Data Set Utility - Sensitive Data Discovery and on-the-fly Data Masking - User's Guide v25.5

Author: Philippe Debois (European Commission)

Table of Contents

1. Data Discovery

Sensitive data discovery refers to the process of identifying and locating sensitive or personally identifiable information (PII) within a dataset or a system. This is an important step in data privacy and security, as it helps organisations identify and protect sensitive information from unauthorised access or disclosure.

1.1. Data Discovery Techniques

Various techniques and approaches can be used for sensitive data discovery. Here are a few commonly employed methods:

- **Keyword-based search**: This technique involves searching for specific keywords or patterns that indicate the presence of sensitive data. For example, searching for terms like "social security number," "credit card number," or "date of birth" can help identify relevant data.
- **Regular expressions**: Regular expressions are patterns used to match specific text strings. They can be employed to identify structured patterns such as credit card numbers, phone numbers, or email addresses within a dataset.
- **Data classification**: This approach involves applying pre-defined classification rules or algorithms to automatically identify and categorise sensitive data based on its characteristics. Machine learning algorithms can be trained on labelled datasets to recognise patterns and classify data accordingly.
- **Statistical analysis**: Statistical techniques can be employed to analyse the distribution and properties of data fields within a dataset. Unusual patterns or outliers in the data can indicate the presence of sensitive information.
- **Data lineage and metadata analysis**: By examining data lineage and metadata, which provide information about the origin, transformations, and characteristics of data, organisations can trace the flow of data and identify potential sources of sensitive information.
- **Data scanning and profiling tools**: There are various software tools available that can scan and profile data to identify sensitive information automatically. These tools often combine multiple

techniques, including keyword search, regular expressions, and machine learning, to detect sensitive data.

This tool mainly supports the first two methods.

It's important to note that sensitive data discovery should be conducted in compliance with relevant data protection and privacy regulations. Organisations should follow best practices and implement appropriate security measures to ensure the confidentiality and integrity of the sensitive data they handle.

1.2. Tool Overview

The data set utility allows to discover sensitive data by searching for patterns in column names, column comments, and/or column values. Data can also be searched in pre-defined or user-defined look-up data sets (e.g., list of countries, cities, currencies, etc.).

The tool comes with a collection of pre-defined search patterns which are organised in categories. Here follows a (non-exhaustive) list of those provided by default:

Category	Name
Address	Address (sys)
Address	City (sys)
Address	Country code alpha-2 (sys)
Address	Country code alpha-3 (sys)
Address	Postal code (sys)
Address	State (sys)
Address	Street (sys)
Banking	American Express card number (sys)
Banking	Basic bank account number - BBAN (sys)
Banking	Credit card number (sys)
Banking	Currency code (sys)
Banking	Discover card number (sys)
Banking	Expiry date (sys)

Category	Name
Banking	Mastercard card number (sys)
Banking	Visa card number (sys)
Internet	E-mail address (sys)
Internet	IP v4 address (sys)
Internet	IP v6 address (sys)
Internet	URL (sys)
Internet	Username (sys)
Large text	Large text (sys)
Large text	Very large text (sys)
National identifier	Belgian National Registration Number - NRN (sys)
Personal	Gender - alpha (sys)
Personal	Gender - any type (sys)
Personal	Identity card number (sys)
Personal	Person birth date (sys)
Personal	Person death date (sys)
Personal	Person family name (sys)
Personal	Person full name (sys)
Personal	Person given name (sys)
Personal	Phone number (sys)
Vehicle	Registration(sys)
Vehicle	Vehicle id (sys)

Table 1 – pre-defined search patterns

The “(sys)” mentioned in the name allows to make the distinction between pre-defined patterns (that in principle should not be modified) and user-created ones. Patterns have been designed with the assumption that your database object names and comments are in English. Should it not be the case, you will need to adapt them to your specific language or create your owns.

1.3. Pattern properties

Search patterns are stored in the `DS_PATTERNS` table and have the following properties:

- A name which uniquely identifies the pattern
- A category to logically group patterns together (e.g., personal, banking, etc.)
- A column data type (+ length, precision, and scale) with which columns must be compatible
- A column name pattern i.e., the regular expression that column names must match
- A column comment pattern i.e., the regular expression that column comments must match
- A column data pattern i.e., the regular expression that column values must match
- A look-up data set name i.e., the name of a data set to which column values must belong
- A minimum hit percentage that must be reached when searching in the data
- A minimum hit count that must be reached when searching in the data
- A logical operator (`OR` or `AND`) to be applied between search patterns
- Some remarks or comments about the pattern
- A system flag which indicates whether the pattern is pre-defined (vs user-created)
- A disabled flag which allows end-user to deactivate a pattern
- The mask that will be applied by default to the data matching the pattern (mask type and parameters)

1.4. Search criteria

A search pattern is a combination of up to 5 search criteria:

- Column data type (+ length, precision, and scale)
- Column name regular expression
- Column comment regular expression
- Column value regular expression
- Column value data set lookup

A least one search criteria must be specified.

1.4.1. Data type

When a pattern data type is defined, a table column must have a data type that is compatible. The data type is an exclusion criterion when not fulfilled. The following data types can be associated with a pattern:

`CHAR`, `CLOB`, `NUMBER`, `DATE` and `TIMESTAMP`.

Example: in well-designed schemas, dates like birth dates or death dates are stored in `DATE` columns; it is therefore useless to search for these dates in non-`DATE` columns.

Optionally, a minimum length can be specified for `CHAR` and a minimum scale and/or precision for `NUMBER`. Syntaxes are `CHAR(n)` where `n` is the minimum length and `NUMBER(x,y)` where `x` and `y` are respectively the minimum scale and precision. When specified, the column data type must meet minimum length, scale and/or precision requirements to be retained.

Example: since internet addresses are composed of four numbers separated with dots (e.g., “0.0.0.0”), their minimum length is 7; it is therefore useless to search them in character columns whose length is lower than 7; for this use case, `CHAR(7)` is be specified as the pattern data type.

The following data types are considered as equivalent for the compatibility check:

- **CHAR**, VARCHAR2, NCHAR and NVARCHAR
- **CLOB** and NCLOB
- **NUMBER**, INTEGER, BINARY_FLOAT and BINARY_DOUBLE
- **DATE**
- **TIMESTAMP**, TIMESTAMP(n) WITH (or without) TIME ZONE

Only the data types in **bold** can be specified at the level of the pattern. Column data types are converted according to the above equivalence table before being compared to the pattern data types.

1.4.2. Regular expressions

Column name, comments and values can be searched using **Oracle regular expressions** (see Table 17 – metacharacters supported in Oracle regular expressions – in annex and Oracle documentation for a full description). Note that the Oracle specific implementation has some limitations compared to other implementations (e.g., look ahead is not implemented in Oracle). A search criterion is considered as fulfilled if the column name, column comment or column value matches the given regular expression.

Pattern matching is always **case insensitive** i.e., lower/upper case differences are ignored. In practice, the `REGEXP_LIKE` function is used with the “i” option (insensitive) passed as parameter.

Pattern matching is also **accent insensitive** i.e., accentuated characters in both column values and search patterns are converted to their ASCII equivalent characters before comparison (as an example, “é” is converted to “e”, “à” to “a”, etc.). In practice, the `unaccentuate_string()` function of the `ds_masker_krn` package is applied to both column value and regular expression.

For column name patterns, consider searching for abbreviations in addition to full words. As an example, when searching for currency codes, consider search for “currency”, “ccy” and “cur” as well as “code” and “cd”. To avoid false positive, patterns should also take care of word separators (the underscore character is the most common, but it depends on your naming conventions).

For column comment patterns, searching for entire words is good enough as the use of abbreviations is less frequent. You should also consider that, in comments, words are separated by a space rather than an underscore.

1.4.3. Lookup data sets

Besides the possibility to search for patterns in your data using regular expressions, you can also specify the name of a look-up data set in which data will be searched. This is particularly useful when a regular expression cannot be used without generating many false positives.

As an example, the regular expression `^[A-Z]{3}$` (which matches exactly 3 alphabetic uppercase characters) cannot be used to find currency codes as it also matches country codes which have exactly the same format.

These look-up data are stored in the `DS_DATA_SETS` table and have two possible types: CSV or SQL.

CSV data sets are stored as CLOB and follow a CSV-like format which is entirely configurable. You can specify whether the CSV has header lines (for column names and column data types) or not, what are the field and line delimiters, etc. CSV data sets can easily be created by simply copying data from an Excel spreadsheet and pasting it in a script.

SQL Data sets are just SQL queries that are can be executed to retrieve data from your own schema. You can consider them as equivalent to views.

The tool comes with several pre-defined CSV data sets (e.g., countries, major EU cities, currencies, common given names, common family names, etc.) from which you can search. They can also be used for generating random values during data masking. You can also create your own.

Here is the (non-exhaustive) list of pre-defined data sets:

Name	Purpose
`EU6_FAMILY_NAMES_217`	Most common family name in EU6 (Germany, France, Italy, Netherlands, Belgium, Luxembourg)
`EU_MAJOR_CITIES_590`	Major European cities

Name	Purpose
`INT_COUNTRIES_250`	Countries (worldwide)
`INT_CURRENCIES_170`	Currencies (worldwide)
`INT_GIVEN_NAMES_250`	Most common given names (worldwide)

Table 2 – pre-defined data sets

Data set names are prefixed with their scope (country, continent, region, etc.) and suffixed with their number of records.

As a data set usually contains several columns (e.g., the “countries” data set contains the country code and the country name), you must specify which one to use.

The syntax is `DATA-SET-NAME.COLUMN-NAME` where `DATA-SET-NAME` is the name of a CSV or SQL data set and `COLUMN-NAME` is one of its columns. As an example, the pre-defined pattern “Currency code” is associated with the `INT_CURRENCIES_170.CCY_CODE` data set column.

Data set columns that are referenced by patterns are cached into memory to speed-up their search. Consequently, it is recommended to avoid creating very large look-up data sets.

1.4.4. Data related criteria

For data related criteria (based on regular expressions or data set look-up), a minimum hit percentage and a minimum hit count can be specified. The hit percentage is calculated as the number of matching values divided by the number non-NULL values contained in the selected sample. The minimum hit count allows to discard false positive matches, when very few values are matching by accident (e.g., some first names like “Bernard” that are also family names). A value is considered as matching if it matches the regular expression or if it is found in the look-up data set. Columns whose data matching hit percentage is below the minimum are not considered as matching.

The minimum hit percentage must carefully be chosen. Setting it too low may lead to wrong matches. Setting it too high may lead to missed matches. As an example, when an exhaustive look-up data set is used (e.g., a list of country codes), a high hit percentage (e.g., 90%) is recommended because it is very unlikely to encounter other values. When a data set is not exhaustive (e.g., a list of most common given names or family names), a low hit percentage (e.g., 10%) is recommended.

1.4.5. Combining criteria

When several criteria are specified, they are logically combined according to the following rules:

- The AND logical operator is applied between the data type criteria and all others (as already mentioned, data type is an exclusion criterion when not fulfilled).
- The OR logical operator is applied between column name and column comment criteria (they are considered on an equal footing).
- The OR logical operator is applied between column value pattern and column value in look-up data set criteria (they are considered on an equal footing).
- The logical operator defined at the level of the pattern (OR/AND) is applied between column name/comment patterns on one hand, and column value pattern/data set lookup on the other hand; when not specified, the default logical operator is OR.

This can be summarised by the following table:

Data type	AND	Name pattern OR Comment pattern	AND or OR (pattern defined)	Data pattern OR Data set lookup

Table 3 – logical combination of criteria

1.4.6. Multiple matches

When a table column matches several patterns, the best matching pattern is retained according to the following priority rules:

- A pattern matching both column name and comment has precedence over a pattern matching only name or comment.
- A pattern matching column values or data has precedence over a pattern matching only a column name and/or comment.
- When several patterns match data, the one having the highest hit percentage wins.

1.5. Data discovery invocation

The sensitive data discovery can be launched by invoking the `discover_sensitive_data()` procedure of the `ds_utility_krn` package.

The discovery process can be launched for the whole schema or just for the tables involved in a specific data set. For an incremental discovery, it can also be limited to some tables or some columns by passing their name in parameter (Oracle wildcards are allowed). Can also be specified the rows sample size (the number of rows selected from each table to look at their data – 200 records by default), the values sample size (the number of matching values to be shown in the report), and the minimum hit count (2 by default) and percentage (10 by default) when not defined at the pattern level. Transaction can be committed on request, at the end of the process.

Here is a description of all parameters:

- `p_set_id` – restricts the discovery to the tables of a data set, NULL by default meaning all
- `p_full_schema` – launch a discovery for the full schema (Y/N), N by default
- `p_table_name` – table name, NULL by default meaning all tables (wildcards allowed)
- `p_column_name` – column name, NULL by default meaning all columns (wildcards)
- `p_rows_sample_size` – number of rows selected in each table, 200 by default
- `p_col_data_min_pct` – minimum hit percentage, 10% by default
- `p_col_data_min_cnt` – minimum hit count, 2 by default
- `p_overwrite` – overwrite results of a previous run, FALSE by default
- `p_commit` – issues a commit at the end of the discovery, FALSE by default
- `p_values_sample_size` – number of sample data shown in the report, 10 by default

1.6. Data discovery report

A report showing the result of the pattern matching exercise can be obtained.

It is particularly useful to understand why a specific pattern has been discarded or, in case of multiple matches, why a specific pattern has been preferred over another. It also shows a sample of matching values that may help in detecting wrong matches.

For each table column, the following can be shown:

- `Matching on name regexp` : column name is matching the regular expression
- `Matching on comm regexp` : column comment is matching the regular expression
- `Matching on data regexp` : column value is matching the data regular expression
- `Matching on data set` : column value has been found in the look-up data set
- The number of matching values compared to non-NULL values (e.g., 192/200 rows)
- The hits percentage compared to the minimum (e.g., 1% vs 10%)
- `DISCARDED on data type` : the patterns and column data types are not compatible
- `DISCARDED on min pct` : the minimum hit percentage is not reached
- `DISCARDED on min cnt` : the minimum hit count is not reached
- `NOT BEST` : the pattern is matching but not the best (see priorities / precedence)
- `RETAINED` : the pattern is matching and retained as the best
- `Data regexp sample` : the list of matching column values (regexp or look-up data set)

Here is an example of such a report:

```
Info:  ANO_COUNTRIES.CNT_CD: Pattern "Currency code (sys)" DISCARDED on: min pct (2% vs 90
Info:  ANO_COUNTRIES.CNT_CD: Pattern "Person given name (sys)" DISCARDED on: min pct (1% v
Info:  ANO_COUNTRIES.CNT_CD: Pattern "Country code alpha-2 (sys)" NOT BEST; matching on: n
Info: *ANO_COUNTRIES.CNT_CD: Pattern "Country code alpha-3 (sys)" RETAINED; matching on: n
Info:  ANO_COUNTRIES.DESCR_NAT_ENG: Pattern "Country code alpha-3 (sys)" DISCARDED on: min
Info:  ANO_COUNTRIES.DESCR_NAT_FRA: Pattern "Country code alpha-3 (sys)" DISCARDED on: min
Info:  ANO_COUNTRIES.DESCR_COUNTRY_ENG: Pattern "Person given name (sys)" DISCARDED on: mi
Info: *ANO_COUNTRIES.ISO_CNT_CD: Pattern "Country code alpha-2 (sys)" RETAINED; matching o
Info:  ANO_COUNTRIES.ISO_CNT_CD: Pattern "Country code alpha-3 (sys)" DISCARDED on: data t
Info:  ANO_COUNTRIES.CAPITAL: Pattern "Gender - alpha (sys)" DISCARDED on: min pct (1% vs 
Info:  ANO_COUNTRIES.CAPITAL: Pattern "Person family name (sys)" NOT BEST; matching on: da
Info: *ANO_COUNTRIES.CAPITAL: Pattern "City (sys)" RETAINED; matching on: data set (18/182
```

By default, only error messages are sent to `DBMS_OUTPUT`. To get the execution report, you need to enable the display of information messages (in addition to error messages) by invoking the following procedure:

```
ds_utility_krn.set_message_filter(‘EI’);
```

To be noted that, if a sample value is encountered several times, it is reported only once. In other words, only the distinct sample values are reported.

1.7. Data discovery results

As a result of the sensitive data discovery process, a mask is created in the `DS_MASKS` table for each column that was judged sensitive, based on the search patterns. The default mask type and its related parameters defined at the level of the pattern are copied over to the created mask.

The following variables can be used in the default mask parameters:

- `:column_name` – name of the column
- `:col_data_length` – data length of the column
- `:col_data_precision` – precision of the (number) column
- `:col_data_scale` – scale of the (number) column

They are substituted with their actual value when the mask is created.

As an example, the following SQL masking expression defined at the level of a pattern:

```
obfuscate_string(p_string=>:column_name,p_seed=>:column_name)
```

will become the following SQL expression in the generated mask:

```
obfuscate_string(p_string=>street,p_seed=>street)
```

when it is applied to a column named `street`.

It is worth mentioning that the data discovery process is not capable of finding relationships between fields, reason why the generated mask must carefully be reviewed and updated as necessary. As an example, first name, last name and full person's name will by default be masked independently while they are inter-dependent (the full person's name is a concatenation of first and last name). Part of a Belgian national registration number depends on the person's gender, but no relationship between these 2 fields will ever be reflected in the default masks generated by the discovery process.

2. Data Masking

2.1. Overview

Data masking, also known as data obfuscation or data anonymisation, is a technique used to protect sensitive or confidential information by replacing real data with fictional or modified data while preserving the data's format and usability for non-sensitive purposes. The primary goal of data masking is to minimise the risk of unauthorised access to sensitive data while still allowing organisations to use the data for various non-production purposes such as software testing, analytics, or outsourcing.

2.1.1. Data masking techniques

There are several common techniques used for data masking:

- **Substitution**: this technique involves replacing sensitive data with fictional or randomly generated data. For example, real names may be replaced with random names, social security numbers with fake numbers, or credit card details with randomly generated numbers.
- **Shuffling**: in this technique, the values of sensitive data are shuffled or randomly rearranged while maintaining the integrity of the data set. For example, the order of birth dates or addresses in a dataset may be shuffled to obscure the original values.
- **Masking**: masking involves partially hiding or obscuring sensitive data by replacing certain characters or digits with masking characters. For instance, masking a credit card number might

involve replacing all but the last few digits with "X" or "*", preserving the last few digits for identification purposes.

- **Perturbation**: this technique involves adding random noise or slight modifications to the original data. For numerical data, such as salaries or ages, a random value can be added or subtracted within a defined range to obfuscate the real values.
- **Encryption**: data encryption involves transforming the original data into an unreadable format using cryptographic algorithms. Encryption typically requires decryption keys to retrieve the original data, making it a more secure technique for protecting sensitive information.
- **tokenisation**: tokenisation involves replacing sensitive data with randomly generated tokens or references. The original data is stored in a secure tokenisation server, and the tokens are used for non-sensitive operations. The tokens cannot be reversed or used to retrieve the original data without proper authorisation.

All above techniques are supported by this tool. They can be applied individually or in combination, depending on the specific requirements of the data masking process and the level of protection needed for the sensitive information. *This tool allows to combine all masking techniques implemented via a SQL function (substitution, masking, encryption, and perturbation). However, Shuffling, sequences, tokenisation, and SQL expressions cannot be combined.*

2.1.2. Data Masking Methods

There are also different approaches or methods used for implementing data masking:

- **On-the-fly data masking**, also known as real-time data masking, involves applying data masking techniques in real-time as the data is accessed or retrieved from a database or data source. The sensitive data is dynamically transformed or obfuscated on the fly, just before it is presented to the user or application. This approach ensures that sensitive data always remains protected, regardless of the data's usage or storage location. On-the-fly data masking is typically implemented using database proxies, virtualisation techniques, or specialised middleware.
- **Static data masking**, also referred to as offline data masking or pre-production data masking, involves applying data masking techniques to a copy of the production database or dataset before it is used for non-production purposes such as development, testing, or analytics. The sensitive data is masked or obfuscated in advance, and the masked dataset is then distributed to various non-production environments. Static data masking ensures that sensitive information is replaced with fictional or modified data that cannot be traced back to the original values. This approach allows organisations to use realistic but non-sensitive data for various purposes while minimising the risk of data breaches or unauthorised access.
- **Dynamic data masking** is a data protection technique that provides an additional layer of security by selectively revealing sensitive data based on user roles or privileges. With dynamic data masking,

> the original sensitive data remains intact in the database, but its visibility is restricted or altered for specific users or applications. For example, sensitive data such as credit card numbers or social security numbers might be masked or partially masked to display only a subset of the digits, while authorised users or applications with the necessary permissions can view the complete data. Dynamic data masking helps to ensure that sensitive information is only accessible to authorised individuals or systems while maintaining the original data's integrity for authorised use cases.

This tool performs sub-setting and data masking on-the-fly, during data transportation. Depending on the chosen transportation method, data are masked while being copied via database links, when SQL scripts are generated or when data are extracted as XML. Static data masking can also be achieved in a roundabout way by applying update operations to the source schema itself. Furthermore, masked data can be previewed before transportation via the creation of ad-hoc views. Compared to static data masking, which is the approach followed by most tools, on-the-fly data masking has the advantage that it does not require the creation of a copy of the production database, which can be a heavy and lengthy process.

2.2. Mask properties

Sensitive columns as well as techniques used to mask them are defined in the `DS_MASKS` table. These are the results of the sensitive data discovery process, but they can also be created or updated manually via the provided APIs.

Here are the mask properties:

- `MSK_ID` : internal identifier
- `TABLE_NAME` : Table name
- `COLUMN_NAME` : Column name
- `SENSITIVE_FLAG` : Sensitive column (Y/N)?
- `DISABLED_FLAG` : DISABLED (Y/N)?
- `LOCKED_FLAG` : Locked (Y/N)?
- `MSK_TYPE` : Mask type (`SQL` , `SHUFFLE` , `INHERIT` , `SEQUENCE` , `TOKENIZE`)
- `SHUFFLE_GROUP` : Shuffle group
- `PARTITION_BITMAP` : Partition bitmap
- `PARAMS` : Mask parameters (e.g., the SQL statement)
- `PAT_CAT` : Category of the source pattern
- `PAT_NAME` : Name of the source pattern
- `REMARKS` : Mask remarks
- `VALUES_SAMPLE` : Sample of values

The `SENSITIVE_FLAG` is set to Y by the sensitive data discovery process when the column matches one of the active patterns of the library (i.e., all those that are not disabled). In case of false positive i.e., when a column was marked as sensitive during the discovery while it is not, you can change this `SENSITIVE_FLAG` to N to prevent it from being masked. Any column which is declared sensitive will be mask according to the masking type and parameters.

When created or modified manually, mask records must be locked by setting their `LOCKED_FLAG` to Y to avoid your encoding or changes from being overwritten upon next sensitive data discovery run.

Masks can be temporarily (or indefinitely) disabled by setting their `DISABLED_FLAG` to Y.

The `PAT_CAT` and `PAT_NAME` fields give the category and name of the best matching pattern found during the discovery process, while remarks explain which matching criteria was fulfilled. `VALUES_SAMPLE` give a few examples of matching data.

The following mask types are supported:

- `SQL` – the SQL expression is defined in `params`.
- `SHUFFLE` – shuffles columns using `shuffle_group` and `partition_bitmap` (see later).
- `SEQUENCE` – generates a sequential number based on an in-memory sequence (for pk/uk).
- `INHERIT` – inherits the value from a foreign key (cannot be defined by end-user).
- `TOKENIZE` – value is replaced with a randomly generated token.

`SEQUENCE` has no effect when defined on a column that is not part of a simple (i.e., non-composite) numeric primary or unique key.

`INHERIT` is automatically set for columns that inherit their value from a foreign key that references a masked primary or unique key column.

2.3. SQL masking

When the masking type is SQL, the column value will be replaced with the result of the SQL expression given in params. Any SQL expression that proves to be valid when selected from the table is allowed. It can involve any native SQL function or operator supported by Oracle (e.g., `UPPER`, `LOWER`, `SUBSTR`, `NVL`, `CASE`, …). Any functions of the standard Oracle DBMS packages can also be invoked (e.g., `DBMS_RANDOM.VALUE`). The tool also comes with pre-defined masking functions that can be used to mask your data (see next section).

An expression can reference any column of the table with 2 different syntaxes:

- When prefixed with a colon (e.g., `:given_name`), it refers to the column value after masking.

- When not prefixed with a colon (e.g., `given_name`), it refers to the column value before masking.

Examples:

- The SQL expression `ds_masker_krn.obfuscate_string(given_name)` associated with the `given_name` column will pass the actual given name (so before masking) to the obfuscation function.
- The SQL expression `:given_name||’ ’||:usual_name` associated with the `full_name` column will concatenate the given and usual names resulting from the masking (so after masking).
- Warning: make sure not to introduce cycles in your column references (e.g., column A references B, B references C, and C references A). When such a loop is detected at runtime, an exception is raised.

Warning: the validity of the SQL expression is currently not checked at encoding time. An exception will be raised at runtime if the SQL expression proves not to be valid. For that reason, it is strongly recommended to carefully test your SQL expression before encoding it.

2.4. Masking functions library

Several masking functions, supporting the various techniques previously described, are provided in the `DS_MASKER_KRN` package.

2.4.1. Function types

Here are the various types of function pre-defined in the `DN_MASKER_KRN` package:

Function prefix	Usage
`random_xxx`	Generate a random value within some size limits (length, scale, and/or precision), independently of the actual value. Applicable to number, integer, string, name, date, time, credit card number, BBAN, IBAN, data set, etc..
`obfuscate_xxx`	Obfuscate a given value by replacing it (in part or in full) with a random value, while preserving format and validity (when applicable). Applicable to date, string, credit card number, BBAN, IBAN, etc.
`mask_xxx`	Hide a value (in part of in full) by replacing some characters with X (or any other masking character) according to a pattern or to some rules. Applicable to string, credit card number, BBAN, IBAN, etc.

Function prefix	Usage
`encrypt_xxx`	Encrypt a value with a private key in such a way that it can later be decrypted. Format and validity (if applicable) are preserved. Applicable to string, number, BBAN, IBAN credit card number, etc.
`decrypt_xxx`	Decrypt a previously encrypted value (using the same private key as during encryption). Applicable to string, number, BBAN, IBAN credit card number, etc.
`is_valid_xxx`	Check the validity of a given value (e.g., by verifying that its checksum or check digit is valid). Applicable to credit card number, BBAN, IBAN, etc.

Table 4 –function types library

See Table 19 – methods of the DS_MASKER_KRN package – for an overview of the provided pre-defined functions.

To preserve the validity, the checksum or the check digit is recomputed.

When passing parameters to these functions, it is recommended to use named parameters (e.g., `p_param=>value`) rather than positional parameters. This syntax is indeed more resilient to any future potential change in the order of parameters.

Before encrypting and decrypting, the encryption key must be globally defined by calling the `set_encryption_key()` function. As this private key is only held in memory, make sure to store it in a safe location to allow subsequent decryption.

2.4.2. Deterministic functions

All randomisation and obfuscation functions accept an optional `p_seed` parameter which, when passed, makes them deterministic. The seed value can be an integer or a string of 2000 characters at maximum.

A deterministic function is a mathematical function or algorithm that always produces the same output for a given input, regardless of the number of times it is executed or the context in which it is executed. In other words, if the input to a deterministic function remains unchanged, the output will be consistent and predictable. Deterministic functions are particularly valuable in scenarios where the same output must be derived consistently from the same input, allowing for the reliable comparison of results.

You are free to decide whether you want random values to be generated consistently or not between two different data set extractions. However, as data masking is performed on-the-fly, a seed must always be passed to masking functions in some specific cases:

- When a column is involved in the computation of other columns (e.g., given name and family name are used to derive full person’s name); as the SQL expression and thereby the function call will be reused from one column to another, it must produce the exact same value.
- When you want to select at random a coherent pair (or n-uples) of values from a data set (e.g., a postal code and city name, which must be kept synchronised); the index generated at random to select a record from the data set must be exactly the same when retrieving each column.

In the above two cases, the exact same seed value must be passed to mask the columns that are combined (given name and family name) or that must be kept in sync (postal code and city name). The `ROWID` of the record is a good candidate for the seed value.

Examples:

```
dbms_masker_krn.obfuscate_string(p_string=>given_name, p_seed=>ROWID);
dbms_masker_krn.obfuscate_string(p_string=>family_name, p_seed=>ROWID);
dbms_masker_krn.random_value_from_data_set (
   p_set_name=>’EU_MAJOR_CITIES_590.POSTAL_CD’, p_seed=>ROWID
);
dbms_masker_krn.random_value_from_data_set (
   p_set_name=>’EU_MAJOR_CITIES_590.CTY_NAME’, p_seed=>ROWID)
;
```

In all other cases, when columns are masked independently from each other, any seed value can be used. The actual value of the column is a good candidate although the `ROWID` can also be used.

Example:

```
dbms_masker_krn.obfuscate_string(p_string=>username, p_seed=>username);
```

2.5. Shuffling

In the context of data masking, shuffling refers to a specific technique used to obfuscate or anonymise data by randomly rearranging the values within a dataset or a specific column while preserving the overall distribution and statistical properties of the data.

The purpose of shuffling in data masking is to break any direct correlation between the original values and their masked counterparts. By shuffling the values, it becomes difficult to link individual records or identify the original data points, thereby enhancing data privacy and protecting sensitive information.

As an example, you may want to mask person’s addresses by shuffling postal codes and cities i.e., exchanging them between persons. As it is desirable to keep the consistency between postal codes and cities, these two columns must be shuffled together, forming what is called a shuffling group.

The tool allows up to 3 shuffling group per table, although this limit could easily be lifted if necessary. Of course, a shuffled column can only be shuffled once so it cannot belong to more than one shuffling group. Shuffling groups are numbered from 1 to 3. The `shuffling_group` column specifies which shuffling group a column is part of.

A partitioning of the data can also be performed. In the context of shuffling, a partition refers to dividing a dataset or a collection of elements into distinct subsets or groups before applying the shuffling process. Each partition represents a subset of the data that will be shuffled independently of the other partitions. The purpose of partitioning is to ensure that the shuffling process maintains the integrity and relationships within the subsets while still introducing randomness and unpredictability.

As an example, you may want to shuffle person’s given names while keeping the consistency with person’s genders (i.e., male given names should remain associated with males, and idem for females). By partitioning on gender, male and female given names will be shuffled independently, thereby keeping the coherence vis-à-vis the gender. Another example would be to shuffle postal codes and cities partitioned by country, again to keep a certain coherence.

On the contrary to shuffled columns, a column can be involved in more than one partition. The `partition_bitmap` column specifies the partitions in which a column is involved. Bit n relates to shuffling group n+1 (so bit 0 relates to shuffling group 1, bit 1 to group 2 and bit 2 to group 3). The value associated with shuffling group n is therefore `POWER(2,n-1)`. As an example, for a column involved in the partitions of shuffling groups 1 and 2, the bitmap is 011, resulting in a value of 1+2=3.

To be noted that primary or unique key columns should not be used in partitioning. Doing so would just annihilate the shuffling.

2.6. tokenisation

tokenisation is a security technique that involves replacing sensitive information, such as credit card numbers or personal identifiers, with unique tokens or placeholder values while preserving the data's format and length. This process allows organisations to protect sensitive data from unauthorised access or exposure while maintaining the integrity and usability of the data for legitimate purposes. tokenisation ensures that even if a breach occurs, the stolen data remains incomprehensible and unusable to malicious actors, reducing the risk of data breaches and enhancing data security.

2.6.1. tokenisation techniques

Data tokenisation can be implemented in various ways depending on your specific needs and requirements. As this tool generates tokens using a SQL expression that can invoke any function of the masking library, almost all common tokenisation techniques are supported:

- **Format-Preserving tokenisation**: This approach replaces sensitive data with tokens that preserve the format and structure of the original data, ensuring that the tokenised data can be used in the same way as the original. For example, a credit card number. Supported by this tool via `obfuscate_xxx()` , `mask_xxx()` , or `encrypt_xxx()` functions.
- **Secure Hash tokenisation**: In this method, sensitive data is transformed into a fixed-length hash value using a cryptographic hash function. The hash value serves as the token, and it cannot be reversed to retrieve the original data. Supported by this tool via standard Oracle functions such as: `ora_hash()` , `standard_hash()` , or `dbms_crypto.hash()` .
- **Random tokenisation**: This approach generates completely random tokens that have no relationship to the original data. Random tokens provide a high level of security but may not be suitable for all use cases because they do not preserve any characteristics of the original data. Supported by this tool via `random_xxx()` functions.
- **tokenisation with a Token Vault**: A token vault is a secure storage system that associates tokens with their corresponding sensitive data in a database. This allows organisations to map tokens back to their original values, when necessary, typically through access controls and authentication. Not supported by this tool but original values can be encrypted in the mapping table.
- **Deterministic tokenisation**: In deterministic tokenisation, the same input data will always produce the same token. This method is useful when consistency is required, such as in database joins or search operations. Supported by this tool when a seed value is passed during masking and under the condition that any used data set remains unchanged.
- **Dynamic tokenisation**: Dynamic tokenisation generates different tokens for the same input data each time, enhancing security by making it more challenging to correlate tokens with their original values. Not supported by this tool (a value is always mapped to the same token by design).
- **Format-Preserving Encryption**: This technique combines encryption and tokenisation by encrypting the sensitive data but maintaining its format, providing an additional layer of security. Supported by this tool via `encrypt_xxx()` functions.

2.6.2. Relationship cardinality

The cardinality of the relationship between a token and the original value in a tokenisation system can vary depending on the implementation and requirements. Cardinality refers to the number of unique values in a set or the degree of uniqueness in a relationship. In the context of tokenisation, the cardinality of the relationship between a token and the original value can be categorised into two main types:

- **One-to-One (1:1) Cardinality**: In a one-to-one tokenisation system, each original value is associated with a unique token, and each token corresponds to a specific original value. This means that there is a direct and unique mapping between the original data and its token. One-to-one tokenisation is often used when you need to maintain a perfect and reversible mapping between the tokenised data and the original data, such as in certain payment processing systems.
- **Many-to-One (M:1) Cardinality**: In a many-to-one tokenisation system, multiple original values can map to the same token. This means that different instances of the same sensitive data can produce the same token. Many-to-one tokenisation is commonly used in scenarios where preserving the uniqueness of each original value is not a requirement, and it can help improve security by reducing the predictability of tokens.

This tool implements both cardinalities. By default, duplicate tokens can be generated (M:1) but unique tokens (1:1) can also be requested via the option `enforce_uniqueness=true`.

2.6.3. Security

This implementation does **not** make use of a token vault or server (a secure database that stores the mapping between tokens and their corresponding original values). Using a token vault provides an additional layer of security by centralising token management, access controls, and auditing. It ensures that even if an attacker gains access to the tokenised data, they cannot easily map tokens back to the original values without proper authorisation.

In this implementation, the mapping is stored in a table that is internal to the tool (`DS_TOKENS`). By default, original values are stored encrypted while tokens are stored as clear text. Original values are decrypted on-the-fly when the mapping needs to be performed (e.g., in views created for data set preview or during data extraction and transportation). The encryption key chosen by the end-user is stored in memory for the duration of the session and is not persisted. It is end-user’s responsibility to store this encryption key in a safe place and to re-establish it at the start of each database session.

Encryption of tokenised values in `DS_TOKENS` can be disabled when needed by calling the `set_encrypt_tokenised_values()` procedure.

2.6.4. Token format

As tokens will replace original values, they must at minimum be compatible with the data types and lengths of the table columns in which they will be stored. For text fields, you have various options for the format of the generated tokens: format-preserving tokens (e.g., for credit card numbers), random tokens, alphanumeric tokens, numeric tokens, hash-based tokens, custom tokens, sequential tokens, or encrypted tokens.

In this implementation, all tokens and their (encrypted) original values are stored in `DS_TOKENS` as string. A data type conversion (e.g., to number) takes place as necessary when token and values are used.

2.6.5. Token identical to value

Technically, a token can be the same value as the original value, but doing so would defeat the primary purpose of tokenisation, which is to replace sensitive data with non-sensitive, irreversible tokens. In a typical tokenisation system, the goal is to make it impossible for someone with access to the tokenised data to retrieve the original sensitive information. Therefore, using the original value as the token would not provide any data protection.

By default, this tool will always try to generate tokens that are different from their original values. When a duplicate token is generated, the SQL expression is evaluated repeatedly until a non-already used token is generated. An error is raised when this cannot be achieved after 100 attempts. A typical example is when a non-random function (like `encrypt_xxx()`) is used in the SQL expression, as evaluating it several times will always return the same (duplicate) token.

To allow the tool to generate tokens that are identical to their original values, you need to set the following option: `allow_equal_value=true` . It is necessary to do so when a non-random function is used in the SQL expression.

The `enforce_uniqueness` and `allow_equal_value` options can be combined.

2.6.6. Custom mapping

Should you need to define an ad-hoc mapping (e.g., that cannot be generated using a simple SQL expression), you can populate yourself the `DS_TOKENS` table. When a token is already associated with an original value in this table, the tool will not try to generate a new one. Tokens stored in this table are furthermore never deleted by the tool. If encryption of tokenised values is enabled (which is by default), don’t forget to encrypt the value using the `encrypt_string()` function.

2.7. Masking of unique identifiers

Masking unique identifiers can be challenging due to the inherent nature and purpose of these identifiers. Unique identifiers, such as social security numbers, email addresses, or customer account numbers, are intended to uniquely identify individuals or entities within a system or dataset. The difficulty in masking unique identifiers arises from the need to balance data protection and privacy requirements while maintaining the uniqueness and integrity of the identifiers. Here are a few reasons why masking unique identifiers is difficult:

- **Uniqueness Preservation**: Unique identifiers must remain unique even after masking. If two or more identifiers are masked to the same value, it can lead to data integrity issues, data linkage problems, or errors when processing or analysing the data. Preserving uniqueness while masking requires careful consideration and implementation to ensure that the masked values are still distinct and cannot be linked to the original identifiers.
- **Referential Integrity**: Unique identifiers often serve as key references in relational databases or systems, linking different data elements or tables together. Masking unique identifiers without affecting the referential integrity of the data can be complex. It requires ensuring that the masked identifiers can still be used as valid references without compromising the privacy of individuals associated with those identifiers.
- **Contextual Constraints**: Some unique identifiers have specific formatting or structure requirements, such as credit card numbers or social security numbers. Masking such identifiers while preserving the format or structure can be challenging. The masked values should adhere to the same format and constraints to avoid data inconsistencies or compatibility issues with downstream systems.
- **Data Reversibility**: In certain cases, it may be necessary to preserve the reversibility of masked unique identifiers. For example, in scenarios where there is a need to restore the original identifiers for specific authorised purposes. Designing reversible masking techniques while maintaining privacy can be a complex task, as it requires additional considerations and security measures.

Masking a unique identifier requires to apply an injective function or algorithm and, when reversibility is desired, a bijective function or algorithm.

- An **injective function**, also known as a one-to-one function, is a function where each element of the domain is mapped to a unique element in the codomain. In the context of masking unique identifiers, an injective function would ensure that each original identifier is mapped to a distinct masked value. This property is crucial to maintain uniqueness and integrity during masking. However, implementing an injective function for masking can be challenging due to the need to generate unique masked values without the possibility of collisions or duplication.
- A **bijective function** is an injective function that is also surjective. Surjective functions are functions where each element in the codomain is mapped to by at least one element in the domain. In other words, a bijective function establishes a one-to-one correspondence between the elements of two sets. While bijective functions are not directly linked to data masking, they highlight the importance of maintaining a one-to-one relationship when transforming or mapping data. In the context of unique identifier masking, preserving a one-to-one correspondence between the original identifiers and the masked values ensures the integrity and consistency of the data.

Such injective or bijective functions or algorithm can be developed by applying multiple/successive permutations and transpositions, but they are often just too easy to guess or reverse engineer.

Several techniques can be used to mask primary or unique key columns:

- **Shuffling**, also known as permutation, involves rearranging the order of the original keys. This technique maintains the uniqueness of each key but changes the sequence, making it difficult to link the masked key to its original value. Shuffling can be done based on predefined rules or by applying random permutations.
- **Encryption** involves transforming the original key using an encryption algorithm and a secret key. The encrypted value is typically a random-looking string that cannot be easily deciphered without the key. Encryption provides stronger security than hashing as the original value can be retrieved by decrypting the masked key with the corresponding decryption key.
- **Generating a surrogate key** is another technique commonly used to mask primary or unique keys. A surrogate key is an artificial or synthetic key that is created solely for the purpose of identifying records in a database. Surrogate keys are typically generated using a sequence or an auto-incrementing mechanism.
- **tokenisation** is yet another way to replace a unique value by another unique value.

The tool supports all techniques described above and propagate the masking to all foreign key columns, thereby preserving referential integrity.

2.7.1. Shuffling

This technique has already been described above. You just need to include primary or unique key columns in a shuffling group. In case of composite keys (key composed of several columns), all columns must be shuffled together in a same group. Shuffling only part of the columns would result in foreign key constraint violations. Also, do not partition on a primary key or a unique key as this would annihilate the shuffling (i.e., it would have no effect as each partition would have a single record). As a final note, shuffling does not guarantee that all keys will be changed. Some keys might remain the same, especially when data sets are small.

2.7.2. Format Preserving Encryption (FPE)

Format Preserving Encryption (FPE) can be used to mask or encrypt unique identifiers while preserving their format and ensuring uniqueness. FPE is a cryptographic technique that allows encryption of data while preserving the original format and length of the input. It provides a way to transform sensitive information, such as unique identifiers, into encrypted or masked values that retain the same format and structure.

FPE works by applying encryption algorithms specifically designed to maintain the characteristics of the input data. It ensures that the resulting encrypted values have the same format, length, and allowable character set as the original data. This property makes FPE particularly useful for masking unique identifiers, as it allows for the preservation of the identifier's format and ensures compatibility with systems or processes that rely on the original format.

Unfortunately, the standard `DBMS_CRYPTO` package of Oracle does not support FPE. Format preserved encryption is only available in Oracle Data Safe (which also provides sensitive data discovery and masking capabilities) for which the EC does not have a licence.

This tools comes with two FPE algorithms: one home-made named RSR (Random Substitutions and Rotations) and the standard FF3 algorithm approved by NIST (National Institute of Standards and Technology).

The `DS_CRYPTO_KRN` package provides FPE functions for encrypting and decrypting numbers, integers, strings, and date with or without time component. Both provided algorithms are symetric meaning that the same key is used for both encryption and decryption. In addition to format and character set, uniqueness is also preserved meaning that these functions can be used to encrypt and decrypt primary or unique identifiers without generating uniqueness violations. Data integrity is also guaranteed by propagating the encryption to the foreign keys in cascade (i.e., from master to details down to any level).

As a final note, encryption does not guarantee that all keys are changed, especially when encrypting a set of small numbers. The ciphered number can also potentially be 0, which is not necessarily an expected value for a unique identifier (sequences often start with 1).

Warning: the home-made RSR algorithm is not a modern standard algorithm like AES (Advance Encryption Standard) or FF3 (Feistel Function-based algorithm). Its level of security has not been assessed by any certification authority. As it is implemented in PL/SQL, it can neither deliver the same level of performance as those implemented in compiled or low-level programming languages. The risk to see this algorithm cracked (i.e., deciphering the encrypted data without having access to the decryption key) via a brute force attack or any other method is however very low although not zero.

2.7.3. Sequence-based masking

Using a sequence to generate (numeric) unique identifiers is another viable approach for masking purposes. Such sequence generates unique sequential numbers automatically. By storing the mapping between the old values and the new values (in table `DS_IDENTIFIERS`), the data masking of a primary or unique key can be easily propagated to foreign keys, thereby ensuring that referential data integrity is preserved.

This tool supports sequence-based masking of simple (i.e., non-composite) numeric primary keys and their propagation to all foreign keys in cascade (i.e., down to any level). To proceed, just set the mask type to `SEQUENCE` for the primary column.

2.7.4. tokenisation

tokenisation is yet another method to mask unique identifiers using a mapping table (`DS_TOKENS`) whose content is either generated by the tool using a user-defined SQL expression, either filled-in manually for an ad-hoc mapping.

To proceed, just set the mask type to `TOKENIZE` and define the SQL expression that generates the tokens.

2.8. Final note

To be able to preview, extract and transport masked data, the following actions must be performed:

- Shuffle records for columns having a `SHUFFLE` mask (in `DS_RECORDS`)
- Generate identifiers for columns having a `SEQUENCE` mask (in `DS_IDENTIFIERS`)
- Generate tokens for columns having a `TOKENIZE` mask (in `DS_MASKS`).
- Propagate primary key masking to foreign keys.

All these actions can be performed with a single call to `mask_data_set()` **before** invoking `create_views()` for preview and `transport_data_set()` for extraction and transportation.

3. Procedure

This section describes the steps to follow to discover and mask your sensitive data.

3.1. Data sub-setting

Data masking is applied on-the-fly to the extracted data so, in principle, there is no need to perform a sensitive data discovery on the whole schema. The very first step is therefore to define precisely the data that you want to extract by creating one or several data set definitions.

As this manual is focused on sensitive data discovery and masking, it does not cover data sub-setting. See tutorial and reference manuals for more explanation on how to extract a data set (sub-setting).

3.2. Data discovery

3.2.1. Review pre-defined patterns

The tool comes with a set of pre-defined search patterns that allow to identify several sensitive data types. You need to review those search patterns and make sure that they are complete and that they fit your needs.

As an example, if you know that your database does not contain any financial information (e.g., bank account, credit card numbers, etc.), you may want to disable related search patterns either individually, either for a whole category (e.g., banking). Reducing the number of search patterns will speed up the data discovery process. To proceed, simply set their `disabled_flag` to Y.

If you know some sensitive data types that are not covered by the pre-defined ones, you can simply create your own search patterns by inserting new records in `DS_PATTERNS` , preferably using the API's. Search patterns are also logically grouped into categories, and you can define your own as and when necessary (this is a free text field with no constraint).

Depending on your naming conventions and the language used in your schema, you may also need to adapt the provided regular expressions. They are based on the assumptions that English is used, words within column names are separated with a dash and words within column comments are separated with a space.

To make your changes repeatable, it is recommended to create a script to apply your changes (configuration as code) instead of directly updating the database. This will also avoid losing your changes when a new release of the tool is deployed. Using APIs is also preferred over executing DML statements.

Besides regular expressions, sensitive data can also be detected based on pre-defined look-up data sets (list of countries, cities, currencies, etc.). You can also define your own look-up data sets and reference them in your search patterns.

3.2.2. Launch the data discovery

Launch the sensitive data discovery, for the data set that you want to extract or for the whole schema. See sections 1.4 Data discovery invocation, 1.5. Data discovery report and 1.6. Data discovery results for a description of which procedure to invoke, its parameters, how to get the execution report, and what are the results of this process.

3.2.3. Review the execution report

Examine the report to double check whether the correct sensitive data type has been detected for each table column, especially when several patterns are matching. Check whether patterns have been retained or discarded for good reasons, based on the number of hits or the hit percentage. Looking at sample data may also help to detect incorrect matches.

3.2.4. Review the results

Examine the results stored in the `DS_MASKS` table i.e., which data type or pattern was finally retained for each table column and, more importantly, which mask will be ultimately applied (which is not visible in the execution report).

Should a column be detected as sensitive while it is not, change its `sensitive_flag` to N.

Review the default mask type proposed and its parameters and adapt them as necessary. See section 3.3.1. Review masks for a description of the various cases that need to be tackled.

When you update a record, lock it by setting its `locked_flag` to Y to prevent your change from being overwritten by a subsequent execution of the data discovery.

3.2.5. Iterate as necessary

Based on the knowledge of your business and database schema, you may detect some table columns that have not been reported as sensitive while they should have been. This may be because some search patterns are not well configured (e.g., regular impression is incorrect or not precise enough) or because no search pattern exists to detect them. In this case, review and adapt search patterns as necessary and launch again the sensitive data discovery.

3.3. Data masking

3.3.1. Review masks

Launching the sensitive data discovery is recommended but optional. If you don’t run it, the `DS_MASKS` table will be empty, and you will have to create your masks by hand.

The data discovery process is working column by column and is therefore not capable of detecting relationships between columns.

As an example, it cannot detect that a full person’s name is the concatenation of a first name and last name. You will have to change the SQL expression of the mask to build this full name based the two other columns.

The default masking associated with the pre-defined search patterns is always of type SQL, while SHUFFLING would maybe more appropriate to preserve the consistency between columns.

As an example, postal codes depend on cities which, in turn, depend on countries. Rather than generating random values for each of these columns (without keeping any relationship between them), a more appropriate mask would be to shuffle these columns together. You may also want to use shuffling partitions (e.g., shuffling given names partitioned by gender to make sure that male given names stay associated with males).

Rather than generating random values, you might also want to pick values at random from a pre-defined CSV data set, or from a user-defined data set (CSV or SQL based). As an example, if your schema already contains person given names, you can easily define a sub-set (e.g., with the most common ones), using a SQL expression, from which values will be picked at random.

To avoid reidentification, unique identifiers of sensitive data (e.g., a person id) must also be masked and it is up to you to determine which method should be used (e.g., format preserved encryption, surrogate keys generated from a sequence, etc.).

Depending on your needs, you might also want to encrypt your data rather than obfuscating or randomise them.

3.3.2. Preview masked data

Masked data can be previewed by creating ad-hoc views using `create_views()` . Keep in mind that, for partially extracted tables, no data are shown if record rowids have not been extracted first by calling `extract_data_set_rowids()` . Masking must also be performed beforehand by calling `mask_data_set()` .

These views show exactly how data will be look like during their extraction. As you can see by looking at their definition (DDL statement), SQL masks are applied (function calls), records are shuffled (by joining with `DS_RECORDS`), and masking of primary/unique keys are propagated to the foreign keys (by joining detail tables with their master).

3.3.3. Iterate as necessary

Previewing data may reveal masking problems (e.g., column not masked at all or not masked as expected), in which case you will need to review and adapt the masks and iterate. Note that views may need to be dropped and created again to take into account some changes made to the masks.

3.4. Data extraction and transportation

Data are masked during extraction and transportation, whatever is the method used (scripts, XML, database links, etc.). See the tutorial and the reference manual for a description of how to proceed.

4. Tutorial

4.1. Overview

This tutorial demonstrates how to perform a sensitive data discovery and a data masking on a sample database. The `ds_persons.sql` script containing all SQL statements explained in this tutorial can be found in the `tutorial/scripts directory` .

4.2. Data model

The sample database is made up of the following tables:

- `TMP_PERSONS` : persons
- `TMP_PER_CREDIT_CARDS` : person’s credit cards
- `TMP_PER_TRANSACTIONS` : person’s transactions

A person has the following attributes:

- `PER_ID` : person unique identifier
- `GIVEN_NAME` : person’s given name
- `FAMILY_NAME` : person’s family name
- `GENDER` : gender, male or female
- `BIRTH_DATE` : date of birth

A person may have one or several credit cards with the following attributes:

- `PER_ID` : person unique identifier (FK+PK)
- `CREDIT_CARD_NUMBER` : credit card number (PK)
- `EXPIRY_DATE` : date of expiration of the credit card

To demonstrate the propagation of primary key masking to foreign keys, a credit card is identified by both the person and the credit card number (in reality, the credit card number is sufficient).

A person may perform one or several transactions with each of its credit cards. Transactions have the following attributes:

- `PER_ID` : person unique identifier (FK+PK)
- `CREDIT_CARD_NBR` : credit card number (FK+PK)
- `TRANSACTION_TIMESTAMP` : transaction date and time (PK)
- AMOUNT: amount of the transaction

A transaction is identified by the person, its credit card number, and its timestamp. Note that the credit card number has a slightly different column name in this table (on purpose).

All tables and columns have comments (that are used during sensitive data discovery).

4.3. Sample data

The tool comes with some pre-defined data sets and a library of functions (amongst which random functions) that can be used not only to mask data but also to generate sample data easily.

10 persons are generated in the following way:

- `PER_ID` : random number between 1000 and 1999
- `GIVEN_NAME` : random name from `INT_GIVEN_NAMES_250.GIVEN_NAME` data set
- `FAMILY_NAME` : random name from the `EU6_FAMILY_NAMES_217.FAMILY_NAME` data set
- `GENDER` : random value from `INT_GIVEN_NAMES_250.GENDER` data set
- `BIRTH_DATE` : at random in such a way that persons are between 21 and 65 years old

Note that, gender and given name are picked at random from the same data set. To keep them aligned, the same seed value (rownum) is passed to the `random_value_from_data_set()` function.

2 credit cards are generated for each person in the following way:

- `PER_ID` : from persons
- `CREDIT_CARD_NUMBER` : random credit card number
- `EXPIRY_DATE` : random expiry date

2 transactions are generated for each credit card in the following way:

- `PER_ID` : from credit cards
- `CREDIT_CARD_NBR` : from credit cards

- `AMOUNT` : random amount between 90 and 990

For the purpose of this tutorial, it is desirable to always generate the same set of data (to have them in sync with this manual). Consequently, a constant seed value is passed to all random functions to make them deterministic.

Here is the content of the `TMP_PERSONS` table:

PER_ID	FIRST_NAME	LAST_NAME	GENDER	BIRTH_DATE
1036	Joseph	Wouters	M	08/01/60
1098	Sarah	Barry	F	11/10/62
1106	Elizabeth	Sow	F	07/02/63
1117	Eric	Bernard	M	31/07/63
1420	Nathalie	Marchetti	F	09/12/76
1647	Guy	Diederich	M	30/11/86
1713	Pamela	De Groot	F	29/10/89
1740	Maurice	Van Dijk	M	09/01/91
1750	Frances	Bakker	M	22/06/91
1800	Irene	Sanchez	F	01/09/93

Table 5 – generated persons

Here is the content of the `TMP_PER_CREDIT_CARDS` table:

PER_ID	CREDIT_CARD_NUMBER	EXPIRY_DATE
1036	6011110076415299	30/06/27
1036	6011860385779351	31/03/27
1098	367697632760955	30/11/23
1098	6011370298651499	31/03/27
1106	4026916201143731	31/07/25
1106	5562414551273958	31/08/26

PER_ID	CREDIT_CARD_NUMBER	EXPIRY_DATE
1117	303295936884568	31/08/23
1117	4457161279244404	31/10/25
1420	369529412905815	30/04/24
1420	5978054329281087	28/02/27
1647	305413347409085	30/06/24
1647	6011948344238070	30/06/27
1713	303834669323863	31/12/23
1713	5520206148974556	31/05/26
1740	316094038857605	31/03/24
1740	5395218193873577	31/12/26
1750	4058631345172659	31/07/25
1750	6011146986687437	29/02/28
1800	339906777485111	31/01/24
1800	4461453377524173	31/05/25

Table 6 – generated credit cards

Here is the content of the TMP_PER_TRANSACTIONS table:

PER_ID	CREDIT_CARD_NBR	TIMESTAMP	AMOUNT
1036	6011110076415299	07/06/23 08:17:30,047912000	766
1036	6011110076415299	07/06/23 08:17:44,278714000	179
1036	6011860385779351	07/06/23 08:17:30,047912000	179
1036	6011860385779351	07/06/23 08:17:44,278714000	845
1098	367697632760955	07/06/23 08:17:30,047912000	673
1098	367697632760955	07/06/23 08:17:44,278714000	142
1098	6011370298651499	07/06/23 08:17:30,047912000	122

PER_ID	CREDIT_CARD_NBR	TIMESTAMP	AMOUNT
1098	6011370298651499	07/06/23 08:17:44,278714000	558
1106	4026916201143731	07/06/23 08:17:30,047912000	757
1106	4026916201143731	07/06/23 08:17:44,278714000	440
1106	5562414551273958	07/06/23 08:17:30,047912000	811
1106	5562414551273958	07/06/23 08:17:44,278714000	783
1117	303295936884568	07/06/23 08:17:30,047912000	185
1117	303295936884568	07/06/23 08:17:44,278714000	156
1117	4457161279244404	07/06/23 08:17:30,047912000	732
1117	4457161279244404	07/06/23 08:17:44,278714000	563
1420	369529412905815	07/06/23 08:17:30,047912000	469
1420	369529412905815	07/06/23 08:17:44,278714000	203
1420	5978054329281087	07/06/23 08:17:30,047912000	195
1420	5978054329281087	07/06/23 08:17:44,278714000	649
1647	305413347409085	07/06/23 08:17:30,047912000	811
1647	305413347409085	07/06/23 08:17:44,278714000	178
1647	6011948344238070	07/06/23 08:17:30,047912000	780
1647	6011948344238070	07/06/23 08:17:44,278714000	480
1713	303834669323863	07/06/23 08:17:30,047912000	473
1713	303834669323863	07/06/23 08:17:44,278714000	249
1713	5520206148974556	07/06/23 08:17:30,047912000	521
1713	5520206148974556	07/06/23 08:17:44,278714000	952
1740	316094038857605	07/06/23 08:17:30,047912000	248
1740	316094038857605	07/06/23 08:17:44,278714000	685
1740	5395218193873577	07/06/23 08:17:30,047912000	275

PER_ID	CREDIT_CARD_NBR	TIMESTAMP	AMOUNT
1740	5395218193873577	07/06/23 08:17:44,278714000	571
1750	4058631345172659	07/06/23 08:17:30,047912000	622
1750	4058631345172659	07/06/23 08:17:44,278714000	438
1750	6011146986687437	07/06/23 08:17:30,047912000	226
1750	6011146986687437	07/06/23 08:17:44,278714000	113
1800	339906777485111	07/06/23 08:17:30,047912000	937
1800	339906777485111	07/06/23 08:17:44,278714000	275
1800	4461453377524173	07/06/23 08:17:30,047912000	444
1800	4461453377524173	07/06/23 08:17:44,278714000	976

Table 7 – generated transactions

4.4. Data Sub-Setting

The first step is to define the rules for extracting a sub-set of the sample data. Let's suppose that we want to extract half of the persons and their details (credit cards and transactions). As data sub-setting is not the purpose of this tutorial, the process will not be explained in detail.

The steps are the following:

1. Create a data set definition named `tmp_persons`
2. Include `TMP_PERSONS` as the base table with a percentage of 50% and its details recursively (down to 3 levels)
3. Extract data set rowids

Here is the code:

```
exec ds_utility_krn.create_or_replace_data_set_def('tmp_persons');
exec ds_utility_krn.include_tables(
 p_set_id=>ds_utility_krn.get_data_set_def_by_name('tmp_persons')
,p_table_name=>'TMP_PERSONS'
,p_recursive_level=>3
,p_extract_type=>'B'
,p_percentage=>50
);
exec ds_utility_krn.extract_data_set_rowids(
   p_set_id=>ds_utility_krn.get_data_set_def_by_name('tmp_persons')
);
```

Here are the 5 extracted persons (50% of 10):

PER_ID	FIRST_NAME	LAST_NAME	GENDER	BIRTH_DATE
1036	Joseph	Wouters	M	08/01/60
1098	Sarah	Barry	F	11/10/62
1106	Elizabeth	Sow	F	07/02/63
1117	Eric	Bernard	M	31/07/63
1420	Nathalie	Marchetti	F	09/12/76

Table 8 – extracted persons

Their credit card numbers and transactions are also extracted.

4.5. Data discovery

The tool comes with a pre-defined set of search patterns that we can adapt or complemented as needed. Let’s suppose that we do not know anything about our data. In this case, we will keep all pre-defined patterns activated (they are all enabled by default).

Let’s launch a data discovery limited to the tables of the data set that was just created, after having enabled information messages to get an execution report:

```
exec ds_utility_krn.set_message_filter('EI'); -- E)rror and I)nfo messages
exec ds_utility_krn.discover_sensitive_data(
` `p_commit=>TRUE
,p_set_id=>ds_utility_krn.get_data_set_def_by_name('tmp_persons')
)
```

The report sent to `dbms_output` is the following:

```
Info: *TMP_PERSONS.FIRST_NAME: Pattern "Person given name (sys)" RETAINED; matching on: na
Info:  TMP_PERSONS.FIRST_NAME: Pattern "Country code alpha-3 (sys)" DISCARDED on: min pct
Info: *TMP_PERSONS.LAST_NAME: Pattern "Person family name (sys)" RETAINED; matching on: na
Info:  TMP_PERSONS.LAST_NAME: Pattern "Person given name (sys)" NOT BEST; matching on: dat
Info: *TMP_PERSONS.GENDER: Pattern "Gender - alpha (sys)" RETAINED; matching on: name rege
Info:  TMP_PERSONS.GENDER: Pattern "Gender - any type (sys)" NOT BEST; matching on: name r
Info: *TMP_PERSONS.BIRTH_DATE: Pattern "Person birth date (sys)" RETAINED; matching on: na
Info:  TMP_PER_CREDIT_CARDS.CREDIT_CARD_NUMBER: Pattern "Visa card number (sys)" NOT BEST;
Info:  TMP_PER_CREDIT_CARDS.CREDIT_CARD_NUMBER: Pattern "Mastercard card number (sys)" NOT
Info: *TMP_PER_CREDIT_CARDS.CREDIT_CARD_NUMBER: Pattern "Generic credit card number (sys)"
Info: *TMP_PER_CREDIT_CARDS.EXPIRY_DATE: Pattern "Expiry date (sys)" RETAINED; matching on
Info:  TMP_PER_TRANSACTIONS.CREDIT_CARD_NBR: Pattern "Visa card number (sys)" NOT BEST; ma
Info:  TMP_PER_TRANSACTIONS.CREDIT_CARD_NBR: Pattern "Mastercard card number (sys)" NOT BE
Info: *TMP_PER_TRANSACTIONS.CREDIT_CARD_NBR: Pattern "Generic credit card number (sys)" RE
```

You will note the following:

- One first name (Guy) is also a country code, but the country pattern was ultimately discarded based on the minimum hit percentage.
- One last name (Bernard) is also a given name, but the given name pattern was ultimately discarded as it was not the best.
- Several card issuers were detected but it’s the most generic card number pattern that is ultimately retained as the best.
- Sample values help to double check whether the retained pattern is correct or not.

A look in `DS_MASKS` allows to see the sensitive columns that have been detected and the masking type that will be applied by default:

TABLE_NAME	COLUMN_NAME	MSK_TYPE
TMP_PERSONS	BIRTH_DATE	SQL
TMP_PERSONS	FIRST_NAME	SQL

TABLE_NAME	COLUMN_NAME	MSK_TYPE
TMP_PERSONS	GENDER	SQL
TMP_PERSONS	LAST_NAME	SQL
TMP_PER_CREDIT_CARDS	CREDIT_CARD_NUMBER	SQL
TMP_PER_CREDIT_CARDS	EXPIRY_DATE	SQL
TMP_PER_TRANSACTIONS	CREDIT_CARD_NBR	INHERIT

Table 9 – default mask type after discovery

A SQL function will be applied to all columns except the last one that will INHERIT its masking from its mater table (`TMP_PER_CREDIT_CARDS`) through its foreign key. Here follows the masking functions generated by default:

TABLE_NAME	COLUMN_NAME	MSK_PARAMS (function)
TMP_PERSONS	BIRTH_DATE	Random_date
TMP_PERSONS	FIRST_NAME	Random_from_data_set
TMP_PERSONS	GENDER	Random (M/F)
TMP_PERSONS	LAST_NAME	Random from data set
TMP_PER_CREDIT_CARDS	CREDIT_CARD_NUMBER	Obfuscate_credit_card_nber
TMP_PER_CREDIT_CARDS	EXPIRY_DATE	Random_expiry_date
TMP_PER_TRANSACTIONS	CREDIT_CARD_NBR	n/a (inherit from master)

Table 10 – default SQL expressions after discovery

4.6. Data masking

The masks generated by the sensitive data discovery are not necessarily complete (e.g., `PER_ID` was not detected as sensitive) and don’t necessarily meet your needs either (e.g., genders generated at random will lead to incoherences with given names generated from a data set).

We are therefore going to make the following changes to the masks:

- Instead of generating random first name and gender, shuffle existing first names and genders, partitioned by gender (to keep the relationship between them).
- Instead of obfuscating the credit card number (i.e., randomise only part of it), encrypt it using a private key so that it can be decrypted at a later stage.

Here is how to update existing masks using the APIs:

```
exec ds_utility_krn.update_mask_properties (
  p_table_name=>'TMP_PERSONS', p_column_name=>'FIRST_NAME', p_msk_type=>'SHUFFLE'
, p_shuffle_group=>1, p_params=>NULL, p_locked_flag=>'Y'
);
exec ds_utility_krn.update_mask_properties (
  p_table_name=>'TMP_PERSONS', p_column_name=>'GENDER', p_msk_type=>'SHUFFLE'
, p_shuffle_group=>1, p_partition_bitmap=>1, p_params=>NULL, p_locked_flag=>'Y'
);
exec ds_utility_krn.update_mask_properties (
  p_table_name=>'TMP_PER_CREDIT_CARDS', p_column_name=>'CREDIT_CARD_NUMBER'
, p_params=>'ds_masker_krn.encrypt_credit_card_number(credit_card_number)'
, p_locked_flag=>'Y'
);
```

You will note:

- Mask type is set to `SHUFFLE` for both columns.
- These 2 columns are part of shuffling group #1 to be shuffled together.
- Shuffling group #1 is partitioned on `GENDER` (bitmap 1 means group 1); so male and female given names will be shuffled separately.
- `PARAMS`, which is not applicable to shuffling, is reset.
- Changes are locked to avoid being overwritten by any subsequent data discovery.

Here are the modified masks (SQL expressions are not shown):

TABLE_NAME	COLUMN_NAME	TYPE	GROUP	PARTITION
TMP_PERSONS	*FIRST_NAME*	*SHUFFLE*	*1*	
TMP_PERSONS	LAST_NAME	SQL		
TMP_PERSONS	*GENDER*	*SHUFFLE*	*1*	*1*
TMP_PERSONS	BIRTH_DATE	SQL		
TMP_PER_CREDIT_CARDS	CREDIT_CARD_NUMBER	SQL		

TABLE_NAME	COLUMN_NAME	TYPE	GROUP	PARTITION
TMP_PER_CREDIT_CARDS	EXPIRY_DATE	SQL		
TMP_PER_TRANSACTIONS	CREDIT_CARD_NBR	INHERIT		

Table 11 – adapted mask types

We also want to declare the `PER_ID` column as sensitive by inserting a mask for this column:

```
exec ds_utility_krn.insert_mask(
 p_table_name=>'TMP_PERSONS',p_column_name=>'PER_ID'
,p_sensitive_flag=>’Y’
);
```

This `PER_ID` , which is a primary key column, can be masked in 3 different ways:

1. `PER_ID` s can be generated as a sequential number using an in-memory sequence
2. `PER_ID` s can be shuffled (exchanged randomly)
3. `PER_ID` s can be encrypted

As a reminder, the function or algorithm that is applied must preserve uniqueness, so no random function or algorithm is allowed here as it would potentially lead to collisions.

Scenario 1 – Sequence

```
exec ds_utility_krn.update_mask_properties(
  p_table_name=>'TMP_PERSONS', p_column_name=>'PER_ID'
, p_msk_type=>'SEQUENCE', p_params=>'START WITH 10 INCREMENT BY 10'
, p_locked_flag=>'Y')
;
```

You will notice that a start value and an increment can be defined in the parameters (with the same syntax as for an Oracle sequence).

Here are the resulting masks:

TABLE_NAME	COLUMN_NAME	MSK_TYPE
TMP_PERSONS	BIRTH_DATE	SQL
TMP_PERSONS	FIRST_NAME	SHUFFLE

TABLE_NAME	COLUMN_NAME	MSK_TYPE
TMP_PERSONS	GENDER	SHUFFLE
TMP_PERSONS	LAST_NAME	SQL
TMP_PERSONS	*PER_ID*	*SEQUENCE*
TMP_PER_CREDIT_CARDS	CREDIT_CARD_NUMBER	SQL
TMP_PER_CREDIT_CARDS	EXPIRY_DATE	SQL
TMP_PER_CREDIT_CARDS	*PER_ID*	*INHERIT*
TMP_PER_TRANSACTIONS	CREDIT_CARD_NBR	INHERIT
TMP_PER_TRANSACTIONS	*PER_ID*	*INHERIT*

Table 12 – mask types for scenario 1

You will note the `SEQUENCE` masking type on `PER_ID` and the fact that this mask is propagated to (or inherited by) all foreign key columns.

Scenario 2 – Shuffling

```
exec ds_utility_krn.update_mask_properties(
   p_table_name=>'TMP_PERSONS', p_column_name=>'PER_ID', p_msk_type=>'SHUFFLE'
  ,p_shuffle_group=>2, p_partition_bitmap=>NULL, p_locked_flag=>'Y'
);
```

A second shuffling group is created on the `PER_ID` without any partition. As a reminder, creating a partition on a PK column would annihilate the shuffling.

Here are the resulting masks:

TABLE_NAME	COLUMN_NAME	TYPE	GROUP	PARTITION
TMP_PERSONS	BIRTH_DATE	SQL		
TMP_PERSONS	FIRST_NAME	SQL		
TMP_PERSONS	GENDER	SHUFFLE	1	1
TMP_PERSONS	LAST_NAME	SQL		
TMP_PERSONS	*PER_ID*	*SHUFFLE*	*2*	

TABLE_NAME	COLUMN_NAME	TYPE	GROUP	PARTITION
TMP_PER_CREDIT_CARDS	CREDIT_CARD_NUMBER	SQL		
TMP_PER_CREDIT_CARDS	EXPIRY_DATE	SQL		
TMP_PER_CREDIT_CARDS	*PER_ID*	*INHERIT*		
TMP_PER_TRANSACTIONS	CREDIT_CARD_NBR	INHERIT		
TMP_PER_TRANSACTIONS	*PER_ID*	*INHERIT*		

Table 13 – mask types for scenario 2

Scenario 3 – Encryption

```
exec ds_utility_krn.insert_mask (
   p_table_name=>'TMP_PERSONS',p_column_name=>'PER_ID'
 , p_msk_type=>'SQL', p_params=>'ds_masker_krn.encrypt_number(per_id)'
 , p_locked_flag=>'Y'
);
```

As only the SQL expression was changed, the mask types remain as in Table 11.

Any mask applied to a primary key is automatically propagated to all foreign key columns, thereby overwriting any previously defined mask, if any. `PER_ID` s of credit cards and transactions is therefore INHERITed from the masked person's `PER_ID` , whatever is the scenario.

4.7. Data Preview

Before previewing, extracting, or transporting data, some masking operations (shuffling records, generate unique ids based on a sequence, generate tokens, and propagate PK masking), must be performed by calling `ds_utility_krn.mask_data_set()` .

Data set can be previewed, with or without data masking, by creating ad-hoc views. You can for instance use a different suffix to distinguish original data (e.g., `_ORI`) from masked data (e.g., `_MSK`), as in the following examples.

With data masking:

```
exec ds_utility_krn.create_views (
  p_set_id=>ds_utility_krn.get_data_set_def_by_name('tmp_persons')
, p_view_suffix=>'_MSK', p_mask_data=>TRUE, p_include_rowid=>TRUE
);
```

Without data masking:

```
exec ds_utility_krn.create_views(
  p_set_id=>ds_utility_krn.get_data_set_def_by_name('tmp_persons')
, p_view_suffix=>'_ORI', p_mask_data=>FALSE, p_include_rowid=>TRUE
);
```

The `p_include_rowid` parameter allows to add the rowid to both views so that masked and original data can be more easily compared (especially when the PK is masked).

Please have a look at the DML statements of masked views to check which function or technique is used for each individual column.

- Sequence generation is implemented by calling `ds_utility_krn.in_mem_seq_nextval()` function.
- Shuffling is implemented with an inner join on `DS_RECORDS` plus the involved table.
- Encryption is implemented by calling a `ds_masker_krn.encrypt_xxx()` function.

Here follow person’s data that will be extracted in each scenario.

Scenario 1 – Sequence

PER_ID	FIRST_NAME	LAST_NAME	GENDER	BIRTHDATE
40	Eric	Thill	M	19/11/94
30	Elizabeth	Sanz	F	21/05/79
50	Nathalie	Majerus	F	28/07/89
10	Joseph	Amato	M	15/09/85
20	Sarah	Navarro	F	22/05/87

Table 14 – masked persons for scenario 1

You will note that:

- `PER_ID` s are generated from a sequence with a starting value and an increment of 10.

- Compared to Table 8, first names and genders have been shuffled together (keeping the coherence between them).

Scenario 2 – Shuffling

PER_ID	FIRST_NAME	LAST_NAME	GENDER	BIRTHDATE
1036	Sarah	Navarro	F	22/05/87
1098	Eric	Amato	M	15/09/85
1106	Joseph	Thill	M	19/11/94
1117	Nathalie	Sanz	F	21/05/79
1420	Elizabeth	Majerus	F	28/07/89

Table 15 – masked persons for scenario 2

You will note that `PER_IDs` are shuffled but remain identical to their initial values.

Scenario 3 – Encryption

PER_ID	FIRST_NAME	LAST_NAME	GENDER	BIRTHDATE
8818	Joseph	Thill	M	19/11/94
5330	Nathalie	Sanz	F	21/05/79
5370	Elizabeth	Majerus	F	28/07/89
4794	Eric	Amato	M	15/09/85
6264	Sarah	Navarro	F	22/05/87

Table 16 – masked persons for scenario 3

You will note that `PER_ID` s are encrypted and their format (i.e., their maximum number of digits – 4 in this example) is preserved.

As a last step, views can be dropped in the following way.

With masking:

```
exec ds_utility_krn.drop_views (
  p_set_id=>ds_utility_krn.get_data_set_def_by_name('tmp_persons')
, p_view_suffix=>'_W'
);
```

Without masking:

```
exec ds_utility_krn.drop_views(
  p_set_id=>ds_utility_krn.get_data_set_def_by_name('tmp_persons')
, p_view_suffix=>'_WO'
);
```

4.8. Data extraction and transportation

Masked data can be extracted and transported to another schema in various ways:

1. Via a SQL script (transported and executed manually in the target schema)
2. Via a SQL script (directly executed in the target schema through a database link)
3. Via a database link (direct execution of DML statements)
4. Via XML (transported and re-imported manually in the target schema)
5. Via a SQL script that is prepared but not executed

```
truncate table ds_output;
exec ds_utility_krn.transport_data_set (
  p_set_id=>ds_utility_krn.get_data_set_def_by_name('tmp_persons')
, p_method=>'PREPARE-SCRIPT',p_output=>'DS_OUTPUT',p_mode=>'I'
);
select text from ds_output order by line;
```

4.8.1. Via a SQL script that is directly executed

```
exec ds_utility_krn.transport_data_set (
  p_set_id=>ds_utility_krn.get_data_set_def_by_name('tmp_persons')
, p_method=>'EXECUTE-SCRIPT',p_output=>'DS_OUTPUT',p_mode=>'I'
, p_db_link=>'DBCC_DIGIT_01_T.CC.CEC.EU.INT'
);
```

Note that the database link must have the same name as the target database to allow remote procedure call.

4.8.2. Via database link

```
update ds_tables
   set target_db_link = 'DBCC_DIGIT_01_T.CC.CEC.EU.INT'
 where set_id=ds_utility_krn.get_data_set_def_by_name('tmp_persons')
;
exec ds_utility_krn.transport_data_set (
  p_set_id=>ds_utility_krn.get_data_set_def_by_name('tmp_persons')
, p_method=>'DIRECT-EXECUTE',p_output=>'DS_OUTPUT',p_mode=>'I'
);
```

4.8.3. Via XML

```
exec ds_utility_krn.export_data_set_to_xml (
  p_set_id=>ds_utility_krn.get_data_set_def_by_name('tmp_persons')
);
```

Data are stored as XML in the `DS_RECORDS` and `DS_TABLES` internal tables. You must transport these tables to the target schema and then reimport the data from XML:

```
exec ds_utility_krn.import_data_set_from_xml (
  p_set_id=>ds_utility_krn.get_data_set_def_by_name('tmp_persons')
);
```

This method can also be used for an in-place subsetting (i.e., in the same schema). You need to delete all the data before reimporting the subset from XML.

5. References

5.1. Oracle Regular Expressions

Here follows the metacharacters supported by Oracle in regular expressions:

Metacharacter Syntax	Operator Name	Description
.	Any Character -- Dot	Matches any character
+	One or More -- Plus Quantifier	Matches one or more occurrences of the preceding subexpression
?	Zero or One -- Question Mark Quantifier	Matches zero or one occurrence of the preceding subexpression
*	Zero or More -- Star Quantifier	Matches zero or more occurrences of the preceding subexpression
{*m*}	Interval--Exact Count	Matches exactly*m* occurrences of the preceding subexpression
{*m*,}	Interval--At Least Count	Matches at least *m* occurrences of the preceding subexpression
{*m*,*n*}	Interval--Between Count	Matches at least *m*, but not more than *n* occurrences of the preceding subexpression
[...]	Matching Character List	Matches any character in list ...
[^ ...]	Non-Matching Character List	Matches any character not in list ...
|	Or	'a|b' matches character 'a' or 'b'.
(...)	Subexpression or Grouping	Treat expression ... as a unit. The subexpression can be a string of literals or a complex expression containing operators.
n	Backreference	Matches the nth preceding subexpression, where *n* is an integer from 1 to 9.
\	Escape Character	Treat the subsequent metacharacter in the expression as a literal.
^	Beginning of Line Anchor	Match the subsequent expression only when it occurs at the beginning of a line.

Metacharacter Syntax	Operator Name	Description
$	End of Line Anchor	Match the preceding expression only when it occurs at the end of a line.
[:*class*:]	POSIX Character Class	Match any character belonging to the specified character *class*. Can be used inside any list expression.
[.*element*.]	POSIX Collating Sequence	Specifies a collating sequence to use in the regular expression. The *element* you use must be a defined collating sequence, in the current locale.
[=*character*=]	POSIX Character Equivalence Class	Match characters having the same base character as the *character* you specify.

Table 17 – metacharacters supported in Oracle regular expressions

5.2. APIs

This section lists the methods that are available in each package of the tool. Only those related to data discovery, data masking and data encryption are listed here. See each package specification for a detailed description of their parameters.

5.2.1. DS_UTILITY_KRN

METHOD	PURPOSE
`delete_mask`	Delete mask(s)
`delete_pattern`	Delete pattern(s)
`discover_sensitive_data`	Launch a sensitive data discovery
`generate_identifiers`	Generate unique identifiers based on a sequence
`insert_mask`	Insert new mask(s)
`insert_pattern`	Insert a new pattern
`mask_data_set`	Mask a data set (i.e., generate identifiers, shuffle records, propagate pk masking, etc.).

METHOD	PURPOSE
`propagate_pk_masking`	Propagate masking of PKs to FKs
`set_masking_mode`	Enable/disable masking
`shuffle_records`	Shuffle records
`update_mask_properties`	Update mask(s) properties
`update_pattern_properties`	Update pattern(s) properties

Table 18 – methods of the DS_UTILITY_KRN package

5.2.2. DS_MASKER_KRN

METHOD	PURPOSE
`decrypt_bban`	Decrypt a BBAN
`decrypt_credit_card_number`	Decrypt a credit card number
`decrypt_number`	Decrypt a number
`decrypt_number_with_mod97`	Decrypt a number having a modulo 97 checksum
`decrypt_string`	Decrypt a string
`encrypt_bban`	Encrypt a BBAN
`encrypt_credit_card_number`	Encrypt a credit card number
`encrypt_number`	Encrypt a number
`encrypt_number_with_mod97`	Encrypt a number having a modulo 97 checksum
`encrypt_string`	Encrypt a string
`filter_characters`	Filter characters from a string according to a mask
`format_number`	Format a number according to a pattern
`is_valid_bban`	Check if a BBAN is valid
`is_valid_credit_card_number`	Check if a credit card number is valid
`is_valid_iban`	Check if an IBAN is valid

METHOD	PURPOSE
`is_valid_luhn_number`	Check if a number with a Luhn check digit is valid
`is_valid_number_with_mod97`	Check if a number with a modulo 97 checksum is valid
`lorem_ipsum_text`	Generate lorem ipsum text
`luhn_checkdigit`	Compute the Luhn check digit of a number
`luhn_sum`	Compute the Luhn checksum of a number
`mask_bban`	Mask a BBAN
`mask_credit_card_number`	Mask a credit card number
`mask_iban`	Mask an IBAN
`mask_number`	Mask a number
`mask_number_with_mod97`	Mask a number having a modulo 97 checksum
`mask_string`	Mask a string
`obfuscate_bban`	Obfuscate a BBAN
`obfuscate_credit_card_number`	Obfuscate a credit card number
`obfuscate_date`	Obfuscate a date
`obfuscate_number_with_mod97`	Obfuscate a number having a modulo 97 checksum
`obfuscate_string`	Obfuscate a string
`random_bban`	Generate a random BBAN
`random_credit_card_number`	Generate a random credit card number
`random_date`	Generate a random date
`random_time`	Generate a random time
`random_expiry_date`	Generate a random expiry date
`random_integer`	Generate a random integer
`random_luhn_number`	Generate a random number having a Luhn check digit
`random_name`	Generate a random name

METHOD	PURPOSE
random_number	Generate a random number
random_number_with_mod97	Generate a random number having a modulo 97 checksum
random_string	Generate a random string
random_value_from_data_set	Generate a random value from a data set
unaccentuate_string	Remove accentuated characters from a string
set_encryption_key	Set the private key for encryption and decryption

Table 19 – methods of the DS_MASKER_KRN package

Other methods of the package that are not listed above are for internal use only.

Encryption methods provided by this package produce cipher values having exactly the same length, precision and/or scale than the input value. As an example, encrypting a single digit number will produce a single digit number (e.g., “0” might be encrypted as “9”). For strings, only printable characters of the latin aphabet no.1 (ISO-8859-1) can be used. For more advanced options, methods of DS_CRYPTO_KRN should be used instead.

5.2.3. DS_CRYPTO_KRN

This package provides encryption and decryption methods for basic data types (number, integer, string, and date with or without time component).

METHOD	PURPOSE
decrypt_date	Decrypt a date (with its time component if any)
decrypt_integer	Decrypt an integer
decrypt_number	Decrypt a number
decrypt_string	Decrypt a string
encrypt_date	Encrypt a date (with its time component if any)
encrypt_integer	Encrypt an integer
encrypt_number	Encrypt a number
encrypt_string	Encrypt a string

METHOD	PURPOSE
`set_encryption_algo`	Set encryption algorithm (RSR or FF3)
`set_encryption_key`	Set private key for encryption and decryption
`set_tweak`	Set tweak (for FF3 only)

Table 20 – methods of the DS_CRYPTO_KRN package

The encryption and decryption methods of this package offer additional parameters and possibilities than those of the `DS_MASKER_KRN` package. As an example, you can specify exactly the character set that you want to use when encrypting and decrypting strings (e.g., alpha lowercase, uppercase, or mixed case, alphanumeric, accentuated characters, special symbols, etc). You can also specify the length, precision, and scale of the columns in which the ciphered values will be stored. By doing so, the encryption of a single digit number can result in a several digits number (e.g., “0” could be encrypted as “12483” if the maximum precision is 5). It is worth to mention that the same parameters must be passed to both encryption and decryption functions. Not doing so will lead to wrong decrypted values.