

# Data Set Utility- Reference Manual v25.0

**Author: Philippe Debois (European Commission)**

## Table of Contents

- [1. Internal Data Model](#)
  - [1.0.1. Data Model Diagram](#)
  - [1.0.2. Configuration tables](#)
  - [1.0.3. Transactional tables](#)
- [2. Concepts and APIs](#)
  - [2.1. Data sets](#)
    - [2.1.1. Properties](#)
      - [2.1.1.1. Set id](#)
      - [2.1.1.2. Set name](#)
      - [2.1.1.3. Set type](#)
      - [2.1.1.4. System flag](#)
      - [2.1.1.5. Disabled flag](#)
      - [2.1.1.6. Visible flag](#)
      - [2.1.1.7. Capture flag](#)
      - [2.1.1.8. Capture mode](#)
      - [2.1.1.9. Capture sequence \(internal use only\)](#)
      - [2.1.1.10. Line separator character](#)
      - [2.1.1.11. Column separator character](#)
      - [2.1.1.12. Left separator character](#)
      - [2.1.1.13. Right separator character](#)
      - [2.1.1.14. Column names row](#)
      - [2.1.1.15. Column types row](#)
      - [2.1.1.16. Data row](#)
      - [2.1.1.17. Params](#)
    - [2.1.2. Operations](#)
      - [2.1.2.1. Create](#)
      - [2.1.2.2. Create or replace](#)
      - [2.1.2.3. Clone](#)
      - [2.1.2.4. Delete](#)
      - [2.1.2.5. Get last](#)
      - [2.1.2.6. Get by name](#)

- 2.1.2.7. Clear
  - 2.1.2.8. Update properties
- 2.2. Tables
  - 2.2.1. Properties
    - 2.2.1.1. Table Id
    - 2.2.1.2. Set Id
    - 2.2.1.3. Table Name
    - 2.2.1.4. Table Alias
    - 2.2.1.5. Extract type
    - 2.2.1.6. Row limit
    - 2.2.1.7. Percentage
    - 2.2.1.8. Row count
    - 2.2.1.9. Source count (internal use only)
    - 2.2.1.10. Extract count (internal use only)
    - 2.2.1.11. Pass count (internal use only)
    - 2.2.1.12. Group count (internal use only)
    - 2.2.1.13. Seq (internal use only)
    - 2.2.1.14. Where clause
    - 2.2.1.15. Order by clause
    - 2.2.1.16. Columns list
    - 2.2.1.17. Export mode
    - 2.2.1.18. Table data
    - 2.2.1.19. Source schema
    - 2.2.1.20. Source db link
    - 2.2.1.21. Target schema
    - 2.2.1.22. Target db link
    - 2.2.1.23. Target table name
    - 2.2.1.24. User column name
    - 2.2.1.25. Id relocation
      - 2.2.1.25.1. ~~Id shift value~~
      - 2.2.1.25.2. ~~Sequence name~~
    - 2.2.1.26. Batch size
    - 2.2.1.27. Tab seq
    - 2.2.1.28. Gen view name
    - 2.2.1.29. Pre gen code
    - 2.2.1.30. Post gen code
  - 2.2.2. Operations
    - 2.2.2.1. Include tables
    - 2.2.2.2. Exclude tables
    - 2.2.2.3. Update table properties
    - 2.2.2.4. Count table records

- 2.3. Constraints
  - 2.3.1. Properties
    - 2.3.1.1. Constraint id
    - 2.3.1.2. Set id
    - 2.3.1.3. Constraint name
    - 2.3.1.4. Source table name
    - 2.3.1.5. Destination table name
    - 2.3.1.6. Cardinality
    - 2.3.1.7. Extract type
    - 2.3.1.8. Deferred
    - 2.3.1.9. Percentage
    - 2.3.1.10. Row limit
    - 2.3.1.11. Min rows
    - 2.3.1.12. Max rows
    - 2.3.1.13. Level count
    - 2.3.1.14. Source count
    - 2.3.1.15. Extract count
    - 2.3.1.16. Where clause
    - 2.3.1.17. Order by clause
    - 2.3.1.18. Join clause
    - 2.3.1.19. Master/detail cardinality ok
    - 2.3.1.20. Master/detail optionality ok
    - 2.3.1.21. Master/detail uid ok
    - 2.3.1.22. Batch size
    - 2.3.1.23. Con seq
    - 2.3.1.24. Gen view name
    - 2.3.1.25. Pre gen code
    - 2.3.1.26. Post gen code
    - 2.3.1.27. Source filter
  - 2.3.2. Operations
    - 2.3.2.1. Update constraint properties
    - 2.3.2.2. Include referential constraints
    - 2.3.2.3. Include master detail constraints
    - 2.3.2.4. Exclude constraints
    - 2.3.2.5. Detect true master detail constraints
- 2.4. Records
  - 2.4.1. Properties
    - 2.4.1.1. Rec id
    - 2.4.1.2. Table id
    - 2.4.1.3. Constraint id
    - 2.4.1.4. Record rowid

- 2.4.1.5. Shuffled rowid 1, 2 and 3
  - 2.4.1.6. Source rowid
  - 2.4.1.7. Sequence
  - 2.4.1.8. Pass count
  - 2.4.1.9. User name
  - 2.4.1.10. Undo timestamp
  - 2.4.1.11. Operation
  - 2.4.1.12. Remark
  - 2.4.1.13. Deleted flag
  - 2.4.1.14. Record data
  - 2.4.1.15. Record old data
- 2.4.2. Operations
  - 2.4.2.1. Extract data set rowids
  - 2.4.2.2. Delete data set rowids
  - 2.4.2.3. Shuffle records
- 2.5. Identifiers
  - 2.5.1. Properties
    - 2.5.1.1. Mask Id
    - 2.5.1.2. Old Id
    - 2.5.1.3. New Id
  - 2.5.2. Operations
    - 2.5.2.1. Generate identifiers
    - 2.5.2.2. Delete identifiers
- 2.6. Tokens
  - 2.6.1. Properties
    - 2.6.1.1. Mask id
    - 2.6.1.2. Token
    - 2.6.1.3. Value
  - 2.6.2. Operations
    - 2.6.2.1. Generate\_tokens
    - 2.6.2.2. Get\_value\_from\_token
    - 2.6.2.3. Get\_token\_from\_value
    - 2.6.2.4. Set\_token\_for\_value
- 2.7. Patterns
  - 2.7.1. Properties
    - 2.7.1.1. Pattern id
    - 2.7.1.2. Pattern name
    - 2.7.1.3. Pattern category
    - 2.7.1.4. Column data type
    - 2.7.1.5. Column name pattern
    - 2.7.1.6. Column comment pattern

- 2.7.1.7. Column data pattern
  - 2.7.1.8. Column data set name
  - 2.7.1.9. Column data hit percentage
  - 2.7.1.10. Column data hit count
  - 2.7.1.11. Logical operator
  - 2.7.1.12. Remarks
  - 2.7.1.13. System flag
  - 2.7.1.14. Disabled flag
  - 2.7.1.15. Mask type
  - 2.7.1.16. Mask parameters
  - 2.7.2. Operations
    - 2.7.2.1. Insert pattern
    - 2.7.2.2. Update pattern properties
    - 2.7.2.3. Delete pattern
- 2.8. Masks
  - 2.8.1. Properties
    - 2.8.1.1. Mask identifier
    - 2.8.1.2. Table name
    - 2.8.1.3. Column name
    - 2.8.1.4. Sensitive flag
    - 2.8.1.5. Disabled flag
    - 2.8.1.6. Locked flag
    - 2.8.1.7. Mask type
    - 2.8.1.8. Shuffle group
    - 2.8.1.9. Partition bitmap
    - 2.8.1.10. Mask parameters
    - 2.8.1.11. Mask options
    - 2.8.1.12. Pattern category
    - 2.8.1.13. Pattern name
    - 2.8.1.14. Remarks
    - 2.8.1.15. Values sample
  - 2.8.2. Operations
    - 2.8.2.1. Insert mask
    - 2.8.2.2. Update mask properties
    - 2.8.2.3. Delete mask
    - 2.8.2.4. Propagate pk masking
- 2.9. Table columns
  - 2.9.1. Properties
    - 2.9.1.1. Table id
    - 2.9.1.2. Table name
    - 2.9.1.3. Column name

- 2.9.1.4. Col seq
  - 2.9.1.5. Gen type
  - 2.9.1.6. Params
  - 2.9.1.7. Null value pct
  - 2.9.1.8. Null value condition
- 2.9.2. Operations
  - 2.9.2.1. Insert table columns
  - 2.9.2.2. Update table column properties
  - 2.9.2.3. Clone\_table\_column\_properties
- 2.10. Data set extraction and/or transportation
  - 2.10.1. Operations
    - 2.10.1.1. Configure data set
    - 2.10.1.2. Extract rowids
    - 2.10.1.3. Preview data set
    - 2.10.1.4. Transport data set (formerly handle data set)
    - 2.10.1.5. Security policies
    - 2.10.1.6. Forcing column values
- 2.11. Change data capture
  - 2.11.1. Properties
  - 2.11.2. Operations
    - 2.11.2.1. Create triggers
    - 2.11.2.2. Drop triggers
    - 2.11.2.3. Delete captured operations
    - 2.11.2.4. Undo captured operations
    - 2.11.2.5. Redo captured operations
    - 2.11.2.6. Rollback captured data set
    - 2.11.2.7. Rollforward captured data set
    - 2.11.2.8. Create capture forwarding job
    - 2.11.2.9. Drop capture forwarding job
    - 2.11.2.10. Capture forwarding job exists
    - 2.11.2.11. Generate captured data set script
- 2.12. Sensitive Data Discovery
  - 2.12.1. Operations
    - 2.12.1.1. Discover sensitive data
- 2.13. On-the-fly Data Masking
  - 2.13.1. Operations
    - 2.13.1.1. Propagate pk masking
    - 2.13.1.2. Generate identifiers
    - 2.13.1.3. Shuffle records
    - 2.13.1.4. Generate\_tokens
    - 2.13.1.5. Mask data set

- 3. Technical Aspects
  - 3.1. Message Types
    - 3.1.1. Error (E)
    - 3.1.2. Warning (W)
    - 3.1.3. Information (I)
    - 3.1.4. Debug (D)
    - 3.1.5. SQL statement (S)
    - 3.1.6. ROWCOUNT (R)
  - 3.2. Message masking
  - 3.3. Test mode
  - 3.4. Masking mode
  - 3.5. Encryption of tokenized values
  - 3.6. Output
  - 3.7. Global variables
    - 3.7.1. g\_test\_mode
    - 3.7.2. g\_mask\_data
    - 3.7.3. g\_encrypt\_tokenized\_values
    - 3.7.4. g\_show\_time
    - 3.7.5. g\_time\_mask
    - 3.7.6. g\_timestamp\_mask
    - 3.7.7. g\_msg\_mask
    - 3.7.8. g\_owner
    - 3.7.9. g\_alias\_like\_pattern
    - 3.7.10. g\_alias\_replace\_pattern
    - 3.7.11. g\_alias\_max\_length
    - 3.7.12. g\_alias\_constraint\_type
    - 3.7.13. g\_capture\_job\_name
    - 3.7.14. g\_capture\_job\_delta
    - 3.7.15. g\_xml\_batchsize
    - 3.7.16. g\_xml\_commitbatch
    - 3.7.17. g\_xml\_dateformat
- 4. Frequent errors
  - 4.1. Unsupported data type for column
  - 4.2. Maximum number of iterations exceeded
  - 4.3. Loop in table dependencies detected
  - 4.4. Infinite loop detected in xxx
  - 4.5. Missing join clause for constraint #id
  - 4.6. Row limit not compatible with full extract
  - 4.7. Table xxx has no primary key
- 5. GRAPHVIZ
  - 5.1. Description

- 5.2. Invocation
- 5.3. Visualisation
- 5.4. Graph examples
  - 5.4.1. Data generation (GEN) – without columns
  - 5.4.2. Data generation (GEN) – with all columns
  - 5.4.3. Data subsetting (SUB) – without columns
  - 5.4.4. Data subsetting (SUB) – with all columns
  - 5.4.5. Sensitive Data Discovery – with sensitive and masked columns
  - 5.4.6. Data Structure Diagram
- 6. DEGPL
  - 6.1. Language description
    - 6.1.1. Properties
    - 6.1.2. Data set definition
    - 6.1.3. Extraction path
    - 6.1.4. Extraction direction
    - 6.1.5. Extraction types
    - 6.1.6. Cardinality
    - 6.1.7. All arrow types
    - 6.1.8. Constraint name
    - 6.1.9. Constraint properties
    - 6.1.10. Arrow shortcuts
    - 6.1.11. Table identifiers
    - 6.1.12. Table extract type
    - 6.1.13. Table properties
    - 6.1.14. Column properties
    - 6.1.15. Mask properties
    - 6.1.16. Wildcards
    - 6.1.17. Recursion
    - 6.1.18. Scope
    - 6.1.19. Default values
  - 6.2. Invocation
  - 6.3. Recommendations
  - 6.4. Example
  - 6.5. EBNF
  - 6.6. Syntax diagram
- 7. Format-Preserving Encryption (FPE)
  - 7.1. Introduction
  - 7.2. RSR Algorithm
    - 7.2.1. Overview
    - 7.2.2. Substitution tables
    - 7.2.3. Pseudo-Random Number Generators



- 7.2.4. Encryption of numbers
- 7.2.5. Encryption of strings
- 7.2.6. Encryption of dates
- 7.2.7. Encryption of integers
- 7.2.8. Decryption
- 7.3. FF3 Algorithm
  - 7.3.1. Overview
  - 7.3.2. Encryption of numbers
  - 7.3.3. Encryption of strings
  - 7.3.4. Encryption of dates
  - 7.3.5. Encryption of integers
- 7.4. Comparison
  - 7.4.1. Limitations
    - 7.4.1.1. RSR
    - 7.4.1.2. FF3
  - 7.4.2. Performance
    - 7.4.2.1. RSR
    - 7.4.2.2. FF3
  - 7.4.3. Approval by Certification Authorities
    - 7.4.3.1. RSR
    - 7.4.3.2. FF3
  - 7.4.4. Summary
- 7.5. APIs
  - 7.5.1. Parameters common to all methods
  - 7.5.2. Default settings
  - 7.5.3. Encryption / decryption of numbers
  - 7.5.4. Encryption / decryption of strings
  - 7.5.5. Encryption / decryption of dates
  - 7.5.6. Encryption / decryption of integers

# 1. Internal Data Model

The data set utility has no GUI and is entirely operated by calling APIs and querying its internal tables, reason why a good understanding of its internal data model is crucial:

## 1.0.1. Data Model Diagram

## 1.0.2. Configuration tables

`DATA SETS` are uniquely identified by their `SET_ID` (`SET_PK`). Their `SET_NAME` must also be unique (`SET_UK`). `DATA SETS` are defined for one or more `TABLES` which are uniquely identified by a `TABLE_ID` (`TAB_PK`). `TABLE_NAME` must also be unique within a data set (`TAB_UK`).

`DATA SETS` may have one or more foreign key `CONSTRAINTS` that will be walked through during the extraction or generation process. They are uniquely identified by a `CON_ID` (`CON_PK`). Foreign key `CONSTRAINTS` link source `TABLES` (`CON_TAB_SRC_FK`) with destination `TABLES` (`CON_TAB_DST_FK`).

A constraint can be walked through in both directions in which case it will be registered twice with 2 different cardinalities (1-N for master/detail and N-1 for referential integrity). The pair of columns `CONSTRAINT_NAME` and `CARDINALITY` is therefore unique within a given data set (`DS_CON_UK`).

Sensitive data types are discovered using search `PATTERNS` and masked using `MASKS`. As data discovery and data masking are not dependent on a particular data set, they have no link with `DATA SETS`. See *Sensitive Data Discovery and Data Masking User's guide for more information*.

For synthetic data generation, `TAB COLUMNS` define how each column must be generated. See *Synthetic Data Generation User's guide for more information*.

### 1.0.3. Transactional tables

For tables that are generated or partially extracted, `ROWID` of generated or extracted records are stored in `RECORDS`. This table is also used to shuffle records when masking data. `TABLES` may have one or several `RECORDS`, identified by a `REC_ID`. Records are linked to `CONSTRAINTS` (`REC_CON_FK`) when they are extracted or generated in the context of a master / detail relationship.

The masking of columns (`DS_MASKS`) may require the generation of `TOKENS` or the generation (via an Oracle sequence or an in-memory sequence) of new `IDENTIFIERS` (mapped with old ones). See *Sensitive Data Discovery and Data Masking User's guide for more information*.

`OUTPUT` is used to persist error, warning, and/or information messages as well as SQL scripts.

## 2. Concepts and APIs

This section explains in detail the various concepts of the data model, their properties and the operations that can be performed on them.

### 2.1. Data sets

A data set is a coherent set of data that you want to operate i.e., subset, extract, mask, transport, generate, delete, backup, restore, or use as reference (look-up).

A data set definition is the set of rules defining how to operate the data set.

By abuse of language, one term is sometimes used in place of the other but it's important to make the distinction when it comes to using the APIs. Indeed, deleting a data set definition (i.e., configuration data for this tool) is much different than deleting a data set itself (i.e., business data).

## 2.1.1. Properties

Here follow the properties of data sets and their usage. Most of them can be changed by calling the `set_data_set_def_properties()` procedure.

### 2.1.1.1. Set id

A data set is identified by a unique identifier (positive integer) generated from the `DS_SET_SEQ` Oracle sequence. Most operations require the `set_id` to be passed in parameter.

### 2.1.1.2. Set name

Each data set is also identified by a name that must be unique. The name is case sensitive.

### 2.1.1.3. Set type

A data set has one the following types:

- SUB: for sub-setting purpose.
- GEN: for synthetic data generation purpose.
- CAP: for change data capture.
- CSV: look-up data set for data discovery and/or data masking purpose (CSV format).
- SQL: look-up data set for data discovery and/or data masking purpose (SQL statement).
- JSON: reserved for future use.
- XML: reserved for future use.

### 2.1.1.4. System flag

Indicates that the data set is pre-defined i.e., it is created during the installation of the tool and should not be modified in principle.

### 2.1.1.5. Disabled flag

Indicates that the data set is temporarily or indefinitely disabled i.e., it will not be used at all during various operations.

### 2.1.1.6. Visible flag

Only data sets that are marked as visible will be shown when you create views or security policies without having specified a `set_id`. Allowed values are Y (default) or N.

### **2.1.1.7. Capture flag**

The capture flag allows you to enable or disable the capture of DML operations executed on the tables of a data set. Allowed values are Y (default) or N.

### **2.1.1.8. Capture mode**

Data capture allows you to capture DML operations and replicate them synchronously or asynchronously to another schema or database via a database link.

Possible values are:

- NONE: operations and data are just stored as XML in `DS_RECORDS` tables without replication (default value).
- SYNC: operations are immediately replicated to another schema or database (via db link); therefore, they are not stored.
- ASYN: operations are stored temporarily as XML and replicated asynchronously via a dbms job.

### **2.1.1.9. Capture sequence (internal use only)**

Captured records have a sequence number whose last value is stored in the data set definition. In a nutshell, it's the implementation of a record sequence per data set.

### **2.1.1.10. Line separator character**

Character(s) used to separate lines in the CSV data set (LF or CRLF).

Default character in Oracle is Line Feed (LF = ASCII (10)).

### **2.1.1.11. Column separator character**

Character used to separate columns in the CSV data set (TAB, semi-colon, etc.).

Default character is TAB = ASCII (9), compatible with copy/paste from Excel.

### **2.1.1.12. Left separator character**

Left character used to delimit strings (e.g., single/double quote, open parenthesis/bracket, etc.) in the CSV data set.

Left and right separator characters go by pair (e.g., " , "" , ( , { , []); none by default.

### **2.1.1.13. Right separator character**

Right character used to delimit strings (e.g., single/double quote, open parenthesis/bracket, etc.) in the CSV data set.

Left and right separator characters go by pair (e.g., " , "" , ( , { , []); none by default.

#### **2.1.1.14. Column names row**

Row number of the header row containing column names (1 = first row by default) in the CSV data set.

#### **2.1.1.15. Column types row**

Row number of the header row containing column types (none by default i.e., no column data type is expected) in the CSV data set.

#### **2.1.1.16. Data row**

Row number of the first row containing the data (none by default i.e., after header rows) in the CSV data set. This allows to skip some rows before starting to import the data.

#### **2.1.1.17. Params**

Parameters of the data set, which depends on its type:

- SUB: not applicable (should be NULL)
- GEN: not applicable (should be NULL)
- CSV: CSV data (e.g., copied and pasted from Excel)
- SQL: SQL statement

### **2.1.2. Operations**

Data set definitions can be created, manipulated, and deleted by calling procedures and/or functions. For some operations, you have the choice between using a procedure (if you are not interested by the returned value) or a function.

#### **2.1.2.1. Create**

To create a data set definition, call the `create_data_set_def()` function with a unique name as parameter. The internal identifier of the created data set is returned. An exception is raised if the name is already used by an existing data set. Other properties (described above) can be specified upon creation. They can also be updated at a later stage.

#### **2.1.2.2. Create or replace**

To create or replace a data set definition, call the `create_or_replace_data_set_def()` function with a unique name as parameter. If the data set already exists, it is cleared (see below) and its `set_id` remain unchanged. If it does not exist, it is created. The `set_id` is returned in both cases. Other properties (described above) can be specified upon creation. They can also be updated at a later stage.

#### **2.1.2.3. Clone**

To clone an existing data set definition, call the `clone_data_set_def()` function and pass the `set_id` and a new name in parameter. The function will return the `set_id` of the cloned data set. A few other properties can also be passed in parameter. Records and identifiers which are the results of an extraction are not copied over.

#### **2.1.2.4. Delete**

To delete an existing data set definition, call the `delete_data_set_def()` and pass the `set_id` in parameter. As a result, the data set definition will be completely dropped. Any job created for the capture forwarding functionality will also be dropped.

#### **2.1.2.5. Get last**

To get the `set_id` of the last created data set definition, call the `get_last_data_set_def()` function. Note that if this data set definition has been deleted in the meantime, the highest existing `set_id` will be returned instead. NULL is returned if no data set is found.

#### **2.1.2.6. Get by name**

To get the `set_id` of a named data set, call the `get_data_set_def_by_name()` function and pass the data set name in parameter. The `set_id` of the named data set will be returned or NULL if not found.

#### **2.1.2.7. Clear**

To reset the definition of a data set while keeping it, call the `clear_data_set_def()` procedure and pass the `set_id` in parameter. This will delete tables, constraints, records, and identifiers while keeping the data set definition. Use this procedure when you want to restart your configuration from scratch while keeping the same data `set_id` and `set_name`.

#### **2.1.2.8. Update properties**

To update the properties of an existing data set, call the `update_data_set_def_properties()` and pass the data set id and the values of the properties that you want to change in parameter. By using named parameters, you can selectively modify only the desired properties.

## **2.2. Tables**

A data set may have one or several tables for which you may want to extract or generate records, or capture record changes.

### **2.2.1. Properties**

Here follow the properties of data set tables and their usage. Some of them (but not all) can be changed by calling the `set_tables_properties()` procedure.

#### **2.2.1.1. Table Id**

This is the unique identifier of the table generated from the `DS_TAB_SEQ` sequence.

#### **2.2.1.2. Set Id**

This is the id of the data set to which the table belongs.

#### **2.2.1.3. Table Name**

This is the name of the table.

#### **2.2.1.4. Table Alias**

This is the alias of the table to be used when generating join queries. Table aliases are extracted from the primary or unique constraint names. When an alias cannot be extracted or if its length exceeds the maximum (5 characters by default), a default alias is generated following the pattern: `t<table_id>`. See section [3.7](#) on [Global variables](#) for more information on how to configure the patterns used for aliases extraction.

#### **2.2.1.5. Extract type**

Indicates how records from this table will be extracted or generated.

Allowed values are:

- B)ase extraction or generation: identifies driving tables i.e., tables that are the starting point of the extraction or generation process;
- F)ull extraction: all records of this table will be extracted (used e.g., for reference tables);
- N)o extraction: no record will be extracted or generated for this table (used e.g., for reference tables);
- P)artial extraction or generation (default): records will be extracted or generated via a master/detail (1-N) relationship.

#### **2.2.1.6. Row limit**

This indicates how many records you want to extract from this table. Value must be strictly positive when specified.

If a percentage is also specified, the least of both values is taken.

If sorting clause is specified, records are sorted before extracting the first N records.

#### **2.2.1.7. Percentage**

This indicates the percentage of the total number of rows (source count) that you want to extract from this table. Value must be between 0 and 100 included when specified.

The number of rows to be extracted (N below) is computed as:  $\text{total count} \times \text{percentage} / 100$ .

If a row limit is also specified, the least of both values is taken.

If sorting clause is specified, records are sorted before extracting the first N records.

### **2.2.1.8. Row count**

This specifies how many records must be generated for this table. Value must be  $\geq 0$ . For hierarchic tables (e.g., an organisation chart or a product nomenclature), this property represents the number of root records (i.e., the number of trees in the forest).

### **2.2.1.9. Source count (internal use only)**

This is the number of records in the table either coming from the statistics (default), either computed via the `count_table_records()` procedure.

### **2.2.1.10. Extract count (internal use only)**

This is the number of records that will be extracted or generated for this table (set when calling the `extract_data_set_rowids()` or `generate_data_set()` procedure).

### **2.2.1.11. Pass count (internal use only)**

When several rounds of processing are needed (e.g., for tables having recursive relationships), this indicates the current round number.

### **2.2.1.12. Group count (internal use only)**

This is the number of times the table has been processed while extracting rowids. Tables that have recursive constraints (pig's ear) often need to be processed more than once.

### **2.2.1.13. Seq (internal use only)**

This number indicates the sequence in which tables must be processed to avoid foreign key violations when importing or generating data (tables with lowest sequence must be processed first). Several tables may share the same sequence meaning the processing order between them is meaningless. Sequencing of those tables can be forced using the `tab_seq` property (see below). Tables with sequence equal to 1 are those not referencing any other (e.g., reference tables).

### **2.2.1.14. Where clause**

Filter to be applied when extracting records from this table. When no filter is specified, the full table is extracted.



This filter is only applicable in the following circumstances:

- When extracting records from base table(s) i.e., for driving tables with a B extract type only;
- When capturing DML operations (independently of the extract type which is not used in this case).

This filter is simply ignored in all other circumstances (e.g., extract types other than B).

### **2.2.1.15. Order by clause**

This indicates the sorting order that should be applied when extracting records from this table. When no sorting order is specified, records are extracted in a random order (which depends on their physical storage).

Sorting is always performed before any extraction limitation is applied (percentage or row limit).

### **2.2.1.16. Columns list**

This is a comma-separated list of columns that you want to extract for this table. If none is specified, all columns are extracted (the \* wild card is then used). Instead of specifying columns that you want to include, you can also use a special syntax to indicate columns that you want to exclude:

\* BUT <comma-separated-list-of-columns-to-exclude> . Not including mandatory columns and those that are part of the primary key might generate errors.

### **2.2.1.17. Export mode**

Export mode indicates which operations should be performed for this table.

Its value must be one of following letters:

- I)nsert (default): source record is inserted in target schema.
- U)pdate: record is updated in the target schema.
- D)elete: record is deleted from the source schema.
- R)efresh (=UpSert): record is updated if existing and created otherwise (<=> UI).
- M)ove: record is inserted in the target schema then deleted from the source schema.

Note that deletions are performed in the source schema while other operations are performed in the target schema.

### **2.2.1.18. Table data**

This contains the XML data of fully extracted tables. XML of partially extracted tables is stored in the records ( DS\_RECORDS ).

### **2.2.1.19. Source schema**

Source schema indicates the schema where the source table (the table from which data will be extracted) is stored. This is used when the data set utility is installed in a different schema than where the data are located.

#### **2.2.1.20. Source db link**

Source database link indicates the database link to be used when extracting data from the source table.

#### **2.2.1.21. Target schema**

Target schema indicates the schema where the target table (the table into which data will be transported or generated) is stored.

#### **2.2.1.22. Target db link**

Target database link indicates the database link to be used when transporting or generating data into the target table.

#### **2.2.1.23. Target table name**

This is the name of the target table (if its name is different than in the source schema).

#### **2.2.1.24. User column name**

User column name indicates the name of the column that must be used for filtering captured data changes. When specifying an audit column (e.g., `changed_by`), this gives the possibility to limit the capture to the changes made by a specific user.

#### **2.2.1.25. Id relocation**

When transporting a data set from one environment to another, it's likely that unique identifiers will collide if target tables are not empty. To avoid collisions, this tool can relocate your unique identifiers by shifting their value, or by generating new values via an Oracle sequence belonging to the target schema. This is possible only if the primary key is made up of a single numeric column.

The code implementing this functionality has been dropped in v23.3 in favour of the following data masking techniques:

- SQL masking to shift identifiers by a given value, specify the following SQL expression:  
`<column-name> + <shift-value> .`
- SEQUENCE masking to generate new identifiers based on an Oracle or in-memory sequence.

##### **2.2.1.25.1. Id shift value**

~~This is the positive or negative quantity by which each unique identifier must be shifted (NULL or zero means no shifting).~~

*Deprecated and replaced with data masking.*

#### **2.2.1.25.2. Sequence name**

~~This is the name of a sequence existing in the target schema that must be used to generate new unique identifiers (NULL means no relocation via a sequence). This property is ignored if a shift value is specified (it doesn't make sense to combine both methods). Also note that new identifiers cannot be previewed as they will only be generated during transportation to the target schema.~~

~~To use a local Oracle sequence, an in-memory sequence, or a remote sequence accessible through a database link, please rather use the SEQUENCE data masking feature introduced in v23.2.~~

*Deprecated and replaced with data masking.*

#### **2.2.1.26. Batch size**

When generating data, this indicates the size of the batch used for inserting and/or updating data in bulk. When not set, default value is 101.

#### **2.2.1.27. Tab seq**

This is used to specify a sub-sequence of processing for tables that have the same processing sequence (see seq property above).

#### **2.2.1.28. Gen view name**

This is the name of a so-called generation view that is used to generate base records (instead of using the dual system view). It is typically used to generate historical tables or time series.

#### **2.2.1.29. Pre gen code**

This is the PL/SQL code that is executed before generating data in this table.

#### **2.2.1.30. Post gen code**

This is the PL/SQL code that is executed after having generated data in this table. It is typically used to compute denormalized fields.

### **2.2.2. Operations**

Here follow the operations that you can perform on data set tables.

#### **2.2.2.1. Include tables**

You can add one or several tables to a given data set by calling the `include_tables()` procedure and pass the `set_id` in parameter. Some of the table properties (described above) can also be passed as parameter.

Most important parameters are:

- `p_set_id` : id of the data set.
- `p_table_name` : name pattern of the tables that you want to include (NULL for all).
- `p_extract_type` : B)ase table, F)ull, P)artial (default), N)o extraction or generation.
- `p_recursive_level` : allow you to recursively add detail tables up to the level specified (NULL means no recursion (default), 0 for all levels, n for n levels).
- `p_det_table_name` : name pattern of the detail tables that you want to include (NULL for all).
- `p_optimize_flag` : when set to Y, detail tables having less records than their master will not be included in the data set.

Patterns are evaluated either via the `REGEXP_LIKE` function either via the `LIKE` function, depending on whether some regular expression special characters have been found or not in the pattern (^ for beginning of name, \$ for end of name, . for a single character, \* for several characters, parenthesis, brackets, or rounded brackets). When no regular expression is given, standard Oracle patterns are allowed ( \_ for a single character, % for several).

One-to-many constraints that have been walked through while discovering detail tables are also added to the data set with the same extraction type.

#### **2.2.2.2. Exclude tables**

You can remove one or several tables from a data set by calling the `exclude_tables()` procedure with the following parameters:

- `p_set_id` : id of the data set;
- `p_table_name` : name pattern of the tables that you want to exclude (NULL for all).

The pattern is either a regular expression, either a name possibly containing standard Oracle wildcards ( \_ and % ).

Constraints that are linked to tables that have been removed are also removed from the data set.

#### **2.2.2.3. Update table properties**

To change the properties of one or several tables of a data set, call the `update_table_properties()` procedure. Pass the `set_id` of the data set and the properties that you want to change in parameter.

#### **2.2.2.4. Count table records**

The number of records in a table is by default taken from database statistics which are not always up to date. You can force records to be counted for all tables of a data set by calling the `count_table_records()` and pass the `set_id` in parameter.

## 2.3. Constraints

A data set may have one or more constraints (foreign keys) that will be followed or walked through when extracting or generating records. They are created from the `ALL_CONSTRAINTS` Oracle data dictionary view. As a foreign key can be walked through in both directions (1-N for discovering detail records and N-1 for ensuring referential integrity), it can be stored twice in `CONSTRAINTS` with one of the two cardinalities.

### 2.3.1. Properties

Here follow the properties of data set constraints. Some of them can be changed by using the `set_constraints_properties()` procedure.

#### 2.3.1.1. Constraint id

This is the unique identifier of a constraint generated from the `DS_CON_SEQ` sequence.

#### 2.3.1.2. Set id

Identifier of the set this constraint is belonging to.

#### 2.3.1.3. Constraint name

This is the name of the constraint as reported by the `ALL_CONSTRAINTS` view. The constraint name together with its cardinality must be unique within the data set.

#### 2.3.1.4. Source table name

This is the name of the table from which the constraint is walked through.

#### 2.3.1.5. Destination table name

This is the name of the table being reached when the constraint is walked through.

#### 2.3.1.6. Cardinality

This indicates the cardinality of the constraint when walked through from the source table to the destination table. Valid values are 1-N for one-to-many and N-1 for many-to-one.

#### 2.3.1.7. Extract type

This indicates whether the constraint must be used and how:

- B)ase: the 1-N constraint is used to discover details of base or master records;
- P)artial: the N-1 constraint is used to retrieved referenced data to ensure integrity.
- N)one: the constraint must not be used.

- Extraction type and cardinality are tightly linked.

### **2.3.1.8. Deferred**

This property indicates whether a referential constraint is checked immediately or if it is deferred. When a loop is discovered in the table dependencies, at least one constraint must be declared as deferred in order to break the loop. This property is initialized from the DEFERRED column of the `ALL_CONSTRAINTS` view. Possible values are DEFERRED or IMMEDIATE.

### **2.3.1.9. Percentage**

This property, valid for 1-N constraints only, indicates the percentage of detail records that must be extracted while walking through this 1-N constraint. The percentage is applied to the number of records found in the given relationship (counted on-the-fly and not stored). If a row limit is also specified, the least of the two values is taken.

### **2.3.1.10. Row limit**

This property, applicable to 1-N constraints only, indicates the maximum number of detail records that can be retrieved for the constraint. If a row limit is also specified, the least of the two values is taken.

### **2.3.1.11. Min rows**

This property indicates the minimum number of child records that must be generated for each parent record when following this mater/detail constraint. Applicable only to 1-N constraints. No value is equivalent to 0.

### **2.3.1.12. Max rows**

This property indicates the maximum number of child records that must be generated for each parent record when following this mater/detail constraint. Applicable only to 1-N constraints. When no value is specified, the maximum defaults to the minimum.

For reach parent record, the tool will generate a random number of detail records comprised between the minimum and the maximum.

### **2.3.1.13. Level count**

This property indicates the exact number of levels to be generated for hierarchical tables. Applicable to 1-N constraints only. No hierarchy is generated if the value for this property is not greater or equal to 2. The number of root records is specified at the table level.

### **2.3.1.14. Source count**

This property indicates the number of records in the destination table of the constraint. This is for information only i.e., not used by the tool itself.

### **2.3.1.15. Extract count**

This property indicates how many records will be extracted by walking through the constraint. When a same record is found several times via different constraints, it is counted once for each constraint but only once for the table. The record count of a table is therefore not necessarily equal to the sum of the record counts of the sourcing constraints (it's in fact less or equal).

### **2.3.1.16. Where clause**

This property, applicable to 1-N constraints only, indicates the filter that you want to apply to the destination table when retrieving detail records via the constraint.

### **2.3.1.17. Order by clause**

This property, applicable to 1-N constraints only, indicates a sorting order that you want to apply when retrieving a subset of the detail records (via the percentage and/or the limit properties).

### **2.3.1.18. Join clause**

This property contains the join clause that will be used when joining source and target tables of the constraint. Table columns are prefixed with their respective table alias.

### **2.3.1.19. Master/detail cardinality ok**

This property, applicable to 1-N constraints only, indicates whether the child table contains (Y/N) more records than the parent table. It helps determining whether a 1-N constraint represents a true master/detail relationship or not.

### **2.3.1.20. Master/detail optionality ok**

This property, applicable to 1-N constraints only, indicates whether the foreign key is mandatory or not (Y/N). It helps determining whether a 1-N constraint represents a true master/detail relationship or not.

### **2.3.1.21. Master/detail uid ok**

This property, applicable to 1-N constraints only, indicates whether the foreign key is included in the unique id (pk or uk) of the child table. It helps determining whether a 1-N constraint represents a true master/detail relationship or not.

### **2.3.1.22. Batch size**

When generating data via this constraints, this property indicates the size of the batch used for inserting and/or updating data in bulk. When not set, default value is 101.

### **2.3.1.23. Con seq**

This is used to specify an order of processing between the constraints of a same table. When not set, constraints are processed sorted alphabetically on their name.

#### **2.3.1.24. Gen view name**

This is the name of a so-called generation view that is used to generate child records (instead of using the dual system view). It is typically used to generate historical tables or time series.

#### **2.3.1.25. Pre gen code**

This is the PL/SQL code that is executed before generating data via this constraint.

#### **2.3.1.26. Post gen code**

This is the PL/SQL code that is executed after having generated data via this constraint. It is typically used to compute denormalized fields.

#### **2.3.1.27. Source filter**

This property, applicable to 1-N constraints only, is used to filter master records for which child records must be generated. While `where_clause` is a filter applied to the destination table (when extracting records), `source_filter` is applied to the source table (when generating records).

### **2.3.2. Operations**

Here follow the operations that you can perform on data set constraints.

#### **2.3.2.1. Update constraint properties**

You can change constraint properties by calling the `update_constraint_properties()` procedure and passing the set id, constraint name and cardinality in parameter, as well as the properties that you want to change. It is advised to use named parameters rather than positional parameters. To change the properties of several constraints, a pattern can be passed for the constraint name.

#### **2.3.2.2. Include referential constraints**

You can add all N-1 referential constraints needed to ensure the integrity of your data set by calling the `include_referential_cons()` procedure and pass the data set id in parameter. You can also specify a pattern to limit the process to some tables and/or constraints. Identified constraints will have a P extraction type and a N-1 cardinality. Destination tables that are not in the data set yet will also be included with a P extraction type.

#### **2.3.2.3. Include master detail constraints**

You can add 1-N constraints to your data set by calling the `include_master_detail_cons()` procedure and passing the data set id in parameter. You can also specify a pattern to limit the



process to some master tables and/or detail tables and/or constraints. Some additional properties (applicable to 1-N constraints only) like where clause, percentage and row limit can also be set at the same time. Extraction type and cardinality should be respectively set to B for base extraction and 1-N for master/detail. Detail tables that are not in the data set will also be included automatically.

#### **2.3.2.4. Exclude constraints**

You can delete constraints from your data set by calling the `exclude_constraints()` procedure and passing in parameter the data set id and optionally a pattern for constraint name and cardinality.

Instead of removing a constraint from a data set, you can also change its extract type to N (=No extraction), meaning that it will be ignored in any processing.

#### **2.3.2.5. Detect true master detail constraints**

One-to-many (1-N) constraints discovered by the `include_master_detail_cons()` procedure do not always represent true master/detail relationships. The columns of the `DS_CONSTRAINTS` table prefixed with MD (which stands for Master/Detail) can be set - by calling the `detect_true_master_detail_cons()` procedure - to help determining the likelihood that a constraint implements a true master/detail relationship.

True master/detail relationships are also called identifying relationships and have the following characteristics:

- A child record cannot exist without its parent record i.e., the foreign key of the child table is mandatory (meaning that its columns are mandatory); constraints fulfilling this condition have their `MD_OPTIONALITY_OK` flag set to Y.
- A child record is identified by its parent record plus some additional columns i.e., the foreign key columns of the child table include all primary (or unique) key columns of the parent table; constraints fulfilling this condition have their `MD_UID_OK` flag set to Y.

Non-identifying relationships (those that do not meet the two above conditions) should not be considered when extracting details. As an example, a foreign key towards a look-up or reference table is non-identifying (reference records can exist without necessarily being referenced) and should therefore not be used.

Furthermore, a child table is also expected to contain more records than its parent table (although this is not necessarily always the case, especially if many parents have no child); constraints fulfilling this condition have their `MD_CARDINALITY_OK` flag set to Y.

The `exclude_constraints()` procedure can be invoked to exclude constraints based on the 3 flag described above.

## **2.4. Records**

Data set records are used to identify records (via their ROWID) that will be extracted or that have been generated. XML generated as a result of an extraction or of a change data capture is also stored in `DS_RECORDS` .

## **2.4.1. Properties**

Here follow the properties of data set records and their usage. None of them can be changed by end-users.

### **2.4.1.1. Rec id**

Unique identifier of the record generated from the `DS_REC_SEQ` Oracle sequence.

### **2.4.1.2. Table id**

This property identifies the table which the record belongs to.

### **2.4.1.3. Constraint id**

This property identifies the constraint which led to the creation of this record. This property remains NULL for base records (records extracted from base tables) or records created in the context of the change data capture functionality.

### **2.4.1.4. Record rowid**

This is the rowid of records that will be extracted or that have been generated. The record rowid can be a physical rowid (in case of a standard table) or a logical rowid (in case of an IOT = Index Organized Table).

### **2.4.1.5. Shuffled rowid 1, 2 and 3**

When records shuffling is performed during data masking, shuffled rowids are stored in `DS_RECORDS` for shuffling group 1, 2 and 3.

### **2.4.1.6. Source rowid**

When a record has been identified via the walk-through of a 1-N or N-1 constraint, this property uniquely identifies the originating record (the record at the other side of the relationship) and is used for debugging purpose only.

### **2.4.1.7. Sequence**

This property gives the sequence of discovery of records and their dependencies. Base records extracted from base tables have a sequence number equal to 0. Detail records found via 1-N constraints have their sequence number set to the sequence number of their parent decreased by

one. On the contrary, when N-1 constraints are walked through, sequence number of referenced record is set to the sequence number of referencing record increased by one.

This sequence number can be used to explain the order in which records have been identified during the extraction of rowids. It can also be used to insert records in the destination schema in the right order i.e., in such a way that there is no need to disable foreign key constraints.

#### **2.4.1.8. Pass count**

Because of pig's ear or loops in the data model, records of a same table can be discovered at different stages or iterations. This property gives the iteration number during which the record has been discovered.

#### **2.4.1.9. User name**

This property indicates which user created a record created in the context of the change data capture functionality. It is subsequently used to selectively delete records belonging to a particular user.

#### **2.4.1.10. Undo timestamp**

This property gives the exact date and time (down to the millisecond) when a captured record was undone i.e., rolled back.

#### **2.4.1.11. Operation**

This property is used when capturing data changes only and indicates the type of operation that has been performed i.e., whether the record is the result of an insert, update or delete operation.

#### **2.4.1.12. Remark**

This property is used when extracting records only and gives more information about the primary key of the record and the foreign key for those resulting from a constraint walk-through. It is used automatically populated when the debug mode is activated and not set otherwise.

#### **2.4.1.13. Deleted flag**

This property indicates records that have been logically deleted while deleting duplicate records created during extraction of rowids. Logical deletion is performed only when debug mode is activated. Otherwise, records are physically deleted.

#### **2.4.1.14. Record data**

This CLOB contains the XML representation of any extracted or captured record. For captured records, it contains the new values of the record (as it was after the insert or update).

#### **2.4.1.15. Record old data**

This CLOB contains the XML representation of any captured record. It contains the old values of the record (as it was before the update or delete).

## 2.4.2. Operations

Here follow the operations that may be invoked to create or delete data set records. See also operations in relation with the change data capture functionality.

### 2.4.2.1. Extract data set rowids

To identify the records of a table that must be extracted, call the `extract_data_set_rowids()` procedure and pass the data `set_id` in parameter. This procedure will first extract base records and their details by walking through 1-N constraints. In a second step, it will extract records referenced by those already extracted so far by walking through N-1 constraints. This process will be repeated recursively on newly created records until no new record is found.

### 2.4.2.2. Delete data set rowids

To delete records that have been previously created, call the `delete_data_set_rowids()` procedure and pass the data set id in parameter. This procedure is invoked internally before any extraction to delete any record created during a previous run.

### 2.4.2.3. Shuffle records

To shuffle records for data masking purpose, invoke the `shuffle_records()` method with the `set_id` passed in parameter. Rowids after shuffling are stored in `shuffled_rowid_<n>` where `<n>` is a number between 1 and 3 that represents the shuffling group (maximum 3 per table).

## 2.5. Identifiers

Table `DS_IDENTIFIERS` is used when masking data to generate unique identifiers based on an Oracle or in-memory sequence (for data masking). This table stores the mapping between old and new values which is needed when propagating primary keys to foreign keys.

### 2.5.1. Properties

#### 2.5.1.1. Mask Id

This is the id of the mask applied to a table column.

#### 2.5.1.2. Old Id

This is the value of an identifier before its relocation.

### **2.5.1.3. New Id**

This is the value of an identifier after its relocation.

## **2.5.2. Operations**

### **2.5.2.1. Generate identifiers**

To generate identifiers based on a SEQUENCE masking, invoke the `generate_identifiers()` method with the `msk_id` as parameter. New identifiers and their mapping with old values will be created in the `DS_IDENTIFIERS` table. This procedure must be invoked before previewing or transporting the data set.

### **2.5.2.2. Delete identifiers**

To delete identifiers that have been generated, invoke the `delete_data_set_identifiers()` procedure with the `msk_id` as parameter.

## **2.6. Tokens**

When the tokenization technique is used, table `DS_TOKENS` contains the mapping between tokens and their original value.

### **2.6.1. Properties**

#### **2.6.1.1. Mask id**

The id of the mask applied to a table column.

#### **2.6.1.2. Token**

The token stored as clear text.

#### **2.6.1.3. Value**

The value (cipher value or clear value depending on whether encryption is enabled or not).

### **2.6.2. Operations**

#### **2.6.2.1. Generate\_tokens**

Generate tokens for tokenized columns of the given data set or for the whole schema.

#### **2.6.2.2. Get\_value\_from\_token**

Get the original value associated with a given token. The returned value is decrypted if encryption is enabled. Returns NULL when the token is NULL.

### **2.6.2.3. Get\_token\_from\_value**

Get the token associated with a given original value. If encryption is enabled, the value is encrypted before being searched in `DS_TOKENS` . Returns NULL when the value is NULL.

### **2.6.2.4. Set\_token\_for\_value**

Set the token for the given original value. If encryption is enabled, the value is encrypted before being searched in `DS_TOKENS` . Token record is updated if value is found and inserted otherwise. Token and value cannot be NULL.

## **2.7. Patterns**

Patterns allow to discover sensitive data types that must be masked or anonymized.

### **2.7.1. Properties**

#### **2.7.1.1. Pattern id**

An internal identifier generated from a sequence (for internal use only).

#### **2.7.1.2. Pattern name**

- A name which uniquely identifies a pattern.

#### **2.7.1.3. Pattern category**

- A category that logically groups patterns together (e.g., personal, banking, etc.)

#### **2.7.1.4. Column data type**

- A column data type (with an optional indication of its maximum length, precision, and/or scale) that columns must be compatible with. Supported data types are the following:
  - CHAR (compatible with VARCHAR2, NCHAR and NVARCHAR)
  - CLOB (compatible with NCLOB)
  - NUMBER (compatible with INTEGER, `BINARY_FLOAT` and `BINARY_DOUBLE` )
  - DATE
  - TIMESTAMP (compatible with `TIMESTAMP(n) WITH (or without) TIME ZONE`)
- CHAR accepts a maximum length and NUMBER a maximum precision and/or scale.

#### **2.7.1.5. Column name pattern**

- A column name pattern i.e., an Oracle regular expression that column names must match.

#### **2.7.1.6. Column comment pattern**

- A column comment pattern i.e., an Oracle regular expression that column comments must match.

#### **2.7.1.7. Column data pattern**

- A column value pattern i.e., an Oracle regular expression that column values must match.

#### **2.7.1.8. Column data set name**

- A CSV or SQL look-up data set name and column that column values must belong to. Format is `<data_set_name>.<column_name> .`

#### **2.7.1.9. Column data hit percentage**

- A minimum hit percentage that must be reached when searching in the data (column values). Default is 10.

#### **2.7.1.10. Column data hit count**

- A minimum hit count that must be reached when searching in the data (column values). Default is 2.

#### **2.7.1.11. Logical operator**

- A logical operator (OR or AND) to be applied between column name and comment search criteria on one hand, and column value criteria (pattern and look-up data set) on the other hand.

#### **2.7.1.12. Remarks**

- Some remarks or comments about the pattern.

#### **2.7.1.13. System flag**

- Indicates whether the pattern is pre-defined (vs user-created).

#### **2.7.1.14. Disabled flag**

- Indicates whether the pattern is enabled (N) or disabled (Y). Disabled patterns are not used during the data discovery process.

#### **2.7.1.15. Mask type**

The mask type applicable to the table columns matching this pattern.

### **2.7.1.16. Mask parameters**

The mask parameters applicable to the table columns matching this pattern.

## **2.7.2. Operations**

### **2.7.2.1. Insert pattern**

Create a pattern identified by a given name, with the other properties passed in parameter.

### **2.7.2.2. Update pattern properties**

Update properties of pattern(s) whose name is matching the one given in parameter.

### **2.7.2.3. Delete pattern**

Delete pattern(s) whose name is matching the one given in parameter.

## **2.8. Masks**

Masks define how sensitive data must be transformed when anonymizing them.

### **2.8.1. Properties**

#### **2.8.1.1. Mask identifier**

An identifier generated from a sequence (for internal use only).

#### **2.8.1.2. Table name**

The name of the table which the column belongs to.

#### **2.8.1.3. Column name**

The name of the column for which the mask is defined.

#### **2.8.1.4. Sensitive flag**

Indicates whether the column is sensitive i.e., whether it must be masked or not.

#### **2.8.1.5. Disabled flag**

Indicates whether the mask is enabled (N) or disabled (Y). When disabled, the mask will not be applied to the table column.

#### **2.8.1.6. Locked flag**



Indicates whether the mask is locked (Y) or not (N). Masks modified by end-users must be locked to prevent the sensitive data discovery process from overwriting user's changes.

### 2.8.1.7. Mask type

Indicates the type of masking that must be applied to the table column:

- SQL: original value will be replaced by the result of a SQL expression.
- SHUFFLE: values will be shuffled between records (of the same partition if any).
- SEQUENCE: value will be generated from an in-memory sequence, a local Oracle sequence, or a remote sequence accessible through a database link.
- TOKENIZE: value will be replaced with a token generated using a SQL expression.
- INHERIT: value will be inherited from a masked primary key through a foreign key (this inheritance is automatic to preserve referential integrity and can therefore not be defined by end-users).

To be noted that NULL values are not masked by default i.e., they remain NULL after masking. This behaviour can be changed for SQL masking via the `mask_null_values` option (see below).

### 2.8.1.8. Shuffle group

Indicates which shuffling group (numbered from 1 to 3) the column belongs to. Applicable only when masking type is SHUFFLE. All columns belonging to the same group will be shuffled together.

### 2.8.1.9. Partition bitmap

Indicates which shuffling partition(s) the column belongs to. Applicable only when masking type is SHUFFLE. As a column can belong to several partitions, a bitmap is used to indicate the groups (bit `n` represents group `n+1`). A separate shuffling is performed for each partition of the data i.e., records belonging to different partitions will never be mixed.

### 2.8.1.10. Mask parameters

Parameters of the mask, which depend on its type:

- SQL: a SQL expression that returns the masked value.
- SHUFFLE: not applicable.
- SEQUENCE: depending on the type of sequence:
  - Local Oracle sequence: `<sequence-name>`  
(existence checked against `ALL_SEQUENCES` ).
  - Remote Oracle sequence: `<sequence_name>@<db_link_name>`  
(existence not checked against Oracle data dictionary).
  - In-memory sequence: `[<sequence-name>] [START WITH <x>] [INCREMENT BY <y>]` (each part being optional, `x` and `y` default to 1).
- TOKENIZE: SQL expression used to generate tokens.

- INHERIT: name of the table column from which masked value is inherited via a foreign key (syntax: <table\_name>.<column\_name> ).

#### 2.8.1.11. Mask options

Comma separated list of masking options, which depend on the masking type



- SQL option:
  - mask\_null\_values=[true|**false**]: mask NULL values (no by default)?
- TOKENIZE options:
  - enforce\_uniqueness=[**true**|false]: can the same token be used for 2 different values? (default: true); uniqueness guarantees reversibility; should be set to true when the masked column is part of a PK/UK.
  - allow\_equal\_value=[true|**false**]: can a token be equal to its corresponding original value? (default: false); allowing the same value is against the principle of masking but necessary in some cases (e.g., encrypting small numbers may lead to equal cyphered and clear values).
  - encrypt\_tokenized\_values=[true|false]: must tokenized values be encrypted? Overrides the default behaviour dictated by the g\_encrypt\_tokenized\_values global variable.

#### 2.8.1.12. Pattern category

Category of the pattern that led to the creating of this mask during data discovery.

#### 2.8.1.13. Pattern name

Name of the pattern that led to the creating of this mask during data discovery.

#### 2.8.1.14. Remarks

Explains why the pattern was retained as the best matching or, in case of inheritance, which foreign key is used to propagate a masked primary key value.

#### 2.8.1.15. Values sample

Sample of matching values found during the data discovery process.

### 2.8.2. Operations

#### 2.8.2.1. Insert mask

Create mask(s) for table(s) and column(s) whose names are matching those given in parameters.

#### 2.8.2.2. Update mask properties

Update properties of mask(s) associated with table(s) and column(s) whose name are matching those given in parameter.

### **2.8.2.3. Delete mask**

Delete mask(s) for table(s) and column(s) whose name are matching those given in parameter.

### **2.8.2.4. Propagate pk masking**

To propagate the masking of primary keys to foreign keys, invoke the `propagate_pk_masking()` procedure with the `set_id` passed as parameter. This procedure must be invoked each time masking parameters of a primary key are changed.

As a result, the masking type of all foreign key columns is set to INHERIT. The names of the primary key table and column or the name of the in-memory sequence is stored in PARAMS.

## **2.9. Table columns**

The `DS_TAB_COLUMNS` table allows you to specify how to generate a value for each column. It is used exclusively for the synthetic data generation capability i.e., not for data sub-setting nor for data masking.

### **2.9.1. Properties**

#### **2.9.1.1. Table id**

Internal identifier of the table which the column belongs to.

#### **2.9.1.2. Table name**

Name of the table. This field is denormalized from `DS_TABLES` for a better readability.

#### **2.9.1.3. Column name**

Name of the column.

#### **2.9.1.4. Col seq**

Sequence in which columns must be processed. This is by default inherited from the `column_id` of the `USER_TAB_COLUMNS` data dictionary views.

#### **2.9.1.5. Gen type**

This property specifies how column value is generated:

- SQL: for a SQL expression

- SEQ: for an Oracle sequence
- FK: for foreign key columns

#### **2.9.1.6. Params**

Depending on the generation type, this property specifies:

- SQL: the SQL expression
- SEQ: the name of the Oracle sequence
- FK: the name of the foreign key

#### **2.9.1.7. Null value pct**

This property specifies the percentage of NULL values generated for this column (integer between 0 and 100). Applicable only to optional columns. Can be combined with null value condition.

#### **2.9.1.8. Null value condition**

This property allows to force the generation of a NULL value when the condition is fulfilled. Applicable only to optional columns. Can be combined with null value pct.

### **2.9.2. Operations**

#### **2.9.2.1. Insert table columns**

Columns are not automatically created when tables are included in a data set definition. Invoke the `insert_table_columns()` method to create all columns for tables matching the pattern passed in parameter. Some of the column properties explained above can also be passed in parameter.

The generation type ( `gen_type` ) is automatically set to FK for columns involved in a foreign and the foreign key name is stored in PARAMS.

When a SQL mask exists for a column (because of a sensitive data discovery or a manual encoding), the associated SQL expression is reused and stored in PARAMS if and only if it invokes a random function (other types of functions like obfuscate, mask or encrypt cannot be used as they require an existing value as input).

For mandatory columns, `null_value_pct` is set to 0 and `null_value_condition` is set to NULL. For optional columns, these properties are set to the value specified in parameter (NULL if not passed).

#### **2.9.2.2. Update table column properties**

Invoke the `update_table_column_properties()` method to update the properties of columns existing in `DS_TAB_COLUMNS` (see properties described above).

### 2.9.2.3. Clone\_table\_column\_properties

Invoke the `clone_table_column_properties()` method to copy column properties from one data set to another. Records in `DS_TAB_COLUMNS` of the target data set are updated but never created.

## 2.10. Data set extraction and/or transportation

The major steps for extracting and transporting a data set are the following:

1. Configure the data set (tables and constraints)
2. Extract rowids (base records first then by walking through constraints)
3. Pre-view data set (via ad-hoc views)
4. Extract and/or transport data set

### 2.10.1. Operations

This section describes the operations that can be performed on a data set once it has been configured i.e., once tables and constraints have been included in its definition.

#### 2.10.1.1. Configure data set

Data set can be created and configured using the following procedures and/or functions (already described earlier) of the core package ( `DS_UTILITY_KRN` ):

- `create_data_set()` Or `create_or_replace_data_set()`
- `include_tables()`
- `include_master_detail_cons()`
- `include_referential_cons()`
- `exclude_tables()`
- `exclude_constraints()`

Alternatively, or as a complement, the DEGPL language can be used by invoking the following procedure of the extension package ( `DS_UTILITY_EXT` ):

- `execute_degpl()`

See chapter [6](#) for a detailed description of the EDGPL language.

#### 2.10.1.2. Extract rowids

To identify precisely the rows that make your data set, call the `extract_data_set_rowids()` procedure and pass the data set id in parameter. Records are created with the rowid of identified rows. They act as pointers to the data of the data set.

#### 2.10.1.3. Preview data set

To preview a data set, create (then drop when finished) views by calling `create_views()` (then `drop_views()`) and pass the data set id in parameter. View names are generated from table names by potentially adding a suffix and/or prefix and removing a prefix from table names. Views can be created for the whole schema or only for tables in the data set. It is also possible to limit the creation of views to non-empty tables. Views can be created for a single data set or for several ones, in which case only those marked as visible will be shown. When dropping views, the exactly same parameter values must be passed as when creating views.

#### **2.10.1.4. Transport data set (formerly handle data set)**

This `transport_data_set()` procedure allows you to transport a data set in different ways.

- Direct execution: DML operations are generated depending on the selected mode (insert, update, delete, refresh, move, etc.) and directly executed in the source or target schema (depending in the operation) potentially through a database link;
- Prepare script only: a script with DML operations (again depending on the selected mode) is generated and made available in the `DS_OUTPUT` table or in `DBMS_OUTPUT` depending on the `p_output` parameter;
- Execute script: a script with DML operations is generated and executed on-the-fly in a local target schema (remote execution through a database link is unfortunately not supported due to Oracle limitations).

#### **2.10.1.5. Security policies**

Security policies can be used to export your data set via the standard export utility or data pump. They will guarantee that only the data of your data set are visible and exported.

Security policies can be created and dropped by calling the `create_policy()` and `drop_policy()` procedures. Parameters are like those used when creating and dropping views.

#### **2.10.1.6. Forcing column values**

It can be useful to overwrite the value of some columns during the extraction or transportation process. As an example, you may want to update the audit columns when inserting or updating data in the target schema.

As of v23.3, the specific implemented described below has been replaced with the SQL data masking technique. The following keywords can be used in the SQL expression to check which operation is currently being performed:

- INSERTING
- UPDATING
- DELETING
- SELECTING

These keywords are replaced at run-time with the following TRUE/FALSE expressions:

- 1=1 (TRUE) when the corresponding operation is performed.
- 1=0 (FALSE) when the corresponding operation is not performed.

Example of valid SQL expressions:

- CASE WHEN INSERTING THEN 'x' ELSE 'y' END;
- CASE WHEN INSERTING OR UPDATING THEN 'z' END;

## 2.11. Change data capture

Change data capture is a functionality that allows you to capture all DML operations (insert, update, and delete) performed on the tables included in your data set. Each captured operation and its old and new record values stored as XML are registered as data set records. DML operations are captured via database triggers created at table level. You can undo or redo captured operation, within the same schema or forward them to another schema, synchronously via direct DML or asynchronously via a dbms job.

A first typical use case is when you perform some tests which permanently destroy your data (committed insert, update or delete). You may want to capture all operations made on your data during your tests and rollback them back to their original state at the end of your test session (by undoing them one by one).

A second typical use case is to forward or replicate changes made to one schema (e.g., an operational database) into another one (the staging area of a data warehouse), either synchronously, either asynchronously (to remove dependencies and to avoid latency).

### 2.11.1. Properties

Properties used by the change data capture functionality are stored at the data set level. See the capture flag, capture mode and capture sequence properties of a data set.

### 2.11.2. Operations

Here follow the operations that you can perform in the context of the change data capture functionality. It is assumed that your data set has been created already and the tables for which you want to capture operations added to it.

#### 2.11.2.1. Create triggers

To create the triggers that will capture DML operations performed on a table, call the `create_triggers()` procedure and pass the data set id in parameter. When a generated trigger

does not compile, an exception is raised and the process is interrupted. If you want to continue with the creation of the triggers of other tables, you can specify to ignore any encountered error.

As one table can be part of multiple data sets, several triggers can be created on a table. Triggers are named according to the following pattern: `post_iud_ds{set_id}_tab{table_id}` where `{set_id}` and `{table_id}` are replaced with data set and table unique identifiers. Triggers are created with type "after insert or update or delete" and at record level (they trigger once for each row). Old and new individual values are converted internally to PL/SQL records.

In EXPort mode, DML operations are replicated synchronously. Insert, update and delete are executed directly in the target schema, possibly through a database link, depending on the properties defined for each table. This synchronous replication has the disadvantage that it introduces a dependency between the source schema and the target schema or database. If this latest is not available, the transaction in your source schema will fail.

When XML mode is activated, intermediary PL/SQL records containing new and old values are converted into XML and stored as data set records for subsequent manipulation (undo / redo / asynchronous replication). Generated records are ordered via the `seq` property.

XML and EXP mode can also be combined.

The capture process can be temporarily disabled by setting the `capture_flag` property at the data set level to N. Capture can also be limited to operations made by a specific user by setting the `capture_user` property at the data set level.

### **2.11.2.2. Drop triggers**

To drop the triggers previously created for a given data set, call the `drop_triggers()` procedure and pass the data set id in parameter. As when creating trigger, you can also indicate that you want to ignore any error to prevent the process from being interrupted should this happen.

### **2.11.2.3. Delete captured operations**

To delete records that have been captured so far, call the `delete_captured_operations()` procedure and pass the data set id in parameter. You also have the possibility to selectively delete operations of a specific user and/or keep the n first operations and delete subsequent ones (based on the sequence stored in `seq`).

### **2.11.2.4. Undo captured operations**

To undo (i.e., rollback) captured operations, call the `undo_captured_operations()` and pass the data set id in parameter. You also have the possibility to selectively rollback the operations made by a specific user and/or rollback the last n operations only. You can also specify whether you want to keep rolled back records (in which case they will be marked as logically deleted and their `undo_timestamp` will be set to the date and time of the undo).



The rollback is implemented as the following. An insert is rolled back via a delete (based on pk from new values). An update is rolled back via an update (old values are restored). A delete is rolled back via an insert (of old values). Updating primary keys is supported by considering them as a deletion of the old key followed by an insertion of the new one.

Capture of DML operations is temporarily disabled before the rollback by setting the `capture_flag` to N at the data set level. Previous value is restored once the rollback is over.

### **2.11.2.5. Redo captured operations**

For some reason, you may want to re-execute or redo some captured operations. This can be done by calling the `redo_captured_operations()` procedure and pass the data set id in parameter. As for the undo, you can limit to a specific user, to a number of operations and you can delete captured records once they have been processed.

Capture operations can be replayed in the same schema only if they have been previously rolled back (undo with captured records logically deleted). Their `undo_timestamp` which indicates a logical deletion will then be cleared. Captured operations can be replayed without restriction in another schema – when a target schema and/or a database link is/are specified at the table level. -

Capture operations are replayed in the same sequence as their recording i.e., in the ascending order of seq. Data capture is also temporarily disabled for the time of the redo (even when operations are replayed in another schema).

### **2.11.2.6. Rollback captured data set**

The `rollback_captured_data_set()` procedure is simply an undo of all captured operations (so without the possibility to filter or limit them). You still have the choice however to keep or delete rolled back records.

### **2.11.2.7. Rollforward captured data set**

The `rollforward_captured_data_set()` procedure is simply a redo of all captured operations (so without the possibility to filter or limit them). You still have the choice however to keep or delete roll forwarded records.

### **2.11.2.8. Create capture forwarding job**

Captured operations are replayed asynchronously via a dbms job that is created on-the-fly and executed once the current transaction is committed. In other words, the replication of data is delayed until all operations have been captured during the current transaction. Note that although dbms job is deemed to be replaced with dbms scheduler, this latest cannot be used as it performs an auto-commit (without any possibility of not doing it), on the contrary to dbms job. Dbms scheduler might be used in a future release if such an option is made available by Oracle.

Capture forwarding jobs are created automatically when the capture/replication mode is asynchronous (ASYN). They can also be created on demand by calling the `create_capture_forwarding_job()` procedure and passing the data set id in parameter. Such dbms jobs can be identified via a special comment inserted on their first line of code:

```
-- DS{set_id}_CAPTURE_FORWARDING where {set_id} is replaced with the data set id.
```

These jobs invoke the `rollforward_captured_data_set()` procedure exactly one minute after the last captured operation. They are executed in their own separate transactions which is automatically committed at the end. Any exception raised during this forwarding operation is ignored and not even logged (at least for the time being).

### 2.11.2.9. Drop capture forwarding job

Existing capture forwarding jobs can be dropped by calling the `drop_capture_forwarding_job()` procedure and passing the data set id in parameter. Such jobs are identified by the special comment put on their first line of code. No exception is raised when no job is found. Such jobs are automatically dropped when the data set definition is deleted.

### 2.11.2.10. Capture forwarding job exists

To check the existence of a capture forwarding job, call the `capture_forwarding_job_exists()` function and pass the data set id in parameter. This function returns a Boolean indicating the existence of such a job.

### 2.11.2.11. Generate captured data set script

DML operations that have been captured via triggers are recorded in the `DS_RECORDS` table and their related data (old and new values) are stored as XML. You may want to generate a script which redo or undo these operations for a subsequent manual execution. This can be achieved by calling the `gen_captured_data_set_script()` procedure or function. For both of them, pass the id of your data set and whether you consider a redo or an undo via the `p_undo_flag` (Y/N).

When the procedure is invoked, you can decide where the script must be output (`DBMS_OUTPUT` or in the `DS_OUTPUT` internal table (default)) via the `p_output` parameter. The pipelined function must be invoked via a SQL statement like:

```
SELECT * FROM TABLE(ds_utility_krn.gen_captured_data_set_script(<set_id>))
```

## 2.12. Sensitive Data Discovery

See Sensitive Data Discovery and On-the-fly Data Masking manual for more details.

### 2.12.1. Operations

### **2.12.1.1. Discover sensitive data**

Launch the sensitive data discovery process based on active patterns in `DS_PATTERNS` and, as results, store masks to be applied to sensitive columns in `DS_MASKS`. A report of the discovery run can be obtained by enabled information messages.

## **2.13. On-the-fly Data Masking**

See Sensitive Data Discovery and On-the-fly Data Masking manual for more details.

### **2.13.1. Operations**

#### **2.13.1.1. Propagate pk masking**

Propagate masking of primary key columns to foreign key columns. Foreign key columns will therefore INHERIT their value from the referenced primary key columns. Any mask that would have been defined on these foreign key columns will be overwritten (inheritance cannot be avoided). This operation is automatically performed by the tool before previewing or exporting data.

#### **2.13.1.2. Generate identifiers**

Generate the table mapping old identifiers with new identifiers generated from an in-memory sequence. This operation must be performed before previewing or exporting data.

#### **2.13.1.3. Shuffle records**

Shuffle records and column values based on shuffling groups and partitions. As a result, `shuffled_rowid_[1-3]` columns of records in `DS_RECORDS` are initialized. This operation must be performed after extracting rowids and before previewing or exporting data.

#### **2.13.1.4. Generate\_tokens**

Generate tokens in `DS_TOKENS` for columns having a TOKENIZE mask type. This operation must be performed before previewing or exporting data.

#### **2.13.1.5. Mask data set**

Execute all above operations (generate identifiers, shuffle records, generate tokens, and propagate pk masking) in a single call before previewing and transporting data.

## **3. Technical Aspects**

## 3.1. Message Types

The data set utility may display messages of the following types:

### 3.1.1. Error (E)

This type of message sent to the output indicates that an unexpected condition has been met and that the current processing must be halted. It normally triggers an ORA-20000 exception. The state of the database is unknown and it is advised to manually roll back the ongoing transaction.

### 3.1.2. Warning (W)

This type of message sent to the output warns you about an abnormal condition which however does not prevent the current processing from continuing.

### 3.1.3. Information (I)

This type of message sent to the output provides you with some useful information about the ongoing processing.

### 3.1.4. Debug (D)

This type of message sent to the output is used for debugging purpose.

### 3.1.5. SQL statement (S)

This type of message sent to the output is used to display the SQL statements that are dynamically executed during the current processing.

### 3.1.6. ROWCOUNT (R)

This type of message sent to the output is used to display the number of records (ROWCOUNT) inserted, update or deleted by the latest executed SQL statements.

## 3.2. Message masking

You can define very precisely which type of message you want to get displayed by calling the `set_message_mask()` procedure. You must pass in parameter a string that is the concatenation of the first letters of the message types you want to activate. The default mask is 'E' meaning that only error messages are displayed. It is of course not advised to disable error messages. Letters can appear in any order.

Examples:

- `set_message_mask('ED')` activates error and debugging messages
- `set_message_mask('EWI')` activates error, warning, and information messages
- `set_message_mask('SER')` activates logging of executed SQL statements and the number of impacted records i.e., `SQL%ROWCOUNT` (in addition to error messages)

### 3.3. Test mode

For debugging purpose, you may want to see the generated SQL statements without having them being executed. Test mode can be set via the `set_test_mode()` procedure with a Boolean as parameter.

Examples:

- `set_test_mode(TRUE)` will enable test mode
- `set_test_mode(FALSE)` will disable test mode

### 3.4. Masking mode

You may want to disable masking when previewing or exporting records. Masking mode can be changed via the `set_masking_mode()` procedure with a Boolean as parameter.

Examples:

- `set_masking_mode(TRUE)` will enable masking mode.
- `set_masking_mode(FALSE)` will disable masking mode.

### 3.5. Encryption of tokenized values

You may want to disable encryption of tokenized value. Encryption can be enable or disabled via the `set_encrypt_tokenized_values()` procedure with a Boolean as parameter.

Examples:

- `set_encrypt_tokenized_values(TRUE)` will enable encryption.
- `set_encrypt_tokenized_values(FALSE)` will disable encryption.

### 3.6. Output

Some outputs (e.g., generated scripts) are too long to be sent to the standard `dbms_output` (buffer overflow). The result of some procedures is instead stored in the internal `DS_OUTPUT` table. To delete the content of this table, the `delete_output()` procedure must be invoked.

## 3.7. Global variables

The following global variables can be defined (most of them through the APIs, others via direct assignment). Only those pertinent for end users are listed (others are for internal use only).

### 3.7.1. g\_test\_mode

The test mode allows you to display DDL statements instead of executing them.

Data Type: `BOOLEAN`

Default value is `FALSE` .

### 3.7.2. g\_mask\_data

Indicates whether data are masked during preview, export, or transportation.

Data Type: `BOOLEAN`

Default value is `TRUE` .

### 3.7.3. g\_encrypt\_tokenized\_values

Indicates whether tokenized values must be encrypted or not in `DS_TOKENS` .

Data Type: `BOOLEAN`

Default value is `TRUE` .

### 3.7.4. g\_show\_time

This variable allows you to print the date and time in all messages sent to the output thereby allowing you to see when each message was raised.

Data Type: `BOOLEAN`

Default value: `FALSE`

### 3.7.5. g\_time\_mask

This is the format used when printing date and time.

Data Type: `VARCHAR2(40)`

Default value: `'DD/MM/YYYY HH24:MI:SS'`

### 3.7.6. g\_timestamp\_mask

This is the format used when printing timestamps.

Data Type: VARCHAR2(40)

Default value: 'DD/MM/YYYY HH24:MI:SS.FF'

### 3.7.7. g\_msg\_mask

This allows you to filter which message types should be displayed.

Allowed values are a combination of: E)rror, W)arning, I)nfo, D)ebug, S)QL, R)owcount

Data Type: VARCHAR2(5)

Default value: 'WE'

### 3.7.8. g\_owner

This is the owner used when querying ALL dictionary views.

To be set to the schema owning the tables from which you want to extract data (when different than the schema in which the data set tool is installed).

Data Type: all\_objects.owner%TYPE

Default value: SYS\_CONTEXT('USERENV', 'CURRENT\_SCHEMA')

### 3.7.9. g\_alias\_like\_pattern

This is the pattern used to extract the alias from UK or PK constraint names.

Data Type: VARCHAR2(100)

Default value:

'([A-Z]\\*\_|^)([A-Z]\\*)((\_PK)|(\_UK\d\\*))\$'; -- {app\_alias}\_{table\_alias}\_PK|UKx

### 3.7.10. g\_alias\_replace\_pattern

This indicates which part of the pattern should be kept.

Data Type: VARCHAR2(100)

Default value: '\2'; -- {table\_alias}

### 3.7.11. g\_alias\_max\_length

This is the maximum length (in characters) of table aliases.

Data Type: `INTEGER`

Default value: `5`

### **3.7.12. g\_alias\_constraint\_type**

These are the constraint types to consider for extracting aliases.

Data Type: `VARCHAR2(20)`

Default value: `'PU'; -- consider PK and UKs`

Valid values are a combination of: P)primary (PK), U)nique (UK), R)elational (FK), C)heck (CK)

### **3.7.13. g\_capture\_job\_name**

This is the pattern used to generate job name for capture forwarding.

Data Type: `VARCHAR2(23)`

Default value: `'DS:1_CAPTURE_FORWARDING';`

### **3.7.14. g\_capture\_job\_delta**

This is the time after which a job starts executing after its creation.

Data Type: `NUMBER`

Default value: `10 / 86400; -- 10 seconds`

### **3.7.15. g\_xml\_batchsize**

This is the batch size when manipulating records in XML.

Null means don't set; 1 means no batch size.

Data Type: `NUMBER`

Default value: `NULL`

### **3.7.16. g\_xml\_commitbatch**

This indicates after how many XML operations a COMMIT should be performed.

NULL means don't set. 0 means no commit.



Data Type: NUMBER

Default value: NULL

### 3.7.17. g\_xml\_dateformat

This is the date and time format used in XML.

NULL means don't set.

Data Type: VARCHAR2(30)

Default value: 'dd/MM/yyyy HH:mm:ss'

See <http://docs.oracle.com/javase/7/docs/api/java/text/SimpleDateFormat.html>

## 4. Frequent errors

Here follow some errors that might be raised by the tool with their root cause and solution if any. Please note the many errors raised when passed parameters are invalid are self-explanatory and therefore not listed here.

### 4.1. Unsupported data type for column

Cause: the reported column has a data type which is not currently supported by the tool.

Solution: exclude the reported column from you data set definition by updating the columns list property of the table via `update_table_properties()` ; should you think this data type should be supported, please contact the support.

### 4.2. Maximum number of iterations exceeded

Cause: a potential infinite loop has been detected in the invoked procedure. The maximum limit defined for preventing such infinite loop has been reached.

Solution: review your data set definition or contact the support if you cannot solve it.

### 4.3. Loop in table dependencies detected

Cause: a loop has been detected in the dependencies of two or more tables. If table A depends on B and B depends on A, it's impossible to insert rows in A and B without disabling foreign key constraints, unless one of them is declared as `DEFERRED` . This is also true with indirect

dependencies i.e., those involving more than two tables like  $A \rightarrow B \rightarrow C \rightarrow A$  (where  $\rightarrow$  is a many-to-one relationship).

Solution: tables that are involved in this dependency loop have their `pass_count` property equals to zero; look at the constraints between those tables and decide on which one could be declared as `DEFERRED` via the `set_constraint_properties()` procedure and also at the database level (`ALTER CONSTRAINT`).

## 4.4. Infinite loop detected in xxx

Cause: an infinite loop has been detected in the reported procedure or function.

Solution: it's likely to be a bug; please contact the support.

## 4.5. Missing join clause for constraint #id

Cause: the join clause of a constraint is empty.

Solution: define a join clause for the constraint using the right table aliases; if the same problem arises again, please contact the support.

## 4.6. Row limit not compatible with full extract

Cause: you have defined a row limit for a fully extracted table.

Solution: reset the row limit property to `NULL` via the `update_table_properties()` procedure.

## 4.7. Table xxx has no primary key

Cause: one table in you data set has no primary key defined. A primary key is necessary to update and delete records of a data set.

Solution: add a primary key to your table or exclude the table from the data set definition.

# 5. GRAPHVIZ

This section describes how to visualise the configuration of a data set created for data subsetting (type `SUB`) or synthetic data generation (type `GEN`), in the form of a graph.

## 5.1. Description

The data set utility generates the description of a graph expressed in the DOT language that can subsequently be visualised using – e.g., an online version of – Graphviz (an open-source graph visualisation software).

## 5.2. Invocation

To generate the DOT code that will show the configuration of your data set, invoke the `graph_data_set()` pipelined function of the extension package ( `DS_UTILITY_EXT` ) with the following SQL statement:

```
SELECT * FROM TABLE(ds_utility_ext.graph_data_set(...));
```

Parameters of this procedure are the following:

Name	Usage (defined value is underlined/bolded)
p_set_id	Unique identifier of the data set for which the graph must be generated. NULL to show data set independent configuration (masks for data masking).
p_table_name	Include only tables whose name is matching (oracle wildcards or regular expression is allowed); NULL (default) means all tables.
p_full_schema	Include all tables of the schema (Y/ <b>N</b> )?
p_show_aliases	Show table aliases in addition to table name (Y/ <b>N</b> )?
p_show_legend	Show a legend explaining colours and arrow styles conventions (Y/ <b>N</b> )?
p_show_conf_columns	Show columns configured for data masking or data generation (Y/ <b>N</b> )?
p_show_cons_columns	Show columns involved in PK, UK, and FK (Y/ <b>N</b> )?
p_show_ind_columns	Show columns involved in indexes (Y/ <b>N</b> )?
p_show_all_columns	Show all tables columns (Y/ <b>N</b> )?
p_hide_dis_columns	Hide disabled and deleted masked columns (Y/ <b>N</b> )?
p_column_name	Include only columns whose name is matching (oracle wildcards or regular expression is allowed); NULL (default) means all columns.
p_show_column_keys	Show column keys i.e., Primary, Unique, Foreign, and Index (Y/ <b>N</b> )?
p_show_column_types	Show column data types (Y/ <b>N</b> )?

Name	Usage (defined value is underlined/bolded)
p_show_stats	Show statistics i.e., number of extracted/generated rows vs total number of rows ( <b>Y/N</b> )?
p_show_config	Show table, constraint, and columns/masks configuration ( <b>Y/N</b> )?
p_show_constraints	Show table PK, UK, and FK constraints (name and involved columns) ( <b>Y/N</b> )?
p_show_indexes	Show table indexes (name and involved columns) ( <b>Y/N</b> )?
p_show_triggers	Show table triggers ( <b>Y/N</b> )?
P_show_all_props	Show all properties in tooltips ( <b>Y/N</b> )? If No, properties shown on the diagram are not listed in the tooltips.
p_show_prop_regexp	Show (in tooltips) properties matching given regular expression; NULL by default which means show all properties.
p_hide_prop_regexp	Hide (in tooltips) properties matching given regular expression; NULL by default which means hide no property.
p_graph_att	Graph default attributes
p_node_att	Node default attributes
p_edge_att	Edge default attributes
p_main_att	Main subgraph attributes
p_legend_att	Legend subgraph attributes

## 5.3. Visualisation

To visualize the graph, copy/paste the result of the above query in an online Graphviz editor e.g., <https://dreampuf.github.io/GraphvizOnline> (but there are many others). The image of the graph can be saved in various formats.

The generated graph shows:

- Tables
  - Table name and, if requested, table alias ( @alias ).
  - The processing sequence number ( #<n> ).
  - Extract type (based on colours).
  - Constraint (PK, UK, FK) details if requested.
  - Index details if requested.

- Configuration: only what can fit on a diagram e.g., not the SQL statements.
- Statistics: number of rows extracted or generated vs total number of rows.
- Property values in a tooltip (may be filtered using inclusion/exclusion regexp).
- Columns (when requested and only those specified)
  - A key indicator showing if the column is involved in a **PK**, **UK**, **FK**, or **Index**.
  - Column name prefixed with a star ( ' \\* ' ) if mandatory.
  - Column data type (if requested).
  - Masking or generation type + some related properties.
  - Property values in tooltip (may be filtered using inclusion/exclusion regexp).
- Constraints
  - Constraint name (not always displayed).
  - Extract type (based on colours).
  - Configuration: only what can fit on a diagram e.g., not the SQL statements.
  - Statistics: number of rows extracted, generated, or referenced via each constraint.
  - Property values in a tooltip (may be filtered using inclusion/exclusion regexp).
- A legend if requested.

## 5.4. Graph examples


### 5.4.1. Data generation (GEN) – without columns

 Data Generation without columns


### 5.4.2. Data generation (GEN) – with all columns

 Data Generation with all columns

### 5.4.3. Data subsetting (SUB) – without columns

 Data Subsetting without columns

### 5.4.4. Data subsetting (SUB) – with all columns

 Data Subsetting with all columns

### 5.4.5. Sensitive Data Discovery – with sensitive and masked columns

 Sensitive Data Discovery Diagram

### 5.4.6. Data Structure Diagram

## 6. DEGPL

This section describes the Data Extraction and Generation Path Language (DEGPL) that can be used, as an alternative or a complement to using APIs, to describe the path that the engine must follow to extract or generate your data. Data masking can also be configured using DEGPL. The syntax of this language is inspired by and very close to the syntax of the Data Access Path Language (DAPL) that is used in the DOC utility.

This language can be used to configure data sets defined for data subsetting (type SUB), synthetic data generation (type GEN), and change data capture (CAP). To simplify, the description below will focus on subsetting and will therefore use the term “extraction”, but it can just be replaced with “generation” for GEN data sets and “capture” for CAP data sets.

### 6.1. Language description

The DEGPL language allows you to describe, in an easy and concise way, how to navigate in your data model to extract or generate data. As a result of parsing DEGPL code, records are created or updated in some internal tables of the tool ( `DS_DATA_SETS` , `DS_TABLES` , `DS_CONSTRAINTS` , `DS_TAB_COLUMNS` and `DS_MASKS` ).

#### 6.1.1. Properties

Data set, table, constraint, column, mask, and default properties can be specified using the following syntax:

```
<object> [<property1>=<id_or_number>, <property2>="<string>", ...]
```

List of properties is enclosed within square brackets and is placed just after the name of the object to which it applies. Properties are separated with a comma (preferred), a semi-colon or a space.

Property values must be enclosed in double quotes except for identifiers (made up of letters, digits, and/or underscore) and positive integers. For optional properties, a NULL value can be encoded as an empty string i.e., '' .

Property names are case insensitive and can be specified in full (e.g., `batch_size=1000`) or in part (e.g., `b=1000`) for the sake of conciseness. When an abbreviation is used, it must uniquely identify a property of the object for which it is defined i.e., there can only be one property whose name starts with the given abbreviation.

## 6.1.2. Data set definition

You must start your DEGPL code by identifying the SUB, GEN, or CAP data set that you want to configure, together with its properties. As an exception, this is not required for configuring data masking as its configuration is data set independent.

The syntax is the following:

```
set <name> /option [<properties>]
```

Where:

- <name> is the name of the data set; it will be converted to uppercase unless you enclose it within double quotes.
- Valid options are “/d” or “/r” to delete before creating or, create or replace the data set; this latest is recommended as it has the advantage that the `set_id` remains the same.
- Data set properties follow the syntax described in section [Properties 6.1.1](#). Their name derives from column names of the `DS_DATA_SETS` table.

Examples:

- `set demo_data_gen/r [set_type=GEN]; -- create or replace.`
- `set demo_data_sub/d [set_type=SUB]; -- delete then create.`
- `set demo_data_cap [capture_mode=SYNC]; -- use existing by name.`
- `set [set_name=DEMO_DATA_GEN];` alternative syntax.
- `set [set_id=10]; -- identify an existing data set by id.`

In all cases, properties are set or updated as defined.

## 6.1.3. Extraction path

The extraction process consists of extracting records from one or several base/driving tables and to follow one-to-many relationships to extract dependent records from neighbouring tables. As you can define several base tables, DEGPL allows you to specify several extraction paths (separated with a semi-colon).

An extraction path between tables is described using directed arrows; the direction is important as a foreign key constraint can be followed in both directions while extracting data.

The general syntax of an extraction path between table A and B is A->B (read: A arrow B). The type of arrow indicates the extraction direction (e.g., from A to B) and the extraction type for the underlying constraint (e.g., B, P or N).

## 6.1.4. Extraction direction

Syntax	Direction
A->B	From A to B (single direction)
A<-B <=> B->A	From B to A (single direction)
A<->B <=> A->B & B->A	From A to B and B to A (dual direction)

## 6.1.5. Extraction types

Syntax	Extract type
A=>B	B)ase extraction type (from base records)
A->B	P)artial extraction type (for referential integrity)
A≠>B <=> A+>B (1)	N)one (not used for extraction)
A#>B	eXcluded from data set (i.e., to be deleted)
A~>B (2)	Unspecified (keep existing extract type)

Notes:

1. A strikethrough arrow means that the foreign key cannot be crossed during the extraction; as ≠ (not equal to, U+2260) is not part of the ANSI character set and cannot be easily entered via a keyboard, + (plus) is provided as an alternative.
2. ~ (tilde) indicates that the extraction type is not specified meaning that, if it was already specified before, it will remain as it is i.e., it will not be overwritten.

For recursive foreign keys (also called ear's pig), the syntax A->A is ambiguous as it does not specify the direction (1-N or N-1) that must be used to cross the constraint.

To solve this ambiguity, the cardinality of the constraint must be specified using one of the following arrow types:

## 6.1.6. Cardinality

Syntax	Cardinality
A>-B or B-<A	Many-to-one (N-1) from A to B
A-<B or B>-A	One-to-many (1-N) from A to B
A->B or B<-A	Unspecified cardinality (i.e., 1-N or N-1), from A to B

Notes:



- $A \rightarrow B$  ' is the textual description of
- $A \leftarrow B$  is the textual description of
- When cardinality is specified for non-recursive foreign keys, it must match the cardinality of the constraint (so  $A \rightarrow B$  is invalid if the foreign key is from B to A).

## 6.1.7. All arrow types

Here follows the 30 possible arrows resulting from the combination of extraction types, directions, and cardinalities:

A to B	B to A	N-1	1-N	Dual dir	Extraction type
$A \Rightarrow B$	$A \Leftarrow B$	$A \Rightarrow B$	$A \Leftarrow B$	$A \Leftrightarrow B$	B)ase
$A \rightarrow B$	$A \leftarrow B$	$A \rightarrow B$	$A \leftarrow B$	$A \leftrightarrow B$	P)artial
$A \nrightarrow B$	$A \nleftarrow B$	$A \nrightarrow B$	$A \nleftarrow B$	$A \nleftrightarrow B$	N)one
$A \multimap B$	$A \multimap B$	$A \multimap B$	$A \multimap B$	$A \multimap B$	N)one
$A \# \rightarrow B$	$A \# \leftarrow B$	$A \# \rightarrow B$	$A \# \leftarrow B$	$A \# \leftrightarrow B$	eXcluded (=to be deleted)
$A \sim \rightarrow B$	$A \sim \leftarrow B$	$A \sim \rightarrow B$	$A \sim \leftarrow B$	$A \sim \leftrightarrow B$	Unspecified (kept as it is)

## 6.1.8. Constraint name

When several foreign keys exist from table A to table B, the  $A \rightarrow B$  syntax is also ambiguous.

The ambiguity can be removed by specifying the name of the foreign key:  $A \rightarrow B[fk='fk1']$ . The foreign key name is case insensitive, but it is converted to uppercase when stored. The `fk` property is a shortcut for the `constraint_name` property (see next section).

For each specified arrow, a record is created in `DS_CONSTRAINTS` if it does not exist yet or updated otherwise. For dual direction arrows, one record is created for each direction.

## 6.1.9. Constraint properties

Constraint properties as described in section [6.1.1](#) must be specified after the arrow i.e.,  $A \rightarrow [<properties>] B$ . Property names correspond to the column names of the `DS_CONSTRAINTS` table, except `constraint_name` that is replaced with `fk` for the sake of conciseness. Extract type `x` means that the constraint must be eXcluded (i.e., deleted) from the data set.

## 6.1.10. Arrow shortcuts

An extraction path is a succession of FK constraint crossings; from a notational point of view, they are separated with a semi-colon ( ; ). So, an extraction path from A to B, then from B to C, is

denoted  $A \rightarrow B; B \rightarrow C$  .

When a table is crossed by an extraction path i.e., when the tables to the left and to the right of the semi-colon are the same (as table B in the above example), the following equivalent shortcut notation can be used:  $A \rightarrow B \rightarrow C \iff A \rightarrow B; B \rightarrow C$  .

When 2 tables B and C are reached from the same table A, the following shortcut notation can be used:  $A \rightarrow B, C \iff A \rightarrow B; A \rightarrow C$  . Any property specified for the arrow (included extraction type) will be applied to both constraints.

When a table C is reached from 2 tables A and B, the following shortcut notation can be used:  
 $A, B \rightarrow C \iff A \rightarrow C; B \rightarrow C$  .

More generally:

$$A, B \rightarrow C, D \iff A \rightarrow C; A \rightarrow D; B \rightarrow C; B \rightarrow D \iff A, B \rightarrow C; A, B \rightarrow D \iff A \rightarrow CD, B \rightarrow C, D$$

Example:

The following extraction paths are all equivalent:

- $A \rightarrow B; B \rightarrow C; A \rightarrow D$  (the most verbose)
- $A \rightarrow B \rightarrow C; A \rightarrow D$  (the most readable; preferred)
- $D \leftarrow A \rightarrow B \rightarrow C$  (the most concise)
- $A \rightarrow B, D; B \rightarrow C$  (the less readable)

## 6.1.11. Table identifiers

Tables can be identified by their name or by their alias, which are both case insensitive. Table names are converted to uppercase when stored while table aliases are converted to lower case.

When only a table name is specified, table alias is derived from PK or UK constraint(s), depending on your naming conventions, or generated otherwise (by default,  $t_{\langle table\_id \rangle}$  ).

When only a table alias is specified, it must reference a table that has already been included in the data set i.e., which exists in `DS_TABLES` already.

When both a table name and a table alias are specified, the given table alias is used instead of being derived or generated, and it overwrites any alias already defined before.

In all cases, a record is created in `DS_TABLES` if it does not exist yet or is updated otherwise.

## 6.1.12. Table extract type

Table extract type can be specified after the table name with a slash ( / ) followed by letter B, P, F, N, R, X or D e.g., `Table1/B->Table2` . When a record already exists in `DS_TABLES` , its extract type is updated only if it is specified explicitly. When no record exists in `DS_TABLES` , a new record is inserted

with the specified extract type and, when not specified, with P as the default value. Extract type X or D means that the table must be eXcluded (i.e., Deleted) from the data set.

## 6.1.13. Table properties

Table properties as described in section [6.1.1](#) must be specified after the table alias or after the table name when no alias is given e.g., `table_name table_alias [property="value"]` . Property names correspond to the column names of the `DS_TABLES` table.

## 6.1.14. Column properties

For GEN data sets, column properties (as described in section [6.1.1](#)) can be specified with the following syntax:

```
<table>.<column1>[<properties>]...
```

Table definition must be followed by a dot ( . ), the column name and its properties. A column can be deleted from a data set by specifying `/d` (Delete) or `/x` (eXclude) after its name.

Note: column properties are linked to a data set and are stored in `DS_TAB_COLUMNS` .

## 6.1.15. Mask properties

To mask your data during subsetting, mask properties (as described in section [6.1.13](#)) can be specified with the following syntax (identical to column properties):

```
<table>.<column1>[<properties>]...
```

Table definition must be followed by a dot ( . ), column name and its masking properties enclosed between squared brackets ( [ and ] ). A mask can be deleted by specifying `/d` (Delete) or `/x` (eXclude) after its column name.

Note: mask properties are independent from a data set and are stored in `DS_MASKS` .

## 6.1.16. Wildcards

Wildcards `?` and `*` matching respectively 1 and N characters can be used in table, constraint, and column names. This allows you to include several tables and/or constraints in a once.

Examples:

- `demo*` ; includes all demo tables (tables whose name is starting with “demo”).
- `demo*<->demo*` ; includes all connected demo tables and their relationships (in both directions); isolated demo tables (i.e., demo tables which have no relationship with each other) are not included.

- `demo_persons.*[msk_type=SQL]` ; set the mask type to SQL for all columns of the given table.

## 6.1.17. Recursion

Constraints can be followed recursively by specifying a maximum number of recursion level after the arrow (must be >0 or 0 for no recursion limit). If a cardinality is specified (e.g., N-1 or 1-N via >- or -< ), only those having the specified cardinality will be included. The specified extraction type (indicated by the arrow type) will also be applied to all added constraints.

Example:

- `demo_persons/B=<2*` ; includes `DEMO_PERSONS` (master table) and, recursively (up to 2 levels), its detail tables following 1-N relationships (due to `=<` ); extraction type is set to B for master table (due to `/B` ) and all added constraints (due to `=` ).

## 6.1.18. Scope

One of the following scopes can be specified before a table, a constraint (arrow) and/or a column:

- Only tables/constraints/columns that are already included in the data set: “ $\exists$ ” (exist symbol), “ $\subseteq$ ” (include symbol), or “!” (exclamation mark).
- Only tables/constraints/columns that are not included in the data set yet: “ $\nexists$ ” (not exist symbol), “ $\not\subseteq$ ” (not include symbol), or “^” (caret).

Scope makes sense and is allowed only when the table/constraint/column name contains a wildcard.

When no scope is specified, it means that all tables/constraints/columns are considered, irrespective of whether they are already in the data set or not.

Examples:

- `!*^>-*` : will add missing referential constraints from any table of the data set to any target table (in the data set or not).
- `!*` : means any source table already in the data set.
- `^>-` : means missing N-1 relationships (i.e., not in the data set yet).
- `*` : means any target table (will be added to the data set if not included yet).

## 6.1.19. Default values

You can specify default values for table, constraint, column, and mask properties with the following syntax:

- `table[<properties>];`
- `constraint[<properties>];`
- `column[<properties>];`

- `mask[<properties>];`

Examples:

```
table[source_schema="XXX"];
constraint[batch_size=1000];
column[gen_type=SQL];
mask[locked=Y, msk_type=SQL];
```

A scope can also be specified, using the same symbols as just described above:

- `!` (exclamation mark or its equivalent): the property is forced to the given value in all cases i.e., whether the property is explicitly listed or not, whatever is its previous value.
- `^` (carret or its equivalent): the property is set to the given value if it is not explicitly listed, whatever is its previous value.

When no scope is specified, the property is set to the given value only if it is not explicitly listed and if its previous value was NULL.

Examples

```
demo_dual;
table[locked=Y]; -- force value if NULL
demo_dual; -- locked set to Y as it was previously NULL
```

```
demo_dual[locked=N];
^table[locked=Y]; -- force value if not listed
demo_dual[locked=N]; -- value set to N as listed
demo_dual; -- value set to Y as not listed
```

```
!table[locked=Y]; -- force value in all cases
demo_dual; -- value set to Y as not listed
demo_dual[locked=N]; -- value set to Y, even if listed
```

Default values can be reset/deleted by adding a `/d` (Delete) after the object name e.g., `table/d;`

## 6.2. Invocation

To include an extraction path or a generation path described in the DEGPL language to your data set, invoke the `execute_degpl()` procedure available in the extension package ( `DS_UTILITY_EXT` ). Any pre-existing configuration of the data set (e.g., made via API's) can be complemented or completely replaced (via `/D` or `/R` data set options).

## 6.3. Recommendations

Here follow some recommendations on how to structure and test your DEGPL code:

- Preferably use the /R option to create or replace your data set.
- Include all tables that you need (using wildcards); this allows you to subsequently reference them in a shorter way, using their alias rather than their name; exclude those that you don't need.
- Adapt table aliases where needed and specify table properties (e.g., identify base tables and tables that are not extracted/generated like reference tables).
- Include 1-N constraints from base table(s) (using wildcards and recursion).
- Include (missing) N-1 constraints to enforce referential integrity.
- Set or adapt constraint properties according to your needs (e.g., extract type).
- For GEN data set, define how to generate column values via column properties.
- For SUB data set, define how to mask columns via mask properties.
- Parse and execute your DEGPL code via `execute_degpl()`.
- Check visually the configuration of your data set via `graph_data_set()`.
- Extract or generate your data set.
- Iterate as necessary.

## 6.4. Example

The following extraction path are described in the DEGPL language:

```
set demo_data_sub/r[set_type=SUB];
demo_persons per/B[where_clause="per_id<=10", columns_list="*"]=>[row_limit=2]demo_order
per=>demo_per_clockings pcl, demo_per_assignments pas, demo_per_credit_cards pcc;
ord=>demo_order_items oit,demo_per_transactions ptr;
pas->demo_org_entities oen>-oen[];
demo_countries/N; demo_credit_card_types/n; demo_org_entity_types/N; demo_stores sto/F,
```

or, more shortly (using table aliases, wildcards, recursion, and abbreviated property names):

```
set demo_data_sub/r[set_type=SUB];
demo*->demo*; sto/f; cnt/n; cct/n; oet/n;
per/b[where="per_id<=10"]=<2*;
per~>[percent=50]ord;!^>-*;
```

...will result in the following extraction diagram:

The diagram shows how records are extracted (via colours) and give some statistics on the number of records extracted from each tables and through each constraint.

## 6.5. EBNF

Here is the description of the grammar of the DEGPL language expressed in EBNF:

```
(* EBNF description of DEGPL - Data Extraction and Generation Path Language *)

(* Drawn with https://jacquev6.github.io/DrawGrammar/ *)

extraction_path = tables_list, [ paths_list ] , [ ';' , extraction_path ];

tables_list = table_descr, [ ',' , tables_list];

table_descr = table_identifier, [ '/' , table_extract_type ], [ table_properties ],

table_identifier = [scope], (table_name | table_alias | ( table_name , table_alias));

scope = '∃, ⊆ or ! /*already existing or included in data set*/' | '∄, ⊄ or ^ /*not ex:

table_extract_type = ('B /*Base/' | 'P /*Partial*/' | 'F /*Full*/' | 'N /*None*/' | 'X

properties = ('[' , properties_list , ']' ) , [properties];

properties_list = property_descr, [ [',' | ';' | 'space'], properties_list ];

property_descr = property_name, '=', (bare_number | identifier | '"' , string, '"') ;

columns_list = column_descr, [ columns_list ] ;

column_descr = '.', [scope] , column_name , [ '/' , ['D /*Delete*/' | 'X /*eXclude*/' ]

paths_list = path, [ paths_list ];

path = arrow, [ properties ], tables_list;

arrow = [scope], (single_dir_arrow | dual_dir_arrow);

single_dir_arrow = (('<' | '>' ) , cons_extract_type) | (cons_extract_type , ('<' | '>

dual_dir_arrow = '<' , cons_extract_type , '>';

cons_extract_type = '= /*Base*/' | '- /*Partial*/' | '≠ or + /*None*/' | '# /*Delete*/'
```

## 6.6. Syntax diagram

Here is the syntax diagram of the EDGPL language (generated from the EBNF description):

# 7. Format-Preserving Encryption (FPE)

## 7.1. Introduction

**Format-Preserving Encryption (FPE)** is a type of encryption that ensures the ciphertext (the encrypted output) retains the same format as the plaintext (the original input). FPE is crucial in the context of databases because it allows encrypted values to be stored in the same structure (table/column) as the original values, without requiring modifications.

The data set utility provides a library of functions (the `DS_CRYPT0_KRN` package) to encrypt and decrypt basic data types such as numbers, integers, strings, and dates (with or without a time component).

Two FPE algorithms are supported: a custom algorithm named **RSR** (Random Substitutions and Rotations, implemented in the `DS_CRYPT0_RSR_KRN` package), and the standard **FF3** algorithm (implemented in the `DS_CRYPT0_FF3_KRN` package), which is approved by NIST (the U.S. National Institute of Standards and Technology). Each algorithm has its own advantages and disadvantages.

Additional functions for encrypting and decrypting complex data types (e.g., IBANs, credit card numbers, etc.) are also provided in the library of masking functions (the `DS_MASKER_KRN` package).

## 7.2. RSR Algorithm

### 7.2.1. Overview

The custom RSR algorithm is based on a sequence of random substitutions and rotations. The encryption key is used solely to define the seed for multiple pseudo-random number generators (PRNGs). The encryption process consists of a random number of rounds. During each round, a window of two symbols shifts from left to right by one symbol, substituting each pair using a randomly generated substitution table. At the end of each round, the entire string is rotated by one symbol to the left or right, also determined randomly. To encrypt small values consisting of a single symbol, a separate randomly generated substitution vector is used.

### 7.2.2. Substitution tables



The algorithm employs a total of eight substitution tables. Different tables are used depending on whether the input is a number or a string, whether the substitution involves a pair of symbols or single symbols, and whether the operation is encryption or decryption.

Substitution tables do not store the symbols themselves but rather their positions or indices in the alphabet. These tables contain values from 0 to  $n-1$  (where  $n$  is the radix for single-symbol tables and  $\text{radix}^2$  for pair-of-symbol tables).

Substitution tables are initially populated with all possible values, and their elements are then permuted randomly  $n-1$  times. Reverse tables, used for decryption, are generated simultaneously with the forward tables used for encryption.

These tables are also designed so that recursive encryption cycles through all possible values in the domain before repeating. Taking into account this construction, the number of possible substitution tables is equal to  $(n-1)!$ .

- Single-digit tables (10 entries):  $(10-1)! = 362.880$  possible combinations
- Two-digits tables (100 entries):  $(100-1)! \approx 9,33 \times 10^{155}$  (extremely large number)
- Single printable ASCII char tables (156 entries):  $(156-1)! \approx 5.63 \times 10^{260}$  (astronomic number)
- Two printable ASCII char tables (24.336 entries):  $(156^2-1)! \approx 3.16 \times 10^{95.591}$  (astronomic number)

These calculations show the vast number of possible permutations for each type of table, reflecting the complexity and variability of the substitution tables used in the RSR encryption algorithm.

It is important to note that these tables are constructed each time a new encryption key is provided. Since this process is time-consuming, it significantly complicates the implementation of a brute-force attack.

### 7.2.3. Pseudo-Random Number Generators

Pseudo-Random Number Generators (PRNGs) that are used in cryptography should meet several important criteria to ensure security and reliability amongst which:

- State Size: For cryptographic security, use a PRNG with at least a 128-bit state size. For enhanced security, 256-bit state sizes are recommended.
- Period Length: Ensure the PRNG has a sufficiently long period to avoid repeating patterns and ensure unpredictability.

Unfortunately, PL/SQL does not support PRNGs with sufficient precision for 128-bit encryption, as 128 bits require approximately 426 decimal digits to represent all possible values, while Oracle only supports up to 38 decimal digits. As a workaround, four different LCG-based PRNGs are employed, each serving a specific purpose and initialised with different hashing algorithms applied to the encryption key to compute their seed values:

1. **HASH-SHA256 (left half):** Used to generate single-symbol substitution tables.
2. **HASH-SHA256 (right half):** Used to generate two-symbol substitution tables.
3. **HASH-MD4:** Used to generate a random number of rounds.
4. **HASH-MD5:** Used to generate a random rotation direction (left or right).

## 7.2.4. Encryption of numbers

The RSR algorithm supports encrypting numbers of type NUMBER, including both integers and decimal numbers. Precision and scale for the column where the value is stored can be provided as parameters, with the understanding that the value to be encrypted must be compatible with these parameters. If precision and/or scale are not specified, they are computed from the original value. Sign of the original value is preserved.

To encrypt a number, it is first converted to a positive integer by taking its absolute value and multiplying it by  $10^{\text{scale}}$ . This integer is then left-padded with zeros to achieve the target precision. The encryption process involves performing rounds of substitutions and rotations. If the resulting ciphertext does not meet the required format (e.g., having leading zeros when not expected), the process may need to be repeated. The encrypted integer is then converted back to a number.

## 7.2.5. Encryption of strings

The RSR algorithm can encrypt strings composed of characters from a specified alphabet or charset. You can provide a maximum length parameter, corresponding to the length of the column where the values are stored. When not specified, the ciphertext will have the same length as the plaintext. An alphabet (a list of allowed characters) can also be specified, or a format string can be used as an alternative way to define the alphabet (e.g., `aA0` includes all lowercase letters, uppercase letters, and digits). All characters of the plaintext must belong to the specified alphabet. If not specified, the default alphabet includes all printable ASCII characters (156 in total).

To encrypt a string, it is first right-padded with spaces to achieve the desired length. The encryption process involves rounds of substitutions and rotations. If the resulting ciphertext does not meet the requirements (e.g., it contains trailing spaces or characters not in the specified alphabet), the encryption may need to be repeated.

## 7.2.6. Encryption of dates

Dates can be encrypted by first converting them into Julian numbers, encrypting these numbers, and then converting them back to dates. For dates that include a time component, the time is first converted into a number of seconds (up to 5 digits), which is then encrypted and converted back to the time. A range of valid dates can be specified as parameters. If provided, the original date must fall within this range. The encryption process for dates is repeated until the resulting encrypted date falls within the specified range. Similarly, the encryption process for times is repeated until the

resulting time is valid, within the range of 0 to 86,399 seconds. The cyclic property of the substitution tables ensures that the process will complete in a finite number of iterations.

## 7.2.7. Encryption of integers

Integer within a given range can be encrypted using the encryption of numbers. The integer may need to be shifted first to reduce the number of digits (precision) to be encrypted. The encryption process is repeated until the ciphered number falls into the specified range. The cyclic property of the substitution tables ensures that the process will complete in a finite number of iterations.

## 7.2.8. Decryption

Decryption follows the same procedure as encryption but uses reverse substitution tables and applies all operations in reverse order (e.g., rotations are performed before substitutions).

# 7.3. FF3 Algorithm

## 7.3.1. Overview

F3 is a specific type of format-preserving encryption algorithm that is part of a Feistel-based family of encryption techniques, widely used for securely encrypting structured data while maintaining its format. A Feistel network is a cryptographic structure that divides the data into two halves and processes them iteratively using round functions. It is a common approach in many block cipher designs, including well-known algorithms like DES (Data Encryption Standard).

FF3 is standardized by the National Institute of Standards and Technology (NIST) as part of the SP 800-38G recommendation, which describes methods for FPE. FF3 was designed as a more efficient version of the FF1 algorithm with respect to computation and memory. FF3 is used in industries like banking and healthcare, where it's critical to encrypt sensitive data (like account numbers or personal identifiers) while maintaining its original structure to ensure compatibility with legacy systems.

In addition to the private key, FF3 also uses a tweak. A tweak in cryptography is a small piece of additional data used alongside the encryption key to introduce variability in the encryption process without altering the key itself. It ensures that even if the same plaintext is encrypted with the same key, the output (ciphertext) will differ if a different tweak is used.

In the context of format-preserving encryption (FPE) algorithms like FF3 or FF3-1, the tweak helps to add an extra layer of security, allowing the same plaintext to be encrypted differently across different contexts, which is particularly useful in applications like encrypting database records.

The tweak itself does not need to be kept secret like an encryption key, but it must be consistent between encryption and decryption processes. This means that while the tweak can be public, it

should be carefully managed to ensure that the same tweak is used when decrypting data that was encrypted with it.

FF3-1 was developed to address the vulnerabilities found in FF3 (deprecated since then), particularly concerning tweak size and domain size. It uses a slightly different tweak length (56-bit compared to 64-bit in FF3) and has more stringent guidelines for domain sizes (domain must contain at least 1 million values), making it a more secure option for format-preserving encryption.

The data set utility leverages a Java implementation of FF3-1, available on GitHub (<https://github.com/mysto/java-fpe>). An additional `DSFF3Cipher` class is implemented to instantiate the `FF3Cipher` class following the singleton pattern. Encryption and decryption are performed via two PL/SQL wrapper functions: `ff3encrypt()` and `ff3decrypt()`. The key, tweak, and alphabet are passed as parameters, along with the plaintext to encrypt or the ciphertext to decrypt. Changing any of these three parameters will result in a new instantiation of the `FF3Cipher` class.

## 7.3.2. Encryption of numbers

Unlike the RSR algorithm, which has a dedicated implementation for numbers, FF3 only accepts strings as plaintext. To encrypt numbers with FF3, they must first be converted to strings, and the `0123456789` alphabet must be specified.

### Minimum Length Requirement:

- The algorithm requires a minimum of 2 digits.
- FF3-1 introduces a stricter recommendation regarding the domain size, suggesting that the domain should contain at least 1 million unique values. For numbers, this means using at least 6 digits.

### Ways to Address the Limitation:

#### 1. Ignoring the Recommendation:

- The minimum digit requirement can be kept at 2 by ignoring the FF3-1 recommendation.
- This is the default behavior, controlled by the `ff3_min_len` global parameter.
- To revert to the NIST recommendation, you can call the `set_min_len()` method of the `DS_CRYPTO_KRN` package with a `NULL` value, allowing the system to compute the minimum length based on the alphabet.

#### 2. Padding Numbers:

- Numbers can be left-padded with zeros to reach the 6-digit minimum.
- The encryption process then iterates until it finds a ciphertext that matches the original number of digits (after trimming leading zeros).
- This method can be time-consuming, especially for very small numbers (e.g., 1 or 2 digits), potentially taking several seconds, so it is not recommended unless necessary.

### Lifting the Two-Digit Limit:

- Padding is also used to overcome the hard limit of 2 digits imposed by the algorithm.
- In this case, the number of iterations required is typically limited and manageable.

#### **Enhanced FF3 Encryption:**

- An additional layer has been developed on top of the FF3 encryption to address these limitations, allowing for the encryption of numbers with full support for negative and decimal values.

### **7.3.3. Encryption of strings**

The encryption of strings suffers from the same limitations as numbers regarding its minimum length. When the alphabet is made up of 156 ASCII printable characters, the minimum string length is 3 to get a domain of at least 1 million values. In any case, the minimum string length imposed by the FF3 algorithm is 2.

The ways to address these limitations are the same as for numbers except that right padding with spaces are applied instead of left padding with zeros. An additional layer has also been developed on top of the FF3 encryption to address these limitations.

### **7.3.4. Encryption of dates**

The same algorithm as RSR is used for encrypting dates within a range with FF3. Due to the nature of the permutations and the structure of the Feistel network, FF3 is designed to ensure that all possible outputs (ciphertexts) are covered when the algorithm is applied repeatedly. This is crucial because it guarantees that, given sufficient iterations, the encryption will eventually yield a ciphertext that meets any specific criteria (such as falling within a certain numeric range or avoiding leading zeros).

### **7.3.5. Encryption of integers**

The same algorithm as RSR is used for encrypting integers within a range with FF3.

## **7.4. Comparison**

Here's a comparison between the RSR and FF3 encryption algorithms based on limitations, performance, and approval by certification authorities.

### **7.4.1. Limitations**

#### **7.4.1.1. RSR**

- **Minimum Number of Digits/String Length:**

- **Numbers:** RSR can encrypt numbers of any length, even single-digit numbers, without strict limitations on the domain size.
- **Strings:** Similar flexibility applies to strings, with the algorithm handling strings of various lengths without requiring a minimum number of characters.
- **Domain Restrictions:**
  - RSR does not have specific domain size restrictions, allowing it to be more flexible with smaller datasets. However, it may need to repeat the encryption process to meet certain formatting requirements (e.g., removing leading zeros).
- **Customisability:**
  - The algorithm is home-grown, meaning it can be tailored to specific needs, though it might lack some of the robustness and standardised guarantees of more widely accepted algorithms.

### 7.4.1.2. FF3

- **Minimum Number of Digits/String Length:**
  - **Numbers:** FF3 is more restrictive. While the theoretical minimum is 2 digits, the FF3-1 recommendation suggests that the domain size should contain at least 1 million values. This implies that numbers should ideally have at least 6 digits.
  - **Strings:** Similarly, for strings, the minimum length should be sufficient to cover a domain of at least 1 million values based on the alphabet size.
- **Domain Restrictions:**
  - FF3's restrictive domain size is tied to its security guarantees. For instance, using an alphabet of 156 characters, the string should have a minimum length of 3 characters to cover a domain of  $156^3$  (3,796,416 values).

## 7.4.2. Performance

### 7.4.2.1. RSR

- **Speed:**
  - RSR is generally less performant compared to FF3, particularly because it is implemented in PL/SQL, which tends to be slower than Java. The PL/SQL implementation involves more overhead in terms of execution time and resource utilisation.
- **Efficiency:**
  - The algorithm involves random substitutions and rotations, which can be computationally expensive and time-consuming, especially when dealing with larger datasets. The process of repeatedly encrypting until the output meets format requirements (e.g., ensuring a specific number of digits) adds additional overhead.
- **Implementation:**
  - The performance of RSR is further impacted by the fact that it is a home-grown solution. While it can be optimised for specific use cases, its PL/SQL implementation introduces

inefficiencies compared to more streamlined, compiled languages like Java.

#### 7.4.2.2. FF3

- **Speed:**
  - FF3 is designed to be efficient and fast, even though it involves complex operations like multiple rounds of permutation and substitution. Its implementation in Java is generally more performant compared to PL/SQL-based solutions, thanks to Java's optimised runtime and advanced libraries.
- **Efficiency:**
  - The algorithm benefits from its design, which is geared towards speed and efficiency. FF3's approach is optimised for high performance, making it suitable for environments where quick encryption and decryption are critical, especially given its ability to handle larger domains efficiently.
- **Implementation:**
  - Java's efficient handling of cryptographic operations and the use of well-optimised libraries contribute to FF3's superior performance. This contrasts with RSR's PL/SQL implementation, which may be less efficient due to inherent limitations of PL/SQL in terms of execution speed and resource management.

### 7.4.3. Approval by Certification Authorities

#### 7.4.3.1. RSR

- **Certification:**
  - RSR is a home-grown algorithm and, as such, is not approved or certified by any formal cryptographic standardisation bodies like NIST. It may not meet the rigorous security standards required for use in regulated environments.
- **Use Cases:**
  - While RSR can be effective in specific, controlled environments, its lack of certification limits its use in scenarios where compliance with industry standards is required.

#### 7.4.3.2. FF3

- **Certification:**
  - FF3 is a NIST-approved encryption algorithm, making it suitable for use in environments that require adherence to formal cryptographic standards. This approval makes FF3 a strong choice for applications requiring compliance with industry standards, such as finance or government.
- **Use Cases:**
  - FF3 is preferred in scenarios where regulatory compliance, security assurances, and standardization are critical. Its approval by NIST provides confidence in its robustness and security.

## 7.4.4. Summary

- **Limitations:** RSR is more flexible and can handle small datasets more easily, while FF3 has stricter limitations, particularly regarding domain size.
- **Performance:** RSR, implemented in PL/SQL, is generally less performant due to the limitations of PL/SQL compared to Java. FF3, being designed for efficiency and implemented in Java, offers superior performance, especially for larger datasets, thanks to its optimised cryptographic processes.
- **Approval:** FF3 is NIST-approved, making it a better choice for regulated environments, whereas RSR is not certified and is more suited to custom or non-standard applications.

## 7.5. APIs

### 7.5.1. Parameters common to all methods

The APIs provided by the `DS_CRYPT0_KRN` package are shared between both algorithms. Depending on the value of the `p_algo` parameter (RSR - the default - or FF3), encryption and decryption of basic data types are handled by the algorithm-specific package: `DS_CRYPT0_RSR_KRN` for RSR and `DS_CRYPT0_FF3_KRN` for FF3. Algorithms for encrypting dates and integers within a specified range are common to both algorithms and therefore implemented in the `DS_CRYPT0_KRN` package.

The format of the private key ( `p_key` parameter) varies depending on the algorithm used. For RSR, the key is any string of at least 16 characters. For FF3, the key must be a hex-string consisting of 16, 24, or 32 hexadecimal values, which is then converted by FF3 into a byte-string. A key difference is that FF3 keys can contain bytes with values ranging from 0 to 255, whereas RSR keys typically consist of printable ASCII characters only. It is essential that the private key remains confidential, as the same key must be used for both encryption and decryption.

The format of the tweak ( `p_tweak` parameter), which is used only by FF3 and ignored by RSR, must be a hex-string with either 7 or 8 hexadecimal values (8 for FF3 and 7 for FF3-1, as recommended by NIST). The tweak does not need to be kept secret and can be public. However, the same tweak must be used for both encryption and decryption.

These three optional parameters are applicable to all `encrypt_xxx()` and `decrypt_xxx()` methods.

### 7.5.2. Default settings

Default settings can be defined for the whole session instead of specifying them upon each invocation of `encrypt_xxx()` or `decrypt_xxx()` method.

The default encryption algorithm is RSR. To change it to FF3, make the following call:

`set_encryption_algo('FF3')` . To define a default private key, invoke the `set_encryption_key()`



method. To define a default tweak (FF3 only), invoke the `set_tweak()` method.

### 7.5.3. Encryption / decryption of numbers

To encrypt or decrypt a NUMBER, call the `encrypt_number()` or `decrypt_number()` method with the following parameters:

Name	Usage
p_value	number to encrypt or decrypt (mandatory)
p_precision	number of significant digits (optional)
p_scale	number of decimal digits (optional)

When no precision or scale is given, the precision or scale of the input value is used. When specified, the input number must be compliant with given precision and/or scale. The sign of the original number is preserved (a negative number remains negative after encryption). The same parameters must be specified when encrypting and decrypting.

### 7.5.4. Encryption / decryption of strings

To encrypt or decrypt a string, call the `encrypt_string()` or `decrypt_string()` method with the following parameters:

Name	Usage
p_value	string to encrypt or decrypt (mandatory)
p_len	maximum length of the string (optional)
p_format	format (optional)
p_charset	character set or alphabet (optional)

When no length is specified, the length of the input string is used. Any trailing blank is removed before and after encryption (in the same way as they are removed when stored in an table).

Supplying a format is the shortest way to specify the alphabet. It is a string containing a combination of the following symbols:

Symbol	Corresponding regular expression
a (or any lowercase letter)	[a-z]
A (or any uppercase letter)	[A-Z]

Symbol	Corresponding regular expression
9 (or any digit)	[0-9]
é (or any lowercase accentuated character)	[µàáâãäåæçèéêëìíîïðñòóôõöøùúûüýþ]
É (or any uppercase accentuated character)	[ÀÁÂÃÄÅÆÇÈÉÊËÌÍÎÏÐÑÒÓÔÕÖØÙÚÛÜÝÞ]

The alphabet is just a string containing the valid symbols. Format and alphabet are optional and mutually exclusive. When none of them is specified, the 156 printable ASCII characters are used.

In any case, the input string must be compliant with the supplied parameters. The same parameters must be specified when encrypting and decrypting.

### 7.5.5. Encryption / decryption of dates

To encrypt or decrypt a date (potentially within a given range), call the `encrypt_date()` or `decrypt_date()` method with the following parameters:

Name	Usage
p_value	date to encrypt or decrypt (mandatory)
p_min_date	lower bound of date valid range (optional)
p_max_date	upper bound of date valid range (optional)

When no date range is specified, the default range `[01/01/0001-31/12/9999]` is used. When specified, the input date must fall within the range. The same parameters must be specified when encrypting and decrypting.

### 7.5.6. Encryption / decryption of integers

To encrypt or decrypt an integer within a given range, call the `encrypt_integer()` or `decrypt_integer()` method with the following parameters:

Name	Usage
p_value	integer to encrypt or decrypt (mandatory)
p_min_value	lower bound of integer valid range (mandatory)
p_max_value	upper bound of integer valid range (mandatory)

The input integer must fall within the specified range, which is mandatory. The same parameters must be specified when encrypting and decrypting.