increase. That will lead to lots of rows being suppressed, making the query result less useful. In order to suppress as little information as possible, Aircloak will low-count filter columns individually, from right to left. Rows that are suppressed in one iteration are aggregated together and kept for the next round of filtering. That way, the maximum number of rows will be sent back to the analysts.

If the following query is issued:

```sql
SELECT name, age, COUNT(DISTINCT uid)
FROM table
GROUP BY name, age
```

and the non-anonymized results are:

name	age	count	sufficient users
Alice	10	2	false
Alice	20	2	false
Bob	30	1	false
Cynthia	40	2	false

and the system only allows through values where there 3 or more distinct users in the answer set, then the Insights Cloak will attempt to group the low-count values together by the `age` column, and, where necessary, also by the `name` column, as follows:

Step 1: Suppress `age` where necessary

name	age	count	sufficient users
Alice	*	4	true
Bob	*	1	false
Cynthia	*	2	false

Step 2: Suppress `name` where necessary

name	age	count	sufficient users
Alice	*	4	true
*	*	3	true

This process is time-consuming, so it is limited by default to a maximum of 3 columns. For details on how to change this limit, refer to the **Configuring the Insights Cloak** section. A value of 1 results in a single bucket for suppressed data, while a value of 0 will drop the low-count filtered data completely.

Anonymizing aggregation functions

These seven anonymizing aggregation functions may add additional distortion besides the zero mean noise and low-count filtering already described. Note in particular that Aircloak gives no indication of whether any additional distortion occurred, or how severe this additional distortion is. This is because such information itself may leak individual information.

Anonymizing aggregation functions make a variety of computations that require some minimum number of distinct users. It can happen that there are enough distinct users to pass a low-count filter, but not enough distinct users to compute the aggregate. In these cases, Aircloak does not suppress the output, but rather reports `NULL` for all aggregation functions and noise reporting functions except `count()`. Aircloak reports 0 for `count()` because `NULL` is not a valid output for `count()`.

Note also that when there are fewer than 15 distinct users (anonymized output) in a given output row, then the Aircloak web interfaces reports the output in italics. This serves as a reminder to the analyst that the result likely has high relative noise.

sum()

The `sum()` function selects a small number of the highest values, and *flattens* them so that they are roughly the same. In other words, a few high values are lowered, or in the case of negative numbers, a few low values are increased. As a result, the users with high values become a homogeneous group of users within which individual users can hide. The number of users chosen for flattening is itself a noisy value.

By way of example, suppose that values in a given summed column contain the following numbers of distinct users:

Value	Number of distinct users
1	1000
500	20
500K	1
1M	1

Aircloak will flatten the high values by modifying them to fall within a group of high value users. In this example, the high value group has the value 500, and so the users with values 500K and 1M are replaced with 500. The resulting values are these:

Value	Number of distinct users
1	1000

Value	Number of distinct users
500	22

The users with 500K and 1M have, essentially, disappeared from the system. The affect is similar to outlier removal in statistics, and the analyst needs to be aware that this is happening, and that there is no indication from Aircloak that it has happened.

In addition, the sigma of the noise is proportional to the average value of the modified group of high users (in this case 500). This can be observed from the `sum_noise()` function.

In any event, as a result of this flattening, the answer distortion in this particular extreme case is very large. The anonymized answer will be in the neighborhood of 12K where the true answer is over 1.5M. More generally, the amount of distortion depends on how big the outlying values are relative to other values.

count()

The `count()` function actually uses the `sum()` function, where the number of rows contributed by each user is the value being summed. As such, one or a small number of users with a high number of rows will be flattened.

Note that when counting distinct users, there is no added distortion.

avg()

The `avg(col)` function is literally the result of the `sum(col)` function divided by the result of the `count(col)` function. As such, it also flattens the high (or negative low) users.

stddev() and variance()

The functions `stddev()` and `variance()` use the `avg()` function, and so flattening occurs.

max() and min()

The `max()` function drops the rows for a small number of users with the highest values (using a noisy number of users as with `sum()`). It then takes the average value of the next small number of distinct users with the highest values, and uses this average as the max (with potentially some additional noise, if the spread among this set of values is high). As such, the anonymized max may be very far from the true max.

The `min()` function operates the same as `max()`, except using low numbers. Unless the data includes negative numbers, `min()` tends to have less distortion than `max()`.

Functions

Date/time functions

Current date/time functions

The functions `current_date`, `current_time`, `current_datetime`, `now` and `current_timestamp` are supported. They are evaluated during query compilation and replaced with the current date and/or time value.

```sql
current_date()
-- 2016-05-22

-- current_timestamp(), now() and current_datetime() are
identical
current_timestamp()
-- 2016-05-22 12:30:21.340

current_time()
-- 12:22:44.220
```

Date extraction functions

The functions `year`, `quarter`, `month`, `day`, `hour`, `minute`, `second`, `weekday`, and `dow` (a synonym for `weekday`) are supported. They extract the named part from a date or time column.

Functions `weekday` and `dow` return values in interval 1 (Sunday) to 7 (Saturday). This behavior may change if database defaults are modified.

```sql
SELECT YEAR(date_column), MONTH(date_column), DAY(date_column)
FROM table;

SELECT EXTRACT(year FROM date_column) FROM table;
```

date_trunc

"Rounds" the date or time to the given precision. Supported precision levels are `year`, `quarter`, `month`, `day`, `hour`, `minute`, and `second`.

```sql
DATE_TRUNC('quarter', date)
-- 2016-05-22 12:30:00.000 -> 2016-04-01 00:00:00.000

DATE_TRUNC('hour', time)
-- 12:22:44.000 -> 12:00:00.000
```

Working with intervals

When subtracting two date or time columns the result is an interval. The format Aircloak follows when representing intervals is ISO-8601.

```sql
cast('2017-01-02' as date) - cast('2017-01-01' as date)
-- P1D

cast('12:33:44' as time) - cast('11:22:33' as time)
-- PT1H11M11S

cast('2017-02-03 11:22:33' as timestamp) - cast('2016-01-02
12:33:44' as timestamp)
-- P1Y1M2DT22H48M49S

-- Intervals do not have a sign
cast('12:00:00' as time) - cast('13:00:00' as time)
-- PT1H
cast('13:00:00' as time) - cast('12:00:00' as time)
-- PT1H
```

Similarly, an interval can be added or subtracted from a date or time column.

```sql
cast('13:00:00' as time) + interval 'PT1H2M3S'
-- 14:02:03

cast('2015-07-06 12:00:00' as timestamp) - interval
'P1Y1M1DT1H1M1S'
-- 2014-06-05 10:58:59

-- Note that months in intervals will always have 30 days
cast('2015-06-06' as date) - interval 'P1M'
-- 2015-05-07 00:00:00
cast('2015-07-06' as date) - interval 'P1M'
-- 2015-06-06 00:00:00

-- Similarly years will always have 365 days
cast('2015-06-06' as date) + interval 'P1Y'
-- 2016-06-05 00:00:00
cast('2016-06-06' as date) + interval 'P1Y'
-- 2017-06-06 00:00:00
```

Intervals can be multiplied or divided by numbers to yield bigger or smaller intervals.

```sql
2 * interval 'P1Y'
-- P2Y

0.5 * interval 'P1M'
-- P15D

interval 'PT1H' / 2
-- PT30M
```

Restrictions in usage apply

Mathematical operators

The operators `+`, `-`, `/`, and `*` have their usual meaning of addition, subtraction, division, and multiplication respectively.

```sql
1 - 2 + 4 * 3 / 2
-- 5
```

The operator `^` denotes exponentiation.

```
2 ^ 3
-- 8

-- Note that ^ is left-associative

2 ^ 3 ^ 4
-- 4 096

2 ^ (3 ^ 4)
-- 2 417 851 639 229 258 300 000 000
```

The operator `%` computes the division remainder:

```
33 % 10
-- 3
```

Restrictions in usage apply

Mathematical functions

abs

Computes the absolute value of the given number.

```sql
ABS(3)
-- 3

ABS(-3)
-- 3
```

Restrictions in usage apply

bucket

Rounds the input to multiples of N, where N is provided in the `BY` argument. It also accepts an `ALIGN` argument to specify if the rounding should occur down (`ALIGN LOWER` - this is the default), up (`ALIGN UPPER`), or if an average between the two should be returned (`ALIGN MIDDLE`).

```sql
BUCKET(180 BY 50)
-- 150

BUCKET(150 BY 50)
-- 150

BUCKET(200 BY 50)
-- 200

BUCKET(180 BY 100)
-- 100

BUCKET(180 BY 50 ALIGN LOWER)
-- 150

BUCKET(180 BY 50 ALIGN UPPER)
-- 200

BUCKET(180 BY 50 ALIGN MIDDLE)
-- 175
```

This function is useful to prepare buckets/bins for a histogram, for example in a query like the following:

```sql
SELECT BUCKET(price BY 5), COUNT(*)
FROM purchases
GROUP BY 1
-- bucket count
-- 0      10     - all purchases priced below 5
-- 5      10     - purchases priced at or above 5 and below 10
-- 10     20     - purchases priced at or above 10 and below 15
-- etc.
```

The function can also help if the column you want to group by has many unique values and many of the buckets get anonymized away. For example if you have a column containing the length of a call in seconds:

```sql
SELECT call_duration, COUNT(*)
FROM calls
GROUP BY 1
-- call_duration count
-- *             100

SELECT BUCKET(call_duration BY 5), COUNT(*)
FROM calls
GROUP BY 1
-- bucket count
-- *      20
-- 0      10
-- 5      10
-- etc.
```

Restrictions in usage apply

ceil / ceiling

Computes the smallest integer that is greater than or equal to its argument.

```sql
CEIL(3.22)
-- 4
```

Restrictions in usage apply

floor

Computes the largest integer that is less than or equal to its argument.

```sql
FLOOR(3.22)
-- 3
```

Restrictions in usage apply

pow

`POW(a, b)` computes `a` to the `b` -th power. Returns NULL if `a` is negative.

```sql
POW(2, 3)
-- 8

POW(2, 3.5)
-- 11.313708498984761
```

Restrictions in usage apply

round

Rounds the given floating-point value to the nearest integer. An optional second argument signifies the precision. Halves are rounded away from zero (towards the larger absolute value).

```sql
ROUND(3.22)
-- 3

ROUND(3.99)
-- 4

ROUND(3.22, 1)
-- 3.2
```

Restrictions in usage apply

sqrt

Computes the square root.

```sql
SQRT(2)
-- 1.4142135623730951
```

Restrictions in usage apply

trunc

Rounds the given floating-point value towards zero. An optional second argument signifies the precision.

```sql
TRUNC(3.22)
-- 3

TRUNC(-3.22)
-- -3

TRUNC(3.22, 1)
-- 3.2
```

Restrictions in usage apply

String operators

||

Joins two or more strings into one. It is internally translated to the `concat` function.

```sql
'a' || 'b' || 'c'
-- 'abc'
```

Restrictions in usage apply

LIKE / ILIKE

These operators match a text expression against a pattern. `ILIKE` is the case-insensitive version of `LIKE`. Syntax: `text_expression [NOT] LIKE | ILIKE string_pattern [ESCAPE escape_string]`.

If the pattern does not contain any percent signs or underscores, then the pattern only represents the string itself. In that case, `LIKE` acts like the equals operator and `ILIKE` acts like a case-insensitive equals operator. An underscore (`_`) in a pattern matches any single character. A percent sign (`%`) matches any string of zero or more characters.

A pattern match needs to cover the entire string. To match a sequence anywhere within a string, the pattern must therefore start and end with a percent sign.

To match a literal underscore or percent sign without matching other characters, the respective character in the pattern must be preceded by the escape character. No escape character is set by default. To match the escape character itself, write two escape characters.

```sql
'abCD' LIKE 'ab%'
-- TRUE

'abCD' NOT LIKE 'ab__'
-- FALSE

email ILIKE 'a\_b@%.com' ESCAPE '\'
-- BOOLEAN
```

Restrictions in usage apply

String functions

btrim

Removes all of the given characters from the beginning and end of the string. The default is to remove spaces.

```sql
BTRIM(' some text ')
-- 'some text'

BTRIM('xyzsome textzyx', 'xyz')
-- 'some text'
```

Restrictions in usage apply

concat

Joins the passed strings into one.

```sql
CONCAT('some ', 'text')
-- 'some text'

CONCAT('a', 'b', 'c')
-- 'abc'

'a' || 'b' || 'c'
-- 'abc'
```

Restrictions in usage apply

left

`LEFT(string, n)` takes n characters from the beginning of the string. If n is negative takes all but the last |n| characters.

```sql
LEFT('some text', 4)
-- 'some'

LEFT('some text', -2)
-- 'some te'
```

Restrictions in usage apply

length

Computes the number of characters in the string.

```sql
LENGTH('some text')
-- 9
```

Restrictions in usage apply

lower

Transforms all characters in the given string into lowercase.

```sql
LOWER('Some Text')
-- 'some text'

LCASE('Some Text')
-- 'some text'
```

ltrim

Removes all of the given characters from the beginning of the string. The default is to remove spaces.

```sql
LTRIM(' some text ')
-- 'some text '

LTRIM('xyzsome textzyx', 'xyz')
-- 'some textzyx'
```

Restrictions in usage apply

right

`RIGHT(string, n)` takes n characters from the end of the string. If n is negative takes all but the first |n| characters.

```sql
RIGHT('some text', 4)
-- 'text'

RIGHT('some text', -2)
-- 'me text'
```

Restrictions in usage apply

rtrim

Removes all of the given characters from the end of the string. The default is to remove spaces.

```sql
RTRIM(' some text ')
-- ' some text'

RTRIM('xyzsome textzyx', 'xyz')
-- 'xyzsome text'
```

Restrictions in usage apply

substring

Takes a slice of a string.

```sql
SUBSTRING('some text' FROM 3)
-- 'me text'

SUBSTRING('some text' FROM 3 FOR 5)
-- 'me te'

SUBSTRING('some text' FOR 4)
-- 'some'
```

Restrictions in usage apply

trim

Removes all of the given characters from the beginning and/or end of the string. The default is to remove spaces from both ends.

```sql
TRIM(' some text ')
-- 'some text'

TRIM(LEADING ' some text ')
-- 'some text '

TRIM(TRAILING ' tx' FROM ' some text ')
-- ' some te'

TRIM(' osxt' ' some text ')
-- 'me te'

TRIM(BOTH FROM ' some text ')
-- 'some text'
```

Restrictions in usage apply

upper

Transforms all characters in the given string into uppercase.

```sql
UPPER('Some Text')
-- 'SOME TEXT'

UCASE('Some Text')
-- 'SOME TEXT'
```

Casting

You can convert values between different types using a cast expression.

```sql
CAST('3' AS integer)
-- 3

'3'::integer
-- 3

CAST(3, text)
-- '3'

CAST('NOT A NUMBER', integer)
-- NULL
```

Restrictions in usage apply

Types can be converted according to the following tables:

from\to	text	integer	real	boolean
text	✓	✓	✓	✓
integer	✓	✓	✓	✓
real	✓	✓	✓	✓
boolean	✓	✓	✓	✓
date	✓			
time	✓			
datetime	✓			
interval	✓			

from\to	date	time	datetime	interval
text	✓	✓	✓	✓
integer				
real				
boolean				
date	✓			
time		✓		
datetime	✓	✓	✓	
interval				✓

A cast may fail even when it's valid according to the table. For example a text field may contain data that does not have the correct format. In that case a `NULL` is returned.

Casting to/from text

Casting from text will accept the same format as the cast to text produces for the given type. That means:

- `'TRUE'` / `'FALSE'` for booleans
- A base-10 notation for integers

- `1.23` or `1e23` for reals
- ISO-8601 notation for dates, times, datetimes and intervals

Casting to integer

Casting a real to integer rounds the number to the closests integer.

Casting to/from boolean

When converting numbers to booleans non-zero numbers are converted to `TRUE` and zero is converted to `FALSE`. When converting from booleans `TRUE` is converted to `1` and `FALSE` is converted to `0`.

Casting from datetime

Casting from datetime to date or time will select the date/time part of the datetime.

Aggregation functions

Note about noise

Unlike in regular database systems, the results of aggregation functions will usually not be reported precisely. Instead, a small amount of noise will be added or subtracted from the real value to preserve anonymity. See the section about *_noise functions for more on how to get a measure of how much noise is added.

avg

Computes the average of the given expression.

```sql
SELECT avg(age) FROM people

         avg
  -------------------
   29.44782928323982

SELECT lastname, avg(age) FROM people GROUP BY 1

   lastname |        avg
  ----------+--------------------
   ABBOTT   | 28.930111858960856
   ACEVEDO  | 29.933255031072672
   ...      | ...
```

Note that the computed average is anonymized by introducing a certain amount of noise. See Note about noise for more.

count

Computes the number of rows for which the given expression is non-NULL. Use `*` as an argument to count all rows.

```sql
SELECT count(age) FROM people

   count
  -------
   10000

SELECT lastname, count(age) FROM people GROUP BY 1

   lastname | count
  ----------+-------
   ABBOTT   |    10
   ACEVEDO  |    12
   ...      |   ...
```

Note that in order to preserve anonymity Insights Cloak will never report "groups" of just one user and there will be noise added to the result. Because of this, when you see a count of 2, it should be treated as a placeholder value for "a small number, not lower than 2". See Note about noise for more.

max

Finds the maximum value of the given expression.

```sql
SELECT max(age) FROM people

   max
  -----
    43

SELECT lastname, max(age) FROM people GROUP BY 1

   lastname | max
  ----------+-----
   ABBOTT   |  30
   ACEVEDO  |  32
   ...      | ...
```

Note that the computed max value is anonymized - it requires a number of users to share this value, so in many cases the true value will be larger. Furthermore, Insights Cloak's anonymizing `max` function doesn't work on textual values:

```sql
SELECT max(lastname) FROM people

  ERROR:  Aggregator `max` is not allowed over arguments of type
`text` in anonymizing contexts.
  For more information see the "Text operations" subsection of
the "Restrictions" section in the user guides.
```

However, you can still use `max` to postprocess textual results of an anonymizing subquery:

```sql
SELECT max(lastname) FROM (SELECT lastname FROM people GROUP BY 1) x

    max
--------
  ZUNIGA
```

min

Finds the minimum value of the given expression.

```sql
SELECT min(age) FROM people

  min
-----
  16

SELECT lastname, min(age) FROM people GROUP BY 1

  lastname | min
-----------+-----
  ABBOTT   |  26
  ACEVEDO  |  28
  ...      | ...
```

Note that the computed min value is anonymized - it requires a number of users to share this value, so in many cases the true value will be smaller. Furthermore, Insights Cloak's anonymizing `min` function doesn't work on textual values:

```sql
SELECT min(lastname) FROM people

  ERROR:  Aggregator `min` is not allowed over arguments of type `text` in anonymizing contexts.
  For more information see the "Text operations" subsection of the "Restrictions" section in the user guides.
```

However, you can still use `min` to postprocess textual results of an anonymizing subquery:

```sql
SELECT min(lastname) FROM (SELECT lastname FROM people GROUP BY 1) x

    min
--------
  ABBOTT
```

stddev

Computes the sample standard deviation of the given numerical expression.

```sql
SELECT stddev(age) FROM people

       stddev
  -------------------
   4.032164086149982

SELECT lastname, stddev(age) FROM people GROUP BY 1

   lastname |       stddev
  ----------+--------------------
   ABBOTT   | 7.2835052504058195
   ACEVEDO  |  1.587458159104735
   ...      |                ...
```

Note that the computed standard deviation is anonymized by introducing a certain amount of noise. See Note about noise for more.

sum

Computes the sum of the given numerical expression.

```sql
SELECT sum(points) FROM games

    sum
  ---------
   6390144

SELECT date, sum(points) FROM games GROUP BY 1

      date    | sum
  ------------+------
   2013-01-01 | 5510
   2013-01-02 | 6761
   ...        |  ...
```

Note that the computed sum is anonymized by introducing a certain amount of noise. See Note about noise for more.

variance

Computes the sample variance of the given numerical expression.

```sql
SELECT variance(age) FROM people

      variance
  -------------------
   16.25834721763772

SELECT lastname, variance(age) FROM people GROUP BY 1

   lastname |      variance
  ----------+--------------------
   ABBOTT   |  53.04944873268914
   ACEVEDO  | 2.5200234069081944
   ...      |                ...
```

Note that the computed variance is anonymized by introducing a certain amount of noise. See Note about noise for more.

*_noise

You can get a sense of how much noise is being added to an `avg`, `count`, `stddev`, `sum`, or `variance` expression by using an analogous `*_noise` expression. The value returned is the standard deviation of the noise added according to what's described in the section about noise.

```sql
SELECT count(*), count_noise(*), avg(age), avg_noise(age) FROM
people

   count | count_noise |        avg        | avg_noise
  -------+-------------+-------------------+-----------
   10000 |         1.0 | 29.44782928323982 |    0.0029

SELECT lastname, count(*), count_noise(*), avg(age),
avg_noise(age) FROM players GROUP BY 1

   lastname | count |    count_noise     |        avg         |
avg_noise
  ----------+-------+--------------------+--------------------+--
---------
   ABBOTT   |    10 | 1.4000000000000001 | 28.930111858960856 |
4.2
   ACEVEDO  |    12 | 1.4000000000000001 | 29.933255031072672 |
3.5
   ...      |   ... |                ... |                ... |
...
```

Note that the noise added depends on the expression inside the aggregation function used, so you have to provide the exact same expression in the `*_noise` function to get an accurate value:

```sql
SELECT avg(age), avg_noise(age),
  avg(age * age) AS square, avg_noise(age * age) AS square_noise
FROM people

         avg        | avg_noise |      square       |
square_noise
  ------------------+-----------+-------------------+-----------
---
   29.44782928323982 |    0.0029 | 883.4329967124744 |
0.09
```

Special functions

grouping_id

Returns an integer bitmask for the columns used in the current grouping set. Can only be used in the `SELECT`, `HAVING` and `ORDER BY` clauses when the `GROUP BY` clause is specified.

Each `grouping_id` argument must be an element of the `GROUP BY` list. Bits are assigned with the rightmost argument being the least-significant bit; each bit is 0 if the corresponding expression is included in the grouping criteria of the grouping set generating the result row, and 1 if it is not.

```sql
SELECT
   alive,
   bucket(age by 10) as age,
   count(*),
   grouping_id(alive, bucket(age by 10))
FROM demographics
GROUP BY CUBE (1, 2)

   alive |  age  | count | grouping_id
  -------+-------+-------+-------------
   false |   *   |  10   |       0
   false |  20   |   7   |       0
   true  |  10   |   3   |       0
   true  |  20   |   2   |       0
   true  |  30   |   4   |       0
   false |       |  14   |       1
   true  |       |  13   |       1
         |   *   |   2   |       2
         |  10   |   7   |       2
         |  20   |  10   |       2
         |  30   |   6   |       2
         |       |  31   |       3
```

Overview

Before starting the system, you need to configure the Insights Air and Insights Cloak components. Both components are configured through a file called `config.json`. Each component requires its own `config.json` file which must be placed in a separate folder. In other words, you can't have a single `config.json` for both components. When starting each component, you need to mount the folder containing this file, as explained in the Installation guide.

Insights Air configuration

The Insights Air configuration needs to provide the following information:

- database connection parameters (required) - see Database configuration
- web site configuration (required) - see Web site configuration
- Insights Air PostgreSQL interface parameters (optional) - see Insights Air Postgresql interface configuration
- LDAP configuration (optional) - see LDAP configuration
- Configuration for connecting to Diffix Explorer (optional) - see Diffix Explorer configuration

The general shape of `config.json` is therefore:

```
{
  "name": string,
  "database": {
    ...
  },
  "site": {
    ...
  },
  "psql_server": {
    ...
  },
  "ldap": {
    ...
  },
  "explorer": {
    ...
  }
}
```

The `name` property is used to uniquely identify the air instance in the system. If you're running multiple instances, make sure to give each instance a unique name.

Database configuration

The database configuration is used to specify connection parameters for the database used by the Insights Air component to store data such as users, groups, history of queries, and other. The database has to be hosted on a PostgreSQL server version 9.4 or higher.

This section looks as follows:

```
"database": {
  "host": string,
  "port": integer,
  "ssl": boolean,
  "user": string,
  "password": string,
  "name": string
}
```

The following fields are optional:

- `port` - defaults to 5432
- `ssl` - defaults to true
- `password` - defaults to empty string

Web site configuration

This part of the configuration is used to configure the web server of the Insights Air component. The shape of this section is as follows:

```
"site": {
  "auth_secret": secret_string,
  "endpoint_key_base": secret_string,
  "endpoint_public_url": string,
  "cloak_secret": secret_string,
  "master_password": string,
  "certfile": string,
  "keyfile": string,
  "privacy_policy_file": string,
  "license_file": string,
  "users_and_datasources_file": string,
  "browser_long_polling": boolean
},
```

In the snippet above, the type `secret_string` indicates a string which should consist of at least 64 characters. The corresponding parameters are used to sign and encrypt various data. Make sure to choose values which are random enough, or otherwise the security of the system might be compromised. For example, to generate a random secret, you can use the following command:

```
cat /dev/urandom |
  LC_ALL=C tr -dc 'a-zA-Z0-9' |
  fold -w 64 |
  head -n 1
```

Of the parameters above, the only required ones are the `auth_secret` and `endpoint_key_base` parameters, as well as one of `master_password` or `users_and_datasources_file`. The other parameters such as the `privacy_policy_file`, `users_and_datasources_file`, and `license_file` all specify values that can also be configured in the Insights Air web interface. These parameters can be used to fully configure a system ahead of time. This is useful when performing automated deployments. For more information on ahead of time configuration, please read the ahead of time configuration guide.

The `master_password` parameter specifies the password (in clear text) which is required when creating the first administrator in the Insights Air web interface. If you attempt to access the Insights Air interface while no administrative user has been setup, you will be prompted to create one. To do so you have to type in the `master_password` the system is configured with. This password will no longer be needed once the first administrator has been created.

The `endpoint_public_url` should be the full root URL that the Insights Air instance is accessible on the internet. It should be the address you would go to when visiting Insights Air using the browser. In other words, it should also include `http://` or `https://` . This parameter is used to generate correct URLs.

The `cloak_secret` setting is optional. If not set (default) all Insights Cloak instances will be allowed to connect to the Insights Air instance. If set, then only instances with the same `cloak_secret` set in their configuration files will be allowed. See **Insights Cloak configuration** for more.

The final two parameters `certfile` and `keyfile` are optional. They are used to specify the certificate and key for the HTTPS interface. If these parameters are provided, you will also need to put the corresponding files in the same folder as the `config.json` file. Once you do that, the site will accept HTTPS traffic as well as HTTP traffic. If you omit these parameters, the site will only accept HTTP traffic.

The ports on which the site will listen are hardcoded. HTTP traffic is served via port 8080, while HTTPS is served via 8443. As explained in the **Installation guide**, you can use the Docker port mapping option to decide under which port numbers you want to expose these endpoints on the host server.

We strongly suggest only exposing the Insights Air interface to clients using HTTPS. You might want to terminate the SSL connection at a reverse proxy such as **nginx** or **apache**, or alternatively make use of the HTTPS server offered as part of Insights Air.

By default, when Insights Air is accessed from the browser, a websocket connection is established. This connection is used to push real-time notifications in various situations. If Insights Air is behind a proxy, and you don't want to allow forwarding of websocket connections, you can explicitly force the long polling protocol. This can be done by setting the `browser_long_polling` option to `true` .

Insights Air PostgreSQL interface configuration

This part of the configuration allows you to instruct the Insights Air component to accept requests over the PostgreSQL wire protocol. If this is configured, Insights Air can be queried from client applications which understand this protocol, such as Tableau.

The configuration consists of the following parameters:

```
"psql_server": {
  "require_ssl": boolean,
  "certfile": string,
  "keyfile": string,
  "max_connections": positive_integer
}
```

The `require_ssl` parameter specifies whether the connection requires all clients to connect over SSL. If this value is `true`, you also need to provide `certfile` and `keyfile` parameters which specify the file names of the certificate and the key. These files need to be placed in the same folder as the `config.json` file.

If `require_ssl` is false, then the server will accept TCP connection as well as SSL. However, if `certfile` and `keyfile` parameters are not provided, then the server will only work with unencrypted TCP connections.

Regardless of which transport protocol(s) are allowed, the server will always accept requests on the port 8432. As explained in the **Installation guide**, you can use the docker port mapping to expose this port to the outside world.

Once the component is started, you can test the connectivity with the `psql` command line tool:

```
psql -h insights_air_ip_address -p postgresql_interface_port -d
data_source_name -U user_name
```

Where `postgresql_interface_port` is the PostgreSQL interface port provided when the component is started, as explained in the **Installation Guide**.

In order for the above command to work, the cloak component must be started as well, and the user must have permissions to query the given data source.

The `max_connections` property can be used to configure the maximum allowed number of simultaneously open connections. The incoming connections which would cause the limit to be exceeded are immediately closed. This property is optional, and if not provided, the default value of 1024 is used.

LDAP configuration

Insights Air can be configured to allow users to login with credentials managed in an LDAP directory service. Note that this feature is licensed separately. If you would like to add LDAP sync to your license, contact **support@aircloak.com**. The `config.json` snippet below shows all possible configuration options along with their default values where applicable. Note that the `host`, `user_base`, and `group_base` options are required. Options without a default value are indicated with a `null`.

```
{
  ...
  "ldap": {
    "host": null,
    "port": 389,
    "bind_dn": "",
    "password": "",
    "encryption": "plain",
    "verify_server_certificate": false,
    "ca_certfile": null,
    "client_certfile": null,
    "client_keyfile": null,
    "user_base": null,
    "user_filter": "(objectClass=*)",
    "user_login": "cn",
    "user_name": "cn",
    "group_base": null,
    "group_filter": "(objectClass=*)",
    "group_name": "dn",
    "group_member": "memberUid",
    "group_member_key": "login",
    "user_group": null
  }
}
```

The options have the following meaning:

- `host` - the hostname of the LDAP server.
- `port` - the port on which to connect to the LDAP server. Defaults to 389.
- `bind_dn` - the DN of the user used to read from the LDAP server. We recommend you set up a read-only user for this purpose. Defaults to `""`.
- `password` - the password of the user used to read from the LDAP server. You can set both `bind_dn` and `password` to `""` to configure anonymous access. Defaults to `""`.
- `encryption` - the type of encryption to use. Possible values are `"plain"` for no encryption, `"ssl"` for regular SSL, and `"start_tls"` for StartTLS. Set this to the type of encryption used by your server. We recommend you use either `"ssl"` or `"start_tls"`. Defaults to `"plain"`.
- `verify_server_certificate` - set this to `true` to check the certificate of the server for validity. Requires `ca_certfile` to be configured. Defaults to `false`.
- `ca_certfile` - the name of the CA certificate file with which to verify the server certificate. Put the certificate file in the same folder as `config.json`.
- `client_certfile` - the name of the client certificate file to use when connecting to the server. Put the certificate file in the same folder as `config.json`. By default no client certificate is sent.
- `client_keyfile` - the name of the file containing the key to `client_certfile`. Put the key file in the same folder as `config.json`.
- `user_base` - the LDAP subtree in which to look for users.
- `user_filter` - an LDAP filter to restrict which users to sync from `user_base`. See **the LDAP page on filters** for more on how to formulate such filters. Defaults to `"(objectClass=*)"`, which matches all objects.
- `user_login` - the name of the attribute from which to take the user's login. Note that users are required to have a valid login, so if this attribute is empty for an object, it won't be synced as an Insights Air user. Defaults to `"cn"`.

- `user_name` - the name of the attribute from which to take the user's name. Defaults to `"cn"`.
- `group_base` - the LDAP subtree in which to look for groups.
- `group_filter` - an LDAP filter to restrict which groups to sync from `group_base`. See **the LDAP page on filters** for more on how to formulate such filters. Defaults to `"(objectClass=*)"`, which matches all objects.
- `group_name` - the name of the attribute from which to take the group's name. Note that groups are required to have a valid name, so if this attribute is empty for an object, it won't be synced as an Insights Air group. Defaults to `"dn"`.
- `group_member` - the name of the attribute on a group object which lists the group's members. Defaults to `"memberUid"`.
- `group_member_key` - the user attribute which will be listed in group objects under `group_member`. Possible values are `"login"` and `"dn"`. Defaults to `"login"`.
- `user_group` - the name of the attribute on a user object which lists the groups the user belongs to. The attribute is expected to contain the DNs of the groups.

If a valid LDAP configuration is present, Insights Air will periodically sync with the LDAP server to update the list of users and groups. The syncs will occur immediately after Insights Air starts and every hour after that. You can also trigger a sync manually by going to `Admin -> Users` or `Admin -> Groups` and clicking `Sync Now` next to the LDAP section.

The users and groups created in this way can only be managed in LDAP. That is, their details such as user logins, user names, and group names cannot be altered through the Insights Air interface. Furthermore, group membership can also only be altered through LDAP. The only property that can be modified through the Insights Air interface is the list of data sources available to a given group.

The users synchronized from LDAP will be able to login using their LDAP password and the login configured with `user_login`. They cannot login using an Insights Air-specific password nor can they change their password via Insights Air.

When a user is removed from LDAP they will be disabled in Insights Air during the next sync. Only then can the user be removed from Insights Air. Note that if a user with the same LDAP DN appears again in LDAP then the user will be enabled and synchronized with this new user. Users who do not match the `user_filter` are treated as non-existent, so you can disable users by adjusting that filter.

When a group is removed from LDAP that group will be deleted in Insights Air during the next sync.

Examples

If your LDAP data looks something like this:

```
dn: ou=users,dc=example,dc=org
objectClass: organizationalUnit

dn: cn=alice,ou=users,dc=example,dc=org
objectClass: simpleSecurityObject
objectClass: organizationalRole
cn: alice
description: Alice Liddell

dn: ou=groups,dc=example,dc=org
objectClass: organizationalUnit

dn: cn=analysts,ou=groups,dc=example,dc=org
objectClass: posixGroup
cn: analysts
description: Wonderland Analysts
memberUid: alice
```

You might have the following LDAP configuration:

```
{
  ...
  "ldap": {
    "host": "ldap.example.org",
    "user_base": "ou=users,dc=example,dc=org",
    "user_login": "cn",
    "user_name": "description",
    "group_base": "ou=groups,dc=example,dc=org",
    "group_name": "description",
    "group_member": "memberUid",
    "group_member_key": "login"
  }
}
```

Your group membership might be specified in user attributes instead:

```
dn: ou=users,dc=example,dc=org
objectClass: organizationalUnit

dn: cn=alice,ou=users,dc=example,dc=org
objectClass: simpleSecurityObject
objectClass: organizationalRole
cn: alice
description: Alice Liddell
group: cn=analysts,ou=groups,dc=example,dc=org

dn: ou=groups,dc=example,dc=org
objectClass: organizationalUnit

dn: cn=analysts,ou=groups,dc=example,dc=org
objectClass: posixGroup
cn: analysts
description: Wonderland Analysts
```

In that case you'd use a configuration like this:

```
{
  ...
  "ldap": {
    "host": "ldap.example.org",
    "user_base": "ou=users,dc=example,dc=org",
    "user_login": "cn",
    "user_name": "description",
    "group_base": "ou=groups,dc=example,dc=org",
    "group_name": "description",
    "user_group": "group"
  }
}
```

Diffix Explorer Configuration

The Diffix Explorer integration is optional. You can activate it by including the `explorer` parameter in your configuration. It specifies the Diffix Explorer instance Insights Air will connect to. The configuration looks like this:

```
"explorer": {
  "url": string
}
```

The single property `url` is the URL where Insights Air can find a running version of Diffix Explorer. Note that if your Diffix Explorer instance is running behind a reverse proxy that sends an HTTP redirect (status code 301 or equivalent) then the Diffix Explorer integration will fail. Please use the URL being redirect to instead. This includes if your reverse proxy redirects from HTTP to HTTPS. In the latter case, please explicitly include `https://` in the URL. For the Diffix Explorer integration to work properly, you will also need to configure the `site.endpoint_public_url` setting.

In the admin control panel you can choose which tables Diffix Explorer should analyze – by default none are.

Only tables meeting the following criteria can be analyzed:

- the table must contain a `user-id` column, and
- the table must have a least one other column that is selectable (i.e. not marked as unselectable) and not isolating

For more information about selectable and unselectable columns, please have a look at the section on configuring tables in data sources. For more information about isolating columns, read the section that describes what they are and how they can be configured.

Some of Diffix Explorer's behaviors can only be configured through the use of environment variables. For example, in order to limit the number of parallel queries issued, you can use the `Explorer__MaxConcurrentQueries` environment variable.

A full overview of the configuration variables can be found in the project documentation.

Insights Cloak configuration

The Insights Cloak configuration is used to provide the following information:

- URL where the Insights Air component can be reached
- Anonymization salt
- Data sources which can be queried - see Data source configuration

The general shape of `config.json` is:

```
{
  "air_site": string,
  "salt": string,
  "cloak_secret": string,
  "data_sources": string,
  "concurrency": integer,
  "lcf_buckets_aggregation_limit": integer,
  "max_parallel_queries": positive_integer,
  "allow_any_value_in_when_clauses": boolean,
  "allow_any_value_in_in_clauses": boolean,
  "connection_timeouts": {
    "idle": integer,
    "connect": integer,
    "request": integer
  },
  "analysis_queries": {
    "concurrency": positive_integer,
    "time_between_queries": integer,
    "minimum_memory_required": number
  }
}
```

The `air_site` parameter holds the URL where Insights Air component can be reached. It can be in the form of `"ws://air_host_name:port"` or `"wss://air_host_name:port"`, where `air_host_name` is the address of the machine where the Insights Air component is running. You should use the `ws` prefix if Insights Air is serving traffic over HTTP, while `wss` should be used for the HTTPS protocol.

The `salt` parameter is used for anonymization purposes. If your Aircloak Insights installation has multiple Insights Cloak instances you must make sure they use the same salt. Failing to do so has a negative impact on the quality of the anonymization. You can derive a strong `salt` parameter using a command such as:

```
cat /dev/urandom |
  LC_ALL=C tr -dc 'a-zA-Z0-9' |
  fold -w 64 |
  head -n 1
```

The `cloak_secret` setting is used to authenticate the Insights Cloak instance when connecting to Insights Air. It is required only if `cloak_secret` was configured in Insights Air (see Web site configuration), and in that case it needs to be set to the same value.

The `concurrency` field is optional and controls the amount of additional threads used for processing the selected data. The default setting is 0, which means a single thread processes the data coming in from the database server. For small data sets, this is usually sufficient, but for bigger data sets, this might turn out to be a bottleneck during query execution. By increasing this value (to

2 or 4 is recommended), additional threads will be used when ingesting the data, executing the query faster, but also consuming more memory. This setting can be overridden per data-source.

The `lcf_buckets_aggregation_limit` is optional and controls the maximum number of columns for which partial aggregation of low-count filtered rows is done. The default value is 3. This setting can be overridden per data-source. More details can be found in the **Low-count filtering** section.

The `max_parallel_queries` field is optional and controls the maximum number of queries that the cloak will run simultaneously. The default value is 10.

The `allow_any_value_in_when_clauses` field is optional and controls whether restricted `WHEN` clauses are allowed or not to use any value in anonymizing queries. The default value is false, which means only frequent values are permitted.

The `allow_any_value_in_in_clauses` field is optional and controls whether restricted `IN` clauses are allowed or not to use any value in anonymizing queries. The default value is false, which means only frequent values are permitted.

The `connection_timeouts` field is optional and it controls various database connection timeouts.

The `connection_timeouts.idle` field is optional and it determines how many seconds idle database connections are kept before they are closed. It needs to be an integer value between 1 and 86400 (1 day). If not set, a default timeout value of 60 seconds (1 minute) is used.

The `connection_timeouts.connect` field is optional and it determines how many seconds the Insights Cloak waits for a database connection to be established. It needs to be an integer value between 1 and 3600 (1 hour). If not set, a default timeout value of 5 seconds is used.

The `connection_timeouts.request` field is optional and it determines how many seconds the Insights Cloak waits for a database request to complete. It needs to be an integer value between 1 and 86400 (1 day). If not set, a default timeout value of 43200 seconds (12 hours) is used.

The `analysis_queries` field is optional and controls the rate of **column analysis** queries. If Insights Cloak exhausts too many system resources while analyzing columns, then you can specify these parameters to reduce overall load.

The `analysis_queries.concurrency` field is optional and controls the number of maximum concurrent analysis queries issued by Insights Cloak. It needs to be an integer value between 1 and 3. The default value is 3.

The `analysis_queries.time_between_queries` field is optional and it determines how many milliseconds to wait before issuing further analysis queries. This waiting period can be useful to allow Insights Cloak and databases to release resources before handling subsequent requests. It needs to be a non-negative integer value. If not set, a default value of 0 is used, meaning no delay between queries.

The `analysis_queries.minimum_memory_required` field is optional and controls the minimum relative system memory required to run analysis queries. If available memory falls below this threshold, then further analysis queries will be suspended until more memory becomes available. Requires a decimal value

between 0 and 1. For example, a value of 0.3 specifies that analysis queries will not run when available memory is under 30%. If not set, a default value of 0 is used, meaning analysis queries will run even when the system is low on memory.

Data source configuration

The `data_sources` parameter should give the path to subfolder within the folder where the Insights Cloak config is stored that contains the datasource configurations.

Each datasource configuration should be in JSON format and put in an individual file with the `.json` extension. The configuration takes the following form:

```
{
  "name": string,
  "driver": string,
  "parameters": {
    "hostname": string,
    "port": integer,
    "username": string,
    "database": string,
    "password": string
  },
  "concurrency": integer,
  "lcf_buckets_aggregation_limit": integer,
  "max_rare_negative_conditions": integer,
  "analyst_tables_enabled": boolean,
  "tables": tables,
  "load_comments": boolean
}
```

The `name` parameter is a string which will be used to identify the data source throughout the Insights Air interface and APIs.

The `driver` parameter can be one of the following: `postgresql`, `mysql`, `sqlserver`, `oracle`. The `parameters` json, then specifies the database connection parameters.

Some of these drivers use the ODBC protocol to talk to the database. These drivers are `sqlserver`, and `oracle`. Since they rely on ODBC, they accept some additional connection parameters:

- `encoding` which has possible values of "latin1", "unicode", "utf8", "utf16", "utf32", "utf16-big", "utf16-little", "utf32-big", "utf32-little".
- `odbc_parameters` - ODBC-specific parameters for the ODBC driver which is used to talk to the database.

These parameters are optional, and are only required for particular installations, where the default values do not suffice.

The `concurrency` field is optional and controls the amount of additional threads used for processing the selected data. If not present, the global setting is used.

The `lcf_buckets_aggregation_limit` field is optional and controls the maximum number of columns for which partial aggregation of low-count filtered rows is done. If not present, the global setting is used.

The `max_rare_negative_conditions` affects how many negative conditions containing rare values are allowed per anonymizing query. It defaults to a safe value of 0, which rejects all rare negative conditions, and should, under most circumstances, not be altered. Increasing the value above the default should only be done if it has been deemed safe.

The `analyst_tables_enabled` can be set to true to enable creation of analyst tables. By default, this parameter is set to false. See the **Analyst tables** section for more details.

The `load_comments` flag indicates whether database level comments should be loaded from configured tables during data source initialization. Defaults to `true`, meaning comments will be loaded.

Tables

The database tables that should be made available for querying are defined in the `tables` section of the cloak config. The value of the `tables` key is a JSON object that looks as follows:

```
"tables": {
  "table_name_1": {
    "db_name" | "query": string,
    "content_type": "personal" | "non-personal",
    "keys": [{"key_type_1": "column_name_1"}, ...],
    "exclude_columns": ["column1", "column2", ...],
    "unselectable_columns": ["column1", "column2", ...],
    "comments": {
      "table": "Comment on table 1.",
      "columns": {
        "column1": "Comment on column 1",
        "column2": "Comment on column 2"
      }
    }
  },
  "table_name_2": ...
}
```

Each `table_name_x` key specifies the name the table will be available under when querying the data source through Aircloak.

The `content_type` is an optional field which determines whether the data in the table is sensitive or not. It can have one of the following values: `personal` (default) and `non-personal`. Tables with data about individuals or entities whose anonymity should be preserved must be marked with the content type `personal`. If any such table is included in a query, the query will underlie the anonymization restrictions applied by Aircloak Insights and produce anonymized results. If the content type field is set to `non-personal`, the table will be classified as not containing data requiring anonymization. Queries over such tables are not subject to the anonymization restrictions. *No attempts will be made to anonymize the data they contain!*

The database table can be declared by either using `db_name` or as an SQL view using `query`. These options are mutually exclusive.

The `db_name` is the name of the table in the underlying database. In most situations you can use the same name (in which case the field can be omitted), but the distinction allows some special scenarios, such as exposing a table

under a simpler name, or exposing the same database table multiple times under different names. See the Referencing database tables section for details.

If the `query` field is present instead, a virtual table is created, similar to an SQL view. The provided query can gather data from multiple tables, filter what columns are exposed and pre-process, pre-filter or pre-aggregate the data. The supported SQL features are the same as in other Aircloak queries, but the anonymization-specific restrictions (like requiring a numerical range to have an upper and lower bound, for example) do not apply. An example configuration for a virtual table would look like this:

```
"table_name": {
  "query": "
    SELECT cast(t2.uid as integer), t2.age, t1.*
    FROM t1 INNER JOIN t2 ON t1.pk = t2.fk
    WHERE t2.age > 18
  ",
  "keys": [
    {"user_id": "id"}
  ]
}
```

The query can only select data from existing tables (or views) in the source database (it can not reference other virtual or projected tables from the configuration file). If the virtual table contains columns with duplicated names, only the first one is kept and the rest are dropped. Constant columns are also dropped from the table.

The `exclude_columns` is an optional parameter. It takes the form of an array and specifies columns to exclude from the underlying table. Excluded columns will not appear in the data source and cannot be referenced in any way from queries.

The `unselectable_columns` is an optional parameter for personal tables. It takes the form of an array and marks columns as unselectable. Unselectable columns can only be joined together, counted, and/or grouped by.

The `comments` field is optional and can be used to attach a description to tables and columns. Comments are visible in the Insights Air interface and are also returned from `SHOW` statements. Database-level comments are automatically retrieved and attached to tables.

Keys

Entities in a dataset, whether they be persons, transactions, or products, are usually identifiable by a single column value. This could be a user, patient or customer id in the case of a person, a transaction id for a transaction or a product id for a product. We call these types of identifiers *keys*. When you configure your tables you need to mark these columns as keys and declare what type of entity they describe.

When querying tables containing personal data it is a requirement that at least one of the tables queried contains a key of type `user_id`. Other tables that are part of the query need to be joined with a table containing a `user_id` key via the pre-configured key-relationships.

The following restrictions are currently in place when configuring keys:

- A column can have one key tag at the most.
- A `personal` table can have at most one `user_id` key.

- A `non-personal` table can't have any `user_id` keys.

An example configuration file for a database containing information about customers, accounts, transactions and bought products might look like this:

```
"tables": {
  "customers": {
    "keys": [
      {"user_id": "id"}
    ]
  },
  "accounts": {
    "keys": [
      {"user_id": "customer_id"},
      {"account_id_key": "id"}
    ]
  },
  "transactions": {
    "keys": [
      {"account_id_key": "account_id"},
      {"product_id_key": "product_id"}
    ]
  },
  "products": {
    "content_type": "non-personal",
    "keys": [
      {"product_id_key": "id"}
    ]
  }
}
```

while a valid query that accesses all tables might look like this:

```sql
SELECT
  customer.job,
  AVG(transaction.price)
FROM
  customer
  INNER JOIN accounts ON customer.id = accounts.customer_id
  INNER JOIN transactions ON accounts.id =
transactions.account_id
  INNER JOIN products ON transactions.product_id = products.id
WHERE products.type = 'car'
GROUP BY 1
```

Referencing database tables

Database tables are referenced when providing the `db_name` property. They can also be referenced in the query of virtual tables. The rules explained here are the same for both cases.

When referencing a database table, two characters are considered as special: the dot character (`.`) and the double quote character (`"`). If the name contains any of these characters, the name has to be quoted inside double quotes. Since the JSON string is already quoted inside double quotes, you need to use the `\"` syntax:

```
"db_name": "\"some.table\""
```

If the `"` character is a part of the table name, you need to quote the table name, and provide the double quote as `\"\"` inside the quoted name. For example, if the table name is `some"table`, you can specify it as:

```
"db_name": "\"some\"\"table\""
```

In some cases you might need to specify a fully qualified name, for example to provide a different database schema. In this case, you need to separate different parts with the dot character:

```
"db_name": "some_schema.some_table"
```

When quoting a multi-part identifier, you need to quote each part separately. For example, if the schema name is `some.schema`, and the table name is `some.table`, you can specify it as follows:

```
"db_name": "\"some.schema\".\"some.table\""
```

Also note that you only need to quote the part which contains special characters. In the following example, we quote the schema name (because it contains the dot character), but not the table name (because it doesn't contain any special characters).

```
"db_name": "\"some.schema\".some_table"
```

However, it's not an error if you quote each part, regardless of whether it requires quoting or not.

It's also worth mentioning that `db_name` is case sensitive, irrespective of whether it's quoted or not. Therefore, you should use the exact capitalization of the underlying database.

For example, let's say that the table is created with the following statement:

```
create table user_data(uid integer, ...)
```

In PostgreSQL, the table name will be lower-cased, while in Oracle, it will be upper-cased. Therefore, when providing `db_name`, you should specify `"user_data"` if the data source is a PostgreSQL database, or `"USER_DATA"` if the data source is an Oracle database.

Of course, if you explicitly used a non-default capitalization, then you need to use the same capitalization when specifying the `db_name`. For example, let's say that the following create statement was used to create a PostgreSQL table:

```
create table "UserData"(uid integer, ...)
```

In this case, you need to provide `"UserData"` as the `db_name` property.

Manually classifying isolating columns

Insights Cloak can automatically detect whether a column isolates users or not. For large database tables this check can be slow and resource-intensive. An administrator may choose to manually configure whether a given column isolates users or not, removing the need for automated classification.

How Insights Cloak handles classifying columns is configured for each table individually, using `auto_isolating_column_classification` (defaulting to true) and `isolating_columns` (empty by default). `isolating_columns` is a dictionary where each key is the name of a column, and the value of `true` or `false` indicates if the column should be considered as isolating users or not. The behaviour for columns not included in that dictionary is guided by `auto_isolating_column_classification` - if it's set to `true`, then Insights Cloak will try to compute if the column is isolating as normal, if to `false` then it will assume it's isolating.

Take this example:

```
{
  "tables": {
    "regular_table": {
      "db_name": "regular_table"
    },

    "auto_table": {
      "db_name": "auto_table",
      "auto_isolating_column_classification": true,
      "isolating_columns": {"telephone_number": true,
"first_name": false}
    },

    "manual_table": {
      "db_name": "manual_table",
      "auto_isolating_column_classification": false,
      "isolating_columns": {"first_name": false}
    }
  }
}
```

In this case Insights Cloak will automatically compute which columns in `regular_table` are isolating. For `auto_table` it will treat `telephone_number` as isolating, `first_name` as not isolating, and automatically handle all other columns. `manual_table` has the automatic isolating column detection turned off. All columns that have not been manually classified will therefore be treated as if they isolate users.

Warning The safest option is to treat a column as isolating. Manually classifying a column as not isolating may lead to privacy loss. It is safe to classify columns as not isolating only when sure that most values in that column appear for multiple users. Please contact support@aircloak.com if you need help classifying your data.

Column value shadow database

Insights Cloak automatically maintains a cache of column values that occur frequently. This allows certain anonymization practices to be relaxed when doing so does not cause harm. The creation of this cache requires a set of database queries to be run against the database that can become prohibitively expensive for large databases. You can turn of the creation and maintenance of

this shadow database when you either do not need the **extra capabilities** this feature offers, or operate in a resource constrained environment where running the required database queries is of concern.

The shadow database which is created by default can be toggled on and off on a per-table basis. If your data source configuration looked as follows and you wanted to disable the shadow database creation for the `very_large_table` table, you could include the `maintain_shadow_db` parameter and give it the value false:

```
{
  "tables": {
    "regular_table": {
      ...
    },

    "very_large_table": {
      ...
      "maintain_shadow_db": false
    },

    ...
  }
}
```

Analyst tables

Analyst tables make it possible for analysts to create additional tables in the database via the Aircloak user interface. The main purpose of these tables is to allow analysts to prepare a static snapshot of a potentially long running intermediate query. For example, consider the following query:

```
SELECT col_a, col_b
FROM (
  # possibly slow subquery
  SELECT ...
) subquery
```

If the subquery is taking a long time to complete, running different kinds of queries with variations in the top-level outer query can become very cumbersome. This is where analyst tables can help. They allow analysts to create a snapshot of the data returned by the inner query, and allow querying that snapshot instead.

Analyst tables are created in the air user interface. A table is described via a regular Aircloak `SELECT` query which defines the table structure and its content.

The query must be anonymizing, which means that it must select at least one user id column. Queries which lead to emulation (i.e. which can't be completely offloaded to the database) can't be used to create analyst tables.

When an analyst table is submitted for creation via the user interface, the cloak will create the corresponding table in the database and populate it. The table population is running asynchronously, and depending on the query, it might take a while. The table cannot be used for querying while it is being populated.

Once the table is populated, it can be used as any other table in Aircloak queries. The table can also be used from other analyst tables and views.

It's worth noting that each analyst tables is private, meaning that it can only be used by the analyst who created it.

Analyst tables should conceptually be treated as snapshots. A table won't update if data changes in the source tables. To update the content of the table, an analyst must open it for editing in the air user interface, and then press the "Update" button to trigger the table recreation. The table cannot be used for querying until the recreation has completed.

Since analyst tables can potentially cause additional load on the database server, both in terms of processing and disk-usage, they are by default disabled. To enable this feature, set the "analyst_tables_enabled" property in the data source configuration to `true`.

Currently, analyst tables are only supported on PostgreSQL and Oracle data sources.

If the air name or the datasource name is changed, duplicate copies of analyst tables might appear in the cloak database. This happens because the analyst table name depends on the air name and the datasource name. The database administrator can safely manually delete the obsolete analyst tables should such an event happen.

The administrator can use the table `__ac_analyst_tables_X` (where `X` is an integer) to list analyst tables. This table contains the list of all currently known analyst tables. The administrator can use the following columns to determine which tables are no longer needed:

- `air` - name of the air instance
- `data_source` - name of the data source where the table is created
- `analyst` - the numerical id of the table owner
- `name` - the table name, as seen in the air by its owner
- `db_name` - the name of the table in the database

If the administrator is certain that some analyst tables are no longer needed, for example if an air instance or some datasource have been renamed or decommissioned, they can drop these tables, and delete the corresponding entries from the `__ac_analyst_tables_X` table.

Tips and tricks

It is common to have multiple Insights Cloak instances sharing the same datasource definitions. Maintaining separate copies of the datasource definition for each Insights Cloak instance complicates maintenance, as you will have to update multiple copies of files if a datasource definition changes.

The recommended, and common, solution to this problem is to create a single copy of the datasource configuration files and symlink these into the configuration folders of the individual Insights Cloak instances.

Here is an example of how one can do this in pratice:

Let's consider a scenario where we have a shared folder where we store all datasource definitions. It is called `data_sources_available`. In the configuration folder of each Insights Cloak instance we create a folder called `data_sources_enabled`. For each datasource we want to enable for a given Insights Cloak instance we create a symlink from the `data_sources_enabled` folder to the datasource definition stored in the `data_sources_available` folder.

If we have two Insights Cloak instances, the folder structure would look like this:

```
$ tree configs

configs/
├── data_sources_available
│   ├── data_source1.json
│   └── data_source2.json
├── insights-cloak1
│   ├── config.json
│   └── data_sources_enabled
├── insights-cloak2
│   ├── config.json
│   └── data_sources_enabled
```

In order to have `insights-cloak1` serve `data_source1`, and `data_source2`, and `insights-cloak2` server `data_source1`, we would create the following symlinks:

```
$ cd config/insights-cloak1/data_sources_enabled/
$ ln -s ../../data_sources_available/data_source1.json
data_source1.json
$ ln -s ../../data_sources_available/data_source2.json
data_source2.json
$ cd ../../insights-cloak2/data_sources_enabled
$ ln -s ../../data_sources_available/data_source1.json
data_source1.json
```

The resulting file structure would then look as follows.

```
$ tree configs

configs/
├── data_sources_available
│   ├── data_source1.json
│   └── data_source2.json
├── insights-cloak1
│   ├── config.json
│   └── data_sources_enabled
│       └── data_source1.json ->
../../data_sources_available/data_source1.json
│       └── data_source2.json ->
../../data_sources_available/data_source2.json
├── insights-cloak2
│   ├── config.json
│   └── data_sources_enabled
│       └── data_source1.json ->
../../data_sources_available/data_source1.json
```

Enabling or disabling further datasources for individual Insights Cloak instances is then only a matter of adding or removing a symlink.

Hiding columns

In some cases, it might not be desirable that all of the columns in a table are exposed to analysts. Virtual tables can be used to hide one or more columns, by explicitly listing only the columns that are valid for querying.

For example, assuming we have the table `t` with columns `uid, x, y, z`, and we wish to hide column `y` from analysts, the following configuration file will do the trick:

```
"t": {
  "query": "SELECT uid, x, z FROM t",
  "keys": [
    {"user_id": "id"}
  ]
}
```

Alternatively, if the target table has a large number of columns and we don't want to list all of them, we can explicitly select constants with the hidden columns' names, then star-select everything else. This uses the fact that duplicated columns are eliminated by only keeping the first instance and then that constants are dropped from the list of exposed columns. For example:

```
"t": {
  "query": "SELECT 0 AS y, t.* FROM t",
  "keys": [
    {"user_id": "id"}
  ]
}
```

Running without Docker containers

If you're running the system without Docker containers, there are some additional things that need to be configured.

Insights Air shadow server

Insights Air requires access to two PostgreSQL database servers. One is used for storing query results, audit logs and user accounts. This is the PostgreSQL database server described in the Components of Aircloak Insights-chapter of these guides. The second PostgreSQL database server normally runs as part of the Insights Air docker container itself. When docker containers are not used this database server needs to be provided separately. It should be a PostgreSQL database server of version 9.6. The login credentials provided must be for a superuser or for a user having been given privileges to create and destroy databases (`CREATEDB` -role) on this database server. They can be configured in the `config.json` configuration file of the Air component under the `shadow_database` key.

```
...

"shadow_database": {
  "host": string,
  "port": integer,
  "ssl": boolean,
  "user": string,
  "password": string,
  "name": string
},

...
```

Here, the `"name"` parameter configures the name of the database to which the given user can connect. The database name is needed because a PostgreSQL connection can only be established to an existing database. For this purpose, you can use either the `postgres` database, or create a dedicated database. Make sure to grant `CONNECT` permission on the database to the user.

File permissions

The Aircloak Insights software is run inside a docker container under a user called `deployer`. The privileges of the software are limited by those of the `deployer` user. In order for Aircloak Insights to read the configuration files they need file permissions that allow everyone to read them.

Unix file permissions distinguish between the rights of the owner of a file, the members of a particular group, and everyone else. The `deployer` user belongs to the latter of the three, namely the everyone else category.

If you consider file permissions in their symbolic notation (like they are shown when running `ls -la` in the terminal), then the permissions need to end in `r--`. If you consider the privileges in their numeric notation, then the last digit needs to be at least a 4 (meaning it grants read privileges).

Below follows a set of file permissions that would work:

```
$ ls -la
-rwxr--r--  25 owner  group     800 Jan  1 00:01 ideal
-rwxrwxr--  25 owner  group     800 Jan  1 00:01 ideal
-rwxr-xr-x  25 owner  group     800 Jan  1 00:01 ok-but-too-
permissive
-rwxrw-rw-  25 owner  group     800 Jan  1 00:01 ok-but-too-
permissive
-rwxrwxrwx  25 owner  group     800 Jan  1 00:01 ok-but-too-
permissive
```

whereas the following set of file permissions *would not work* because they do not give the `deployer` user permission to read the file:

```
$ ls -la
-rwxr-----  25 owner  group     800 Jan  1 00:01 missing-read-
privileges
-rwxrw----  25 owner  group     800 Jan  1 00:01 missing-read-
privileges
-rwxr-x---  25 owner  group     800 Jan  1 00:01 missing-read-
privileges
-rwxrwx---  25 owner  group     800 Jan  1 00:01 missing-read-
privileges
```

In a unix shell you can add the required read permission with the following command: `chmod o+r file-name`. The `o` signifies the everyone else category (also known as "other") and the `+r` grants read permission.