IBM InfoSphere Optim Data Privacy
ODPP User's Guide

IBM's Optim Enterprise Solution

Version 11 Release 3 Modification 0 Fix Pack 6 (March 2018)
This edition applies to version 11, release 3, modification 0, fix pack 6 of IBM InfoSphere Optim Data Masking solution and to all subsequent releases and modifications until otherwise indicated in new editions.

1. Introduction

ODPP is part of the Optim data privacy components, along with the Optim UDFs and Optim data privacy application. This guide provides comprehensive user-type information for using ODPP.

1.1 What is ODPP?

ODPP is the acronym for “Optim Data Privacy Providers”. ODPP is a set of services for performing data masking/privatization. These ODPP services are provided in DLL/library format on a wide variety of platforms and bitness variations including Windows, UNIX, Linux and z/OS in 31-, 32- and 64-bit formats where appropriate.

1.2 ODPP Features

1.2.1 Single extensible API

The ODPP API provides a flexible and extensible means of accessing the data masking services built-in to ODPP.

1.2.2 Dynamic invocation

The entire structure of the ODPP libraries is based upon a very modular design where layers of functionality are abstracted into their own separate library. This allows for a maximum flexibility.

1.2.3 Modular design

All layers of ODPP are implemented in separate abstracted layers that are connected to each other by a loosely coupled generic API

1.2.4 Usability across products

The generic nature of the ODPP API allows it to be used by products that are built in a variety of languages.

1.2.5 Batch and single entity processing.

ODPP processing may be handled on a single entity or user-defined batch sized entity processing level.

1.2.6 Data source independent design

ODPP processing is independent of the data source. The ODPP API is designed to handle the data and not the data source. Data source independence provides the flexibility of supporting unlimited structured and unstructured data sources. The ODPP API caller is responsible for data extraction and presentation to the ODPP via the ODPP API.

1.2.7 Simple and standard representation of data

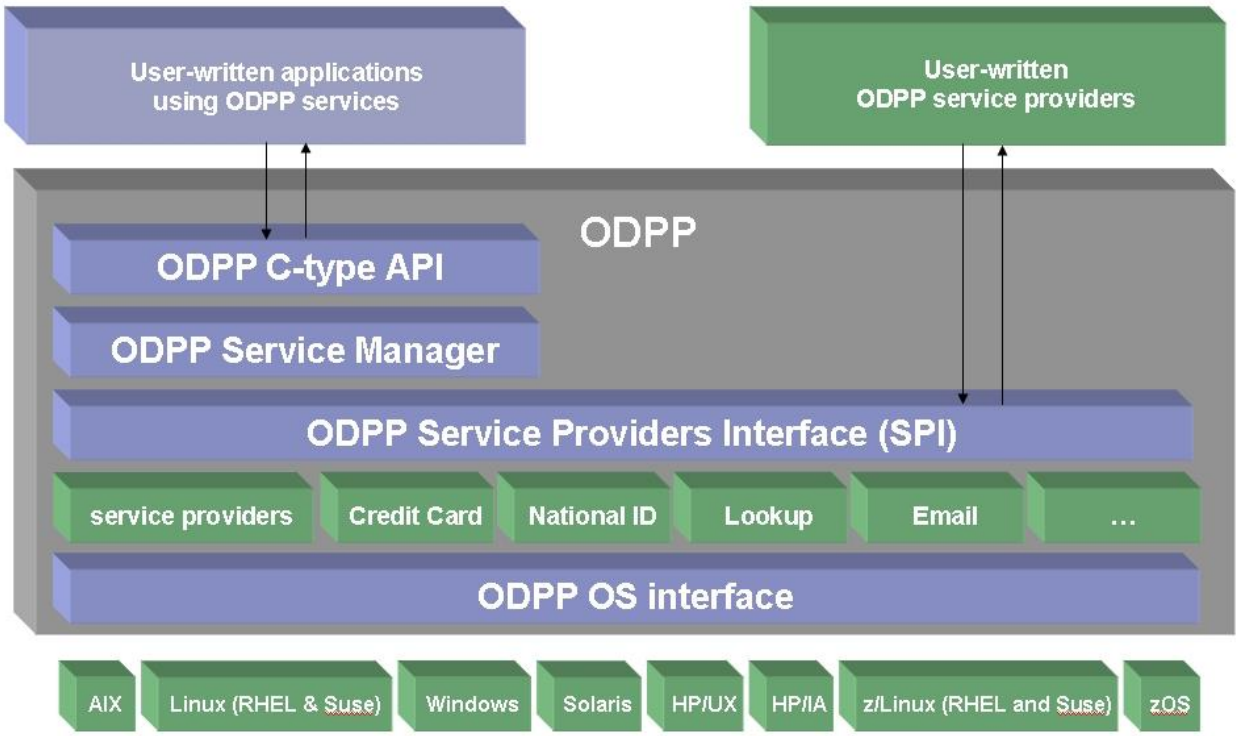
The ODPP input and output data structures simulate data as rows and columns/fields within the rows. Standard data types are used to represent various types of data (e.g. integer, char, null terminated strings, date, time, etc.)

1.3 How is ODPP used?

Applications that require data masking/privatization will typically call into the ODPP services via the C-type API externalized functions.

1.4 The ODPP integrated architecture

The following graphic depicts the ODPP integrated architecture:



1.4.1 User-written applications using ODPP services

This layer depicts the user-written applications (e.g. the Optim solution, Optim Column Map Exits, etc.,) that are written to interface with and use the ODPP data masking services. The medium of communication between the user-written application and ODPP is the ODPP C-type API

1.4.2 User-written ODPP service providers

This layer depicts the user-written service provider. This abstracted layer allows a technically proficient client to write ODPP-type service providers that can then be plugged into the ODPP architecture and accessed by the user-written applications using the ODPP services. The medium of communication between the user-written ODPP service providers and ODPP is the ODPP C-type Service Provider Interface (SPI). This is the exact same architecture that is used for the ODPP-provided and built-in service providers that are furnished with the ODPP package.

1.4.3 ODPP

This encompassing layer depicts all of the separate layers within ODPP including:
ODPP C-type API (The C API also serves the Java API through JNI)
ODPP Service Manager
ODPP Service Provider Interface (SPI)
Service Providers
ODPP OS (Operating System) interface

1.4.4 ODPP C-type API

This layer depicts the ODPP C-type API (Application Programming Interface) for a user-written application to gain access to the ODPP data masking services. This powerful yet simplistic API allows the user to:
- Initialize the ODPP framework
- Initialize the individual ODPP Service Provider for the specific data masking services
- Process a row or a batch of rows of data to be masked

- Terminate the individual ODPP Service Provider
- Terminate the ODPP framework.

These 5 simple functions are the functions most often used from an application seeking to use the ODPP data masking services.

A Java API is also provided. The Java API has a JNI layer that uses the C API.

1.4.5 ODPP Service Manager

This layer depicts the ODPP Service Manager that manages the interface and material being transported from the ODPP API layer to the individual ODPP Service Providers.

1.4.6 ODPP Service Providers Interface (SPI)

This layer depicts the ODPP C-type Service Providers Interface. This interface is used as the interface point to and from each ODPP service provider.

1.4.7 Service Providers, Credit Card, National ID, Lookup, Email, …

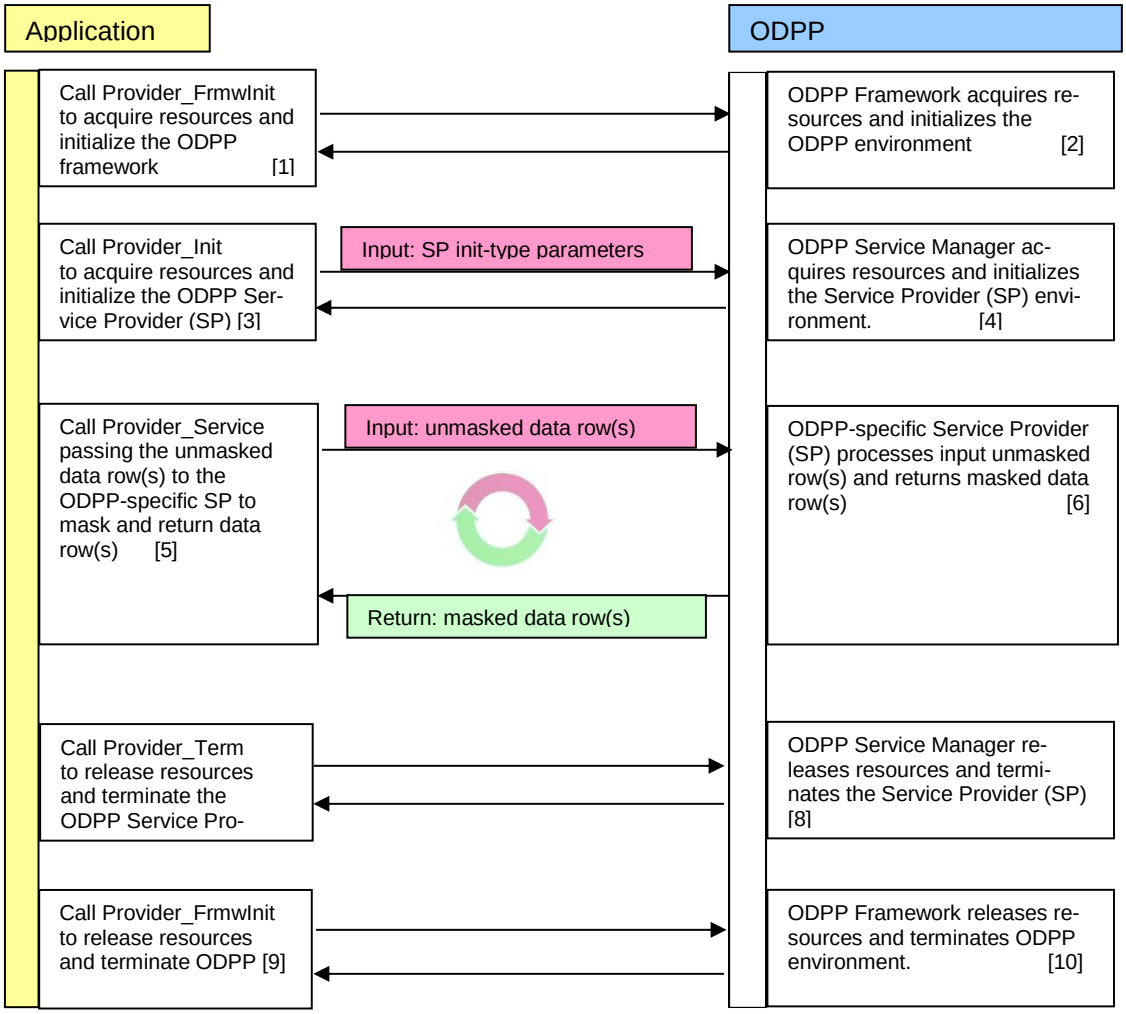
This layer depicts the ODPP Service Providers.

1.4.8 ODPP OS interface

This layer depicts the ODPP operating system interface. This layer handles all of the operating system-specific functions for ODPP for each of the operating platforms and bitness variations supported by ODPP.

1.5 ODPP Flow Diagram

The following graphic depicts the flow of information from the Application that is using ODPP data masking services through ODPP via the simple yet powerful ODPP API.



In the above graphic:

[1] The Application makes an initial call to ODPP via the ODPP API Provider_FrmwInit function. This allows ODPP to acquire its initial resources and to initialize the ODPP framework for servicing the needs of the application to provide data masking services.

[2] ODPP receives control, loads the other framework-related libraries, acquires resources, and initializes ODPP to provide data masking services for any of the available ODPP data masking Service Providers.

[3] The Application prepares the input structure, identifying the specific ODPP Service Provider needed and the parameters to drive this SP and then calls ODPP via the ODPP API Provider_Init to initialize the specific ODPP provider. The expected return is a token identifier for use in subsequent calls to ODPP for this specific service provider.

[4] ODPP receives control, interprets the input structure, acquires resources, loads the specific ODPP Service Provider library, and initializes the service provider for data masking. ODPP returns a token identifier to the Application that must be passed into ODPP for subsequent service calls to distinguish this ODPP Service Provider activity from any other providers currently operating in the same process.

[5] The Application prepares the input structure with the input buffer(s) for the data to be masked as well as the token identifier returned from the previous call and then calls ODPP via the ODPP API Provider_Service to mask the identified data in the input buffer(s)

[6] ODPP receives control, interprets the token identifier, interprets the input buffer(s), masks the data and either returns it in the input buffer(s) or optionally in the output buffer(s) the masked data.

NOTE: At this point, the flow continues between [5] and [6] activities until the Application detects the end of the data to be masked.

[7] The Application passes the token identifier of the ODPP Service Provider, that has been used, and then calls ODPP via the ODPP API Provider_Term to terminate the specific ODPP Service Provider for this application's usage.

[8] ODPP receives control, interprets the token identifier, releases resources and terminates the ODPP Service Provider for the application via the indicated token.

[9] The Application calls ODPP via the ODPP API Provider_FrmwTerm to allow the ODPP framework to be terminated.

[10] ODPP receives control, releases resources and terminates the ODPP framework environment.

2. Documentation

Documentation is available from IBM Knowledge Center. The Knowledge Center provides the most current content as it can be quickly updated. Refer to the section titled "Optim data privacy provider library" at the following link on Knowledge Center.

http://www.ibm.com/support/knowledgecenter/SSMLQ4_11.3.0/

3. ODPP Installation

Complete details on installing ODPP is available from IBM Knowledge Center. Refer to the section titled "Overview for installing the Optim data privacy providers" at the following link on Knowledge Center.

http://www.ibm.com/support/knowledgecenter/SSMLQ4_11.3.0/

4. User-Defined Functions (UDFs)

ODPP supports User-Defined Functions (UDFs). ODPP-type UDF's allow you to dynamically mask data within the framework of the DBMS server. The following sections detail the UDF names as well as the input- and return-type details for each of the supported DBMS.

4.1 ODPP UDF Names

There are two types of ODPP UDF name groupings. They are known as Overloaded UDFs and Specific UDFs

4.1.1 Overloaded UDFs

The overloaded UDFs allow you to specify simply **OptimMask** as the name of the UDF for those situations where the input and return data types are identical and match the 1st parameter or argument within each UDF.

4.1.2 Specific UDFs

The specific UDFs are used in situations where an overloaded and simplified OptimMask UDF cannot be used. These include:

1. When the return data type is not the same as the input data type
2. When the UDF has multiple inputs
3. When the input and return data types are not found in the supplied overloaded UDFs.

4.1.3 DB2 z/OS

The following table groupings identify the ODPP-type DB2 z/OS overloaded and specific UDF names. The 1st column specifies the UDF name, the 2nd column specifies the input data type and the 3rd column specifies the return data type.

4.1.3.1 Overloaded UDFs

UDF Name	Input data type	Return data type
OptimMask	BIGINT	BIGINT
OptimMask	NUMERIC(18,0)	NUMERIC(18,0)
OptimMask	VARCHAR(32704)	VARCHAR(32704)

4.1.3.2 Specific UDFs

UDF Name	Input data type	Return data type
OptimMaskDate	DATE	DATE
OptimMaskDouble	DOUBLE	DOUBLE
OptimMaskInt64	BIGINT	BIGINT
OptimMaskNum	NUMERIC(18,0)	NUMERIC(18,0)
OptimMaskNum18S2	NUMERIC(18,2)	NUMERIC(18,2)
OptimMaskNum18S2	NUMERIC(18,2)	NUMERIC(18,2)
OptimMaskNum18S2	NUMERIC(18,2)	NUMERIC(18,2)
OptimMaskStrEBCDIC	VARCHAR(32704)	VARCHAR(32704)
OptimMaskTimeStamp	TIMESTAMP	TIMESTAMP

4.1.4 Microsoft SQL Server

The following tables identify the Microsoft SQL Server UDF names. The 1st column specifies the UDF name, the 2nd column specifies the input data type, and the 3rd column specifies the return data type.

4.1.4.1 Single-argument UDFs

The UDFs in the following table support only a single data argument with the data privacy provider input syntax.

UDF Name	Input data type	Return data type
OptimMaskDate	DATE	DATE
OptimMaskDouble	FLOAT	FLOAT
OptimMaskInt64	BIGINT	BIGINT
OptimMaskInt64Date	DATE	BIGINT
OptimMaskInt64Double	FLOAT	BIGINT
OptimMaskInt64NStr	NVARCHAR(800)	BIGINT
OptimMaskInt64Time	TIME	BIGINT
OptimMaskInt64Timestamp	DATETIME2	BIGINT
OptimMaskNStr	NVARCHAR(800)	NVARCHAR(800)
OptimMaskNum18S2	DECIMAL(18,2)	DECIMAL(18,2)
OptimMaskNum18S4	DECIMAL(18,4)	DECIMAL(18,4)
OptimMaskNum18S6	DECIMAL(18,6)	DECIMAL(18,6)
OptimMaskTimestamp	DATETIME2	DATETIME2

Note: 800 characters is the maximum size of an email address and the longest NVARCHAR data type that is supported. You may create custom UDFs to support other lengths.

4.1.4.2 Multi-argument UDFs

Use the multi-argument UDFs in the following table when you have more than a single column, expression, or literal as input to the UDF.

UDF Name	Number of arguments	Input data type	Return data type
OptimMaskInt64NStr2	2	NVARCHAR(800)	BIGINT
OptimMaskInt64NStr3	3	NVARCHAR(800)	BIGINT
OptimMaskNStr2	2	NVARCHAR(800)	NVARCHAR(800)
OptimMaskNStr3	3	NVARCHAR(800)	NVARCHAR(800)

4.1.4.3 Data Type Mapping

The following table shows the supported ODPP FLDDEF data types for each SQL Server parameter data type. It also shows the corresponding C# data type when creating custom UDFs.

SQL Server Data Type	ODPP FLDDEF Data Type	C# Data Type
BIGINT	LONG_LONG U_LONG_LONG	Nullable<Int64>
DATE	DATE	Nullable<DateTime>
DATETIME2	TIMESTAMP	Nullable<DateTime>
DECIMAL	DECIMAL_370	Nullable<Decimal>
FLOAT	DOUBLE	Nullable<Double>
NVARCHAR	WCHAR WVARCHAR_SZ DATETIME_WCHAR DATETIME_WSZ	String
TIME	TIME	Nullable<TimeSpan>

4.1.4.4 Custom UDFs

There may be cases when you want to create a custom UDF that supports different arguments and lengths than the UDFs supplied. The samples directory contains sample code for a C# assembly that you can use as an example to code your own UDFs if the supplied UDFs do not meet your requirements. To create a custom UDF, you must create a C# assembly that uses the OptimMask class from the IBM.InfoSphere.ODPP namespace.

4.1.4.4.1 Public APIs

The OptimMask constructor is used to create an instance of the OptimMask object and supply the provider string:

```
public OptimMask(String providerString)
```

The addArgument method is used to add any source arguments to be masked:

```
public void addArgument(Nullable<Decimal> argument,
                        int precision,
                        short scale)
public void addArgument(Nullable<Double> argument)
public void addArgument(Nullable<Int64> argument)
public void addArgument(String argument)
public void addArgument(Nullable<TimeSpan> argument)
public void addDateArgument(Nullable<DateTime> argument)
public void addTimestampArgument(Nullable<DateTime> argument)
```

The maskScalar method is used to retrieve the masked result. For most data types the masked result is returned in an *out* parameter, but for string data the parameter is a *ref* StringBuilder. The StringBuilder object must be instantiated with the maximum capacity of the result prior to calling maskScalar:

```
public void maskScalar(out Nullable<Decimal> result,
                       int precision,
                       short scale)
public void maskScalar(out Nullable<Double> result)
public void maskScalar(out Nullable<Int64> result)
public void maskScalar(ref StringBuilder result)
public void maskScalar(out Nullable<TimeSpan> result)
public void maskScalarDate(out Nullable<DateTime> result)
public void maskScalarTimestamp(out Nullable<DateTime> result)
```

4.1.4.4.2 Sample Files

The samples directory contains the following sample files for creating custom UDFs:

File name	Description
MSSQLUserMask.cs	C# source file that defines several custom UDFs
MSSQLUserMaskInstall.sql	SQL script to register the compiled C# assembly and create the UDFs
MSSQLUserMaskTest.sql	SQL script to test the custom UDFs
MSSQLUserMaskUninstall.sql	SQL script to uninstall the custom UDFs

4.1.4.4.3 Compiling the C# Assembly

To compile the C# assembly, you must use the .NET Framework SDK which can be downloaded for free from Microsoft. The following command invokes csc, the C# compiler. It specifies that the output should be a DLL named MSSQLUserMask.dll, that it references the IOQMSSQLUD-FdotNET.dll library that contains the OptimMask class, and that the C# source is in a file named MSSQLUserMask.cs:

```
csc /target:library /out:MSSQLUserMask.dll /R:IOQMSSQLUDFdotNET.dll
MSSQLUserMask.cs
```

4.1.4.4.4 Registering the C# Assembly and Custom UDFs

Once the C# source file has been compiled, the installation SQL script can be executed using the SQL Server sqlcmd utility, for example:

```
sqlcmd -E -S Server\Instance -d Database -I -i MSSQLUserNameInstall.sql
```

4.1.4.4.5 Testing the Custom UDFs

Once the custom UDFs have been created, the test SQL script can be executed, for example:

```
sqlcmd -E -S Server\Instance -d Database -I -i MSSQLUserNameTest.sql
```

4.1.5 Netezza

The following table groupings identify the ODPP-type Netezza overloaded and specific UDF names. The 1st column specifies the UDF name, the 2nd column specifies the input data type and the 3rd column specifies the return data type.

4.1.5.1 Overloaded UDFs

UDF Name	Input data type	Return data type
OptimMask	BIGINT	BIGINT
OptimMask	DOUBLE	DOUBLE
OptimMask	NUMERIC(18,0)	NUMERIC(ANY
OptimMask	NVARCHAR(800)	NVARCHAR(ANY)
OptimMask	TIMESTAMP	TIMESTAMP
OptimMask	VARCHAR(800)	VARCHAR(ANY)

4.1.5.2 Specific UDFs

UDF Name	Input data type	Return data type
OptimMaskDate	VARARGS	DATE
OptimMaskDouble	VARARGS	DOUBLE
OptimMaskInt64	VARARGS	BIGINT
OptimMaskNum	NUMERIC(18,0)	NUMERIC(ANY)
OptimMaskNum18S2	VARARGS	NUMERIC(ANY)
OptimMaskNum18S4	VARARGS	NUMERIC(ANY)
OptimMaskNum18S6	VARARGS	NUMERIC(ANY)
OptimMaskStrLatin	VARARGS	VARCHAR(ANY)
OptimMaskStrUTF8	VARARGS	NVARCHAR(ANY)
OptimMaskTimestamp	VARARGS	TIMESTAMP

4.1.6 Oracle

The following table grouping identifies the ODPP-type Oracle and specific UDF names. The 1st column specifies the UDF name, the 2nd column specifies the input data type and the 3rd column specifies the return data type.

NOTE: Unlike some of the other DBMS, Oracle does not allow for UDFs to be overloaded. Therefore, the following UDFs support only a single data argument plus the ODPP input parameter string.

4.1.6.1 Specific UDFs

UDF Name	Input data type	Return data type
OptimMaskChar1	VARCHAR2	VARCHAR2
OptimMaskCharNum	VARCHAR2	NUMBER
OptimMaskDate	DATE	DATE
OptimMaskDouble	DOUBLE	DOUBLE
OptimMaskNum	NUMBER [1]	NUMBER [1]
OptimMaskNStr	NVARCHAR2	NVARCHAR2
OptimMaskStr	VARCHAR2	VARCHAR2
OptimMaskTimestamp	TIMESTAMP	TIMESTAMP
OptimMaskTS1	TIMESTAMP	TIMESTAMP

[1] The ODPP FLDDEF determines the precision and scale of the input and return data.

4.1.6.2 Generic multi data argument UDFs

UDF Name	Number of arguments	Input data type	Return data type
OptimMaskStr2Int	2	VARCHAR2	PLS_INTEGER
OptimMaskStr2	2	VARCHAR2	VARCHAR2
OptimMaskStr3	3	VARCHAR2	VARCHAR2
OptimMaskStr4	4	VARCHAR2	VARCHAR2
OptimMaskStr5	5	VARCHAR2	VARCHAR2
OptimMaskStr6	6	VARCHAR2	VARCHAR2

The above ODPP UDFs should be used when you have more than a single column/expression/literal as input to the ODPP UDF. When using any of the OptimMaskStr-type multi data argument UDFs and the input data argument is not a VARCHAR2 data type, you must use an Oracle CAST function to convert the non-VARCHAR2 type to VARCHAR2.

4.1.6.3 Custom multi data argument UDFs

For those situations where you want to create your own multi data argument UDFs, ODPP provides a UDF template/skeleton, in source form, so you can easily create your own multi data argument UDFs. You might use this solution when:

1. You are concerned that using any of the above generic multi data argument UDFs, based upon VARCHAR2, will not perform as well as a custom UDF defined with input data arguments based upon the actual built-in Oracle data types or the generic.

- or -

2. The generic UDFs do not fit your requirements

As Oracle does not support overloading of UDFs, a separate UDF is needed for each variation in datatype, number of arguments, order of arguments or return type.

For example, the following two UDFs may seem similar in that they each have two input data arguments where the 1st is VARCHAR2 and the 2nd is NUMBER followed by the ParamString, which contains the ODPP input parameters. However, the 1st UDF returns a VARCHAR2 and the 2nd UDF returns a NUMBER. Therefore this results in two unique UDFs:

```
OptimMask1    (     Arg1 IN Varchar2,
                    Arg2 IN NUMBER
                    ParamString in varchar2) RETURN VARCHAR2 ...

OptimMask2    (     Arg1 IN Varchar2,
                    Arg2 IN NUMBER
                    ParamString in varchar2 ) RETURN NUMBER ...
```

It simply is not practical or feasible for ODPP to provide a UDF for each permutation of arguments given that ODPP supports six unique Oracle built-in data types and your UDF input argument needs could be many columns.

To solve this problem, ODPP supplies a UDF template/skeleton, in source form, to allow you create your own custom ODPP Oracle UDFs with multiple input arguments of any supported Oracle built-in type and to supply this information to ODPP via the standard mask function "OptimMaskUDF".

The OptimMaskUDF function works on a specially built multi argument infrastructure to handle these situations. The primary advantage of this architecture is the simplistic and straight-forward interface for managing the input/output and masking the data means a custom multi data argument UDF can be created with less code and less time for you to develop.

4.1.6.3.1 OptimMaskUDF

The following is the definition of the OptimMaskUDF function with a description of each parameter:

```
retval OptimMaskUDF(OCIExtProcContext  *pCtx,
                    COracleData        **pInData,
                    int                  iArgCntIn,
                    COracleData        *pOutData,
                    int                  iArgCntOut);
```

Member	Data Type		Description
pCtx	OCIExtProcContext	IN	Oracle Call Interface context pointer. Sent by Oracle DB to all the UDFs
pInData	COracleData	IN	Pointer to an array of Data Class pointers carrying the input data
iArgCntIn	Int	IN	Count of arguments in array pInData
pOutData	COracleData	IN/OUT	Pointer to Data Class carrying output data
iArgCntOut	Int	IN	Count of output arguments. Currently set to 1

4.1.6.3.2 How to write a custom multi data argument UDF

For purposes of understanding how to write a custom multi data argument UDF, we will use the example where we have two input arguments. One is a VARCHAR2 and the other is a NUMBER and the UDF is to return a NUMBER. For purposes of this example, we will call this UDF, CustMask1. Use the following 12 steps to create CustMask1:

1. Open for editing **ioqoptimmaskorafunc.cpp** and add the CustMask1 function to the existing multi data argument UDFs already contained in this source file. You could start by copying:
OptimMaskChar1
to
CustMask1.

2. Change the function name
OptimMaskChar1
to
CustMask1

3. Change the return from CustMask1 from
char *
to
OCINumber *
as CustMask1 will return an Oracle NUMBER data type

4. In the function prototype for CustMask1, add two more parameters for the 2nd data argument. Since the 2nd argument is a NUMBER, you will only need to specify the 1st additional parameter as for the 2nd argument as OCINumber where you name it pArg2 and follow that with a short

named sArg2Ind. These additional two parameters would be placed just before the char *pParamString parameter which is the parameter string that carries the ODPP input syntax. Also, be sure to change the return from CustMask1 from the char * to OCINumber * as we indicated above this function would return a NUMBER. Changes and additions to CustMask1 from the copy of OptimMaskChar1 are bolded below.

CustMask1 would appear as follows with the additional parameters shown below in bold:

```
OCINumber *CustMask1(OCIExtProcContext *pCtx, // OCI Ext. Proc. Context ptr
                     char *pArg1,             //1st Data argument
                     short sArg1Ind,          //1st Arg - NULL indicator
                     int iArg1Len,            //1st Arg - string length
                     int iArg1Cid,            //1st Arg - character set identifier
                     int iArg1Cform,          //1st Arg - character set form
                     OCINumber *pArg2,        //2nd Data argument
                     short sArg2Ind,          //2nd Arg - NULL indicator
                     char *pParamString,      //ODPP input provider syntax string (OIPSS)
                     short sPSInd,            //OIPSS - NULL indicator
                     int iPSLen,              //OIPSS - string length
                     int iPSCid,              //OIPSS - character set identifier
                     int iPSCform,            //OIPSS - character set form
                     short *pRetInd,          //function return NULL indicator
                     int *pRetLen)            //function return length (only for strings)
                                              //strings = VARCHAR2 or CHAR
```

NOTE: If the 2nd parameter had been a VARCHAR, then you would need the same parameters as seen above for the 1st Arg but you would need to change the 1 to a 2. If the 2nd parameter had been a DATE or TIME, then you would need the same two parameters as we used for the NUMBER data type.

The following parameters apply for different input data types:

Parameter	Data Type	Example
Data	All supported	char *, OCINumber *, OCIDateTime *, etc.,
Null Indicator	All supported	char *, OCINumber *, OCIDateTime *, etc.,
Length	Only character	char *
Character Set ID	Only character	char *
Character Set form	Only character	char *

4. In the local variables appearing after the CustMask1 prototype, change
char *pRetStr;
to
OCINumber *pRetNum;

5. Create an array of COracleData class pointers equal to the number of arguments which in this case is 3. You would simply change:
COracleData *pDataInArr[2];
to
COracleData *pDataInArr[3];

6. Allocate an appropriate class object for each input data pointer. In this case, you already have a COracleData pointer for COracleString for the 1st argument. Now, you need to simply create a new entry for the 2nd input argument of NUMBER:

```
//Allocate the input data classes
//This is a Number class because the 2nd input parameter is a NUMBER data type
```

***(pDataInArr + 1) = (COracleData*) new COracleNumber(pArg2, sArg2Ind);**

If the 2nd parameter had been a Date or Time type data argument, you would create a DateTime class object as follows:

//Allocate the input data classes
//This is a Number class because the 2nd input parameter is a Date or Time data type
***(pDataInArr + 1) = (COracleData*) new COracleDateTime(pArg2, sArg2Ind);**

NOTE1: You will notice that the pDataInArr pointer is at +1 to indicate the 2nd parameter. For each additional parameter, this index would be incremented by one. For example, for a 3rd parameter this would be specified as pDataInArr+3, etc.,.

NOTE2: You will notice that the input arguments into the COracle-type class method have the argument numbers incremented in both parameters to match the argument parameters within the CustMask1 specification. In this case, the 2nd parameter parameters are pArg2 and sArg2Ind.

NOTE3: You will notice that the last index in the pDataInArr array is always COracleString type for the ODPP provider syntax string as it is the last argument into any ODPP UDF.

7. Change the allocation for the output data class. In the copied from function, OptimMaskChar1 the output data class is COracleString as the return type is VARCHAR2. In CustMask1, the return data type is NUMBER so we need to change this accordingly. Change:
pDataOutArr = (COracleData*) new COracleString(NULL, 0, 0, 0, 0);
to
pDataOutArr = (COracleData*) new COracleNumber(pRetNum, 0);

8. Change the 3rd parameter in the call to OptimMaskUDF from 2 parameters to 3 parameter to account for the 2nd input data argument. Change:
retVal = OptimMaskUDF(pCtx, pDataInArr, 2, pDataOutArr, 1);
to
retVal = OptimMaskUDF(pCtx, pDataInArr, 3, pDataOutArr, 1);

9. Change masked data return data class from:
pDataOutArr->GetString(&pRetStr, pRetInd, pRetLen, NULL, NULL);
to
pDataOutArr->GetOciNumber(&pRetNum, pRetInd);

10. Change the section that deletes the objects created earlier to also delete the Number-type object created for the 2nd input data argument. Add:
delete *(pDataInArr + 2);
after
delete *(pDataInArr + 1);

11. Change the function return from:
return pRetStr;
to
return pRetNum;

This completes the editing changes in **ioqoptimmaskorafunc.cpp.** Please be sure to save it before moving on.

4.1.6.3.3 Complete CustMask1 method

The following is a complete listing of the CustMask1 method based upon the editing changes you made in the previous section:

```
OCINumber* CustMask1(  OCIExtProcContext *pCtx, //OCI External Procedure context pointer
                       char *pArg1,             //1st Data argument
                       short sArg1Ind,          //1st Arg - NULL indicator
                       int iArg1Len,            //1st Arg - string length
                       int iArg1Cid,            //1st Arg - character set identifier
                       int iArg1Cform,          //1st Arg - character set form
                       OCINumber *pArg2,        //2nd Data argument
                       short sArg2Ind,    /     //2nd Arg - NULL indicator
                       char *pParamString,      //ODPP service provider-syntax string (OIPSS)
                       short sPSInd,            //OIPSS - NULL indicator
                       int iPSLen,              //OIPSS - string length
                       int iPSCid,              //OIPSS - character set identifier
                       int iPSCform,            //OIPSS - character set form
                       short *pRetInd,          //function return NULL indicator
                       int *pRetLen)            //function return length (only for strings)
                                                //strings = VARCHAR2 or CHAR
{
  RETVAL        retVal          = ODPPFAILURE;
  OCINumber     *pRetNum        = NULL;

  //Declare enough elements for the input arguments, here there are 3 inputs:
  // The first two are the input data arguments and the last is the ODPP service provider-syntax
  //1. Input Argument (pArg1)                 type=VARCHAR2,
  //2. Input Argument (pArg2)                 type=NUMBER,
  //3. Parameter String (pParamString)        type=VARCHAR2
  COracleData *pDataInArr[3];

  //Declare a pointer for the output
  COracleData *pDataOutArr;

  //Allocate the input data classes
  //This is a string class because the input is an Oracle VARCHAR2 data type
  *(pDataInArr + 0) = (COracleData*) new COracleString(pArg1, sArg1Ind, iArg1Len, iArg1Cid, iArg1Cform);

  //This is a number class because the input is an Oracle NUMBER data type
  *(pDataInArr + 1) = (COracleData*) new COracleNumber(pArg2, sArg2Ind);

  //This is a string class because the ODPP service proivder syntax string is always a char
  *(pDataInArr + 2) = (COracleData*) new COracleString(pParamString, sPSInd,
                        iPSLen, iPSCid, iPSCform);

  //Allocate the output data class, this is a number class because it returns an Oracle NUMBER
  pDataOutArr = (COracleData*) new COracleNumber(NULL, 0);

   //Call the OptimMaskUDF function to mask the input data
  retVal = OptimMaskUDF(pCtx, pDataInArr, 3, pDataOutArr, 1);
  //Is the UDF returning a NULL value?
  if(ODPP_UDF_ERR_ORA_NULL_VALUE == retVal)
  {
    //Yes, then set the return indicator accordingly and return
    *pRetInd = OCI_IND_NULL;
    *pRetLen = 0;
  }
  else
  {
    //No, the get the output
    //Get the output data
    pDataOutArr->GetOciNumber(&pRetNum, pRetInd);
  }
```

```
  //Before returning, free the objects previously allocated
  delete *(pDataInArr + 0);
  delete *(pDataInArr + 1);
  delete *(pDataInArr + 2);
  delete pDataOutArr;

  //Return either the masked NUMBER or NULL
  return pRetNum;
}
```

4.1.6.4 Building the custom multi data argument UDF

4.1.6.4.1 Preparing to build custom multi data argument UDFs

To ensure a successful build, copy the contents of the platform-specific folder (e.g. Images32\win_udf_ora) into a new folder. For purposes of this discussion we will call it ORAUDFBUILD.

This new folder should include everything from the platform-specific folder and sub-folders without the subfolder structure. Basically, flattening out all of platform-specific folder contents into a single folder containing:
1. All ODPP binaries and files from the BIN folder
2. All files from the INCLUDE folder
3. All files present in the SAMPLES folder

4.1.6.4.2 Windows

You have two options for Windows. You may either:
a. Use the supplied and preconfigured Visual Studio 2008 project ioqoraudf.vcproj. You should carefully review this project and change according to your own needs,
- or -
b. You may create, from scratch, a Visual Studio 2008 project following these steps:

1. Create a Visual Studio 2008 SP1 (version 9.0.30729.1) project with a "Configuration Type" as Dynamic Library (.dll).

2. Add all six ODPP-provided files for the ODPP custom UDFs to the project including:
   ioqoptimmaskorafunc.h
   iocoracledata.h
   ioqoraudfinclude.h
   ioqoptimmaskorafunc.cpp
   ioqoraudfwinexp.def

3. Add the following paths to the project:
   - Src = folder where all the public header files, .cpp files and ODPP binaries are located
   - OCI = folder where Oracle oci.h is located with other Oracle headers typically this is $Oracle_HOME\OCI\include

4. Add the following binaries to the link step in the project:
   - odpporaudf.9.1.lib = ODPP main library exporting OptimMaskUDF
   - ioqoracledata.obj = object file for COracleData class implementations

5. Build the project using the Visual Studio with the following command line:

DEVENV.com *<solution-file>* **/rebuild "***<solution-configuration>***"**
/project *<project>* **/useenv /out /buildlog.txt**

<solution-configuration> could be Debug or Release or DevRelease if you are using **ioqoraudf.vcproj**

The buildlog.txt will contain the Visual Studio 2008 project output. Please be sure to review it for any errors.

4.1.6.4.3 Linux/UNIX

1. Compile **ioqoptimmaskorafunc.cpp** using the platform specific C++ compiler. This can be a native compiler or a standard one like gcc.

2. Link the object file from step 1 along with **ioqoracledata.so** into a shared library with **libODPPORAUdf.so** as a dependency.

You may perform the above two steps using either the ODPP provided script or manually per your own needs.

4.1.6.4.3.1 Building on Linux/UNIX using makeoraudf.sh

The ODPP-provided **makeoraudf.sh** script can be used to build the custom Oracle UDFs on all ODPP-supported Linux/UNIX platforms.

At a command prompt you simply need to execute makeoraudf.sh. For example:

/> ./makeoraudf

It will ask you for 4 input parameters:

1. **Platform**:
This can be any one of the following, please note that this is a case sensitive input:
RHEL32, RHEL64, SUSE32, SUSE64, AIX32, AIX64, HPUX32, HPUX64, HPIA32, HPIA64, SUN32, SUN64, ZRHL31, ZRHL64, ZSUS31, ZSUS64.

2. **Path of ODPP binraies**
This is the path to the directory where you copied all the ODPP binaries, header files, source files and object files.

3. **Path of Oracle OCI.H**
This is the directory where the Oracle header OCI.H is located.

4. **Path of compiler**
This is directory containing the platform native compiler.
For example, on RedHat Linux, the compiler is *gcc*, while for HPUX it is *aCC*

For a list of supported compilers see the ODPP Release Notes or the ODPP Developer's Guide available in the DOC folder.

On successful completion, a directory specific to the Platform specified in the build is created and will contain the custom Oracle ODPP UDF shared library named:
libioqoraudf.so.

Note: The makeoraudf.sh script always creates a shared library with the name ***libioqoraudf.so*** although this could be changed to meet your naming standards.

4.1.6.4.3.2 Building on RedHat Linux using the manual method

1. The following command will compile the ioqoptimaskorafunc.cpp source into the object file named ioqoptimmaskorafunc.o:

gcc -m32 -Wall -fPIC -c -D CPLUSPLUS -D GNU_SOURCE
-I$SRC -I$OCI_INC ioqoptimmaskorafunc.cpp -o ioqoptimmaskorafunc.o

2. The following command will link the object ioqoptimmaskorafunc.o into the shared library named libiororaudf.so:

gcc -m32 -Wl,-soname,libioqoraudf.so ioqoracledata.o
ioqoptimmaskorafunc.o -o libioqoraudf.so
-L$SRC -L$SRC -lpthread -lODPPORAUdf

where:

$SRC
is the directory where all of the ODPP-provided public header files, .cpp files and ODPP binaries are also located. For the above examples, this should also be the current folder.

$OCI_INC
is the folder where the Oracle oci.h is located with the other Oracle headers.
Typically, this is &Oracle_HOME\OCI\include

libioqoraudf.so
is the name of the output shared object library. You may change this to meet your needs.

For all other platforms, you will need to develop your own manual procedures, or simply use the ODPP-provided script from above in section Building on Linux/UNIX using makeoraudf.sh

4.1.7 Teradata

The following table groupings identify the ODPP-type Teradata overloaded and specific UDF names. The 1st column specifies the UDF name, the 2nd column specifies the input data type and the 3rd column specifies the return data type. For VARCHAR types, the parameter after the size of the VARCHAR indicates the CHARACTER SET

4.1.7.1 Overloaded UDFs

UDF Name	Input data type	Return data type
OptimMask	BIGINT	BIGINT
OptimMask	DATE	DATE
OptimMask	DECIMAL(18)	DECIMAL(18)
OptimMask	DOUBLE PRECISION	DOUBLE PRECISION
OptimMask	TIMESTAMP(6)	TIMESTAMP(6)
OptimMask	VARCHAR(800) LATIN	VARCHAR(800) LATIN
OptimMaskInt64	DATE	BIGINT
OptimMaskInt64	DECIMAL(18)	BIGINT
OptimMaskInt64	DOUBLE PRECISION	BIGINT
OptimMaskInt64	TIME(6)	BIGINT
OptimMaskInt64	TIMESTAMP(6)	BIGINT
OptimMaskInt64	VARCHAR(800) UNICODE	BIGINT
OptimMaskInt64	VARIANT_TYPE	BIGINT
OptimMaskStrLatin	VARIANT_TYPE	VARCHAR(800) LATIN
OptimMaskStrUTF16	VARIANT_TYPE	VARCHAR(800) UNICODE

4.1.7.2 Specific UDFs

UDF Name	Input data type	Return data type
OptimMaskDate	DATE	DATE
OptimMaskDouble	DOUBLE PRECISION	DOUBLE PRECISION
OptimMaskInt64	BIGINT	BIGINT
OptimMaskNum	DECIMAL(18)	DECIMAL(18)
OptimMaskNum18S2	DECIMAL(18,2)	DECIMAL(18,2)
OptimMaskNum18S4	DECIMAL(18,4)	DECIMAL(18,4)
OptimMaskNum18S6	DECIMAL(18,6)	DECIMAL(18,6)
OptimMaskStrLatin	VARCHAR(800) LATIN	VARCHAR(800) LATIN
OptimMaskStrUTF16	VARCHAR(800) UNICODE	VARCHAR(800) UNICODE
OptimMaskTable	VARIANT_TYPE	VARYING COLUMNS
OptimMaskTimestamp	TIMESTAMP(6)	TIMESTAMP(6)

4.1.7.3 Data Type Mapping

The following table shows the supported ODPP FLDDEF Data Types for each Teradata parameter data type.

Teradata Parameter Data Type	Supported ODPP FLDDEF Data Types
BIGINT	LONG_LONG U_LONG_LONG
DATE	DATE
DECIMAL	DECIMAL_370 (except OptimMaskInt64)[1] LONG_LONG (OptimMaskInt64 only) U_LONG_LONG (OptimMaskInt64 only)
DOUBLE PRECISION	DOUBLE
TIME	TIME
TIMESTAMP	TIMESTAMP
VARCHAR CHARACTER SET LATIN	CHAR VARCHAR_SZ DATETIME_CHAR DATETIME_SZ
VARCHAR CHARACTER SET UNICODE	WCHAR WVARCHAR_SZ DATETIME_WCHAR DATETIME_SZ

1. Normally if the UDF accepts a DECIMAL argument, then the corresponding FLDDEF data type must be DECIMAL_370. But the OptimMaskInt64 function is an exception. When that UDF receives a DECIMAL argument, the corresponding FLDDEF data type must be LONG_LONG or U_LONG_LONG. This is a performance optimization, since Teradata's internal representation for a DECIMAL(18) is the same as an ODPP LONG_LONG and requires no conversion.

5. ODPP Service Providers

The following sub-sections detail the individual ODPP Service Providers currently available with ODPP.

5.1 What is an ODPP Service Provider (SP)?

An ODPP Service Provider (SP) is an entity that provides a specific set of data masking services associated with a type or category of masking or data privacy.

5.2 Service Provider – Age (AGE)

This SP is used to age data values in source columns. Aging is a process of incrementing or decrementing a date value. Aging can be specific to the number of years, months, weeks or days. Optionally, aging may be a combination of these units. Aging can also be based upon a specific 4-digit year value.

The Age SP does not handle BC-type dates and will:
1. Generate an error if a BC date (e.g. a date with a negative year value) is encountered in the input data.
2. Generate an error if an input date is aged to a BC-type date (e.g. a date with a negative year value)

5.2.1 Supported data types

The AGE SP supports the following data types for source and destination fields:

- ODPPDATATYPE_ODBC_DATE
- ODPPDATATYPE_ODBC_TIMESTAMP
- ODPPDATATYPE_DATETIME_SZ
- ODPPDATATYPE_DATETIME_WSZ
- ODPPDATATYPE_DATETIME_CHAR
- ODPPDATATYPE_DATETIME_WCHAR
- ODPPDATATYPE_DATETIME_VARCHAR
- ODPPDATATYPE_DATETIME_WVARCHAR

5.3 Service Provider – Affinity / Column Transformation (COL)

This SP is used to mask data of undifferentiated or dynamically-formatted values.

An undifferentiated value is where there are no parts that have significance therefore, all parts of the value are candidates for masking.
(e.g. 123456, Gizmo, CDE9874)

A dynamically-formatted value has one or more portions that have significance and cannot be altered without affecting the validity of the value.
(e.g. 12-3456789, ItemCode Gizmo, CDE-9874)

For unique inputs, a unique, repeatable output is generated. For non-unique inputs, it is possible to generate a different output for each occurrence of the non-unique input. It is also possible to generate output values with a length longer than the input values.

In Column Transformation, the format of the input value controls the output format; that is, the output character type matches the input character type. For example, input Ab-12 might produce output Xr-86; that is, the upper and lower-case alpha, numeric and special characters in the output follow the template defined by the input

Options are available that allow the specification of which portions of the value must remain unchanged/unmasked.

5.3.1 Supported data types

The Affinity service provider supports the following data types for source and destination fields where ODPP_OPR_COL_BINARY is NOT specified:

- ODPPDATATYPE_CHAR
- ODPPDATATYPE_WCHAR
- ODPPDATATYPE_VARCHAR_SZ
- ODPPDATATYPE_WVARCHAR_SZ
- ODPPDATATYPE_VARCHAR
- ODPPDATATYPE_WVARCHAR
- ODPPDATATYPE_DOUBLE
- ODPPDATATYPE_FLOAT
- ODPPDATATYPE_LONG_LONG
- ODPPDATATYPE_U_LONG_LONG
- ODPPDATATYPE_INTEGER
- ODPPDATATYPE_U_INTEGER
- ODPPDATATYPE_SMALLINT
- ODPPDATATYPE_USMALLINT
- ODPPDATATYPE_TINYINT
- ODPPDATATYPE_UTINYINT
- ODPPDATATYPE_DECIMAL_370

The Affinity service provider supports the following data types for source and destination fields where ODPP_OPR_COL_BINARY is specified:

- ODPPDATATYPE_U_LONG_LONG
- ODPPDATATYPE_LONG_LONG
- ODPPDATATYPE_DOUBLE
- ODPPDATATYPE_FLOAT
- ODPPDATATYPE_U_INTEGER
- ODPPDATATYPE_INTEGER
- ODPPDATATYPE_USMALLINT
- ODPPDATATYPE_SMALLINT
- ODPPDATATYPE_UTINYINT
- ODPPDATATYPE_TINYINT

5.4 Service Provider - Credit Card Number Masking (CCN)

This Service Provider generates a valid and unique credit card number (CCN). By default, the Credit Card Service Provider algorithmically generates a consistently altered CCN based on the source CCN. It also generates a random value when the source data does not have a CCN value or when there is no need for transforming the source CCN in a consistent manner.

A CCN, as defined by ISO 7812, consists of a 6-digit issuer identifier followed by a variable length account number and a single check digit as the final number. The check digit verifies the accuracy of the CCN and is generated by passing the issuer identifier and account numbers through the Luhn algorithm. The maximum length of a CCN currently supported in ODPP is 16 digits.

The CCN SP supports two methods of masking:
- mask
- random

5.4.1 Mask method

The mask method generates a CCN by including the first 4-digits of the issuer identifier from the source CCN and altering the remaining 2-digits of the issuer identifier number and the account number based on the source CCN. A valid check digit is also assigned.

When using the mask method, the following flags may be specified with the ODPP_OPR_CCN_FLAGS parameter to modify the mask action:

5.4.1.1 ODPP_FLAG_CCN_ISSUER6

When using ODPP_FLAG_CCN_ISSUER6, the destination CCN includes the first 6-digits of the issuer identifier from the source CCN. The masked account number is altered based upon the source CCN. The last digit is a check digit.

5.4.1.2 ODPP_FLAG_CCN_RANDOM

When using ODPP_FLAG_CCN_RANDOM, the destination CCN includes the first 4-digits of the issuer identifier from the source CCN. A sequential account number is assigned, where the first account number begins, starting with 1, and for each additional CCN that uses the same issuer identifiers, the account number will be incremented by 1. The last digit is a check digit.

5.4.2 Random method

The random method generates a CCN that has an issuer identifier number that is assigned to American Express, Discover, MasterCard, VISA, JCB, enRoute or Diners Club. A sequential account number is assigned where the first account number begins, starting with 1, and for each additional CCN that uses the same issue identifier, the account number will be incremented by 1. The last digit is a check digit.

5.4.3 Validations

The following basic validations are performed as part of the CCN service provider:
1. The input credit card number is non numeric value.
2. The input credit card number is invalid in length.
3. The input check digit is invalid for credit card number

5.4.4 Supported data types

The CCN SP supports the following data types:

- ODPPDATATYPE_CHAR
- ODPPDATATYPE_WCHAR
- ODPPDATATYPE_VARCHAR
- ODPPDATATYPE_WVARCHAR
- ODPPDATATYPE_VARCHAR_SZ
- ODPPDATATYPE_WVARCHAR_SZ
- ODPPDATATYPE_ORA_VARNUM
- ODPPDATATYPE_DECIMAL_370
- ODPPDATATYPE_U_LONG_LONG

5.5 Service Provider - Email Address Masking (EML)

This Service Provider generates an email address. An email address consists of two parts, a user name followed by a domain name, separated by '@'. For example, user@domain.com.

The Email Service Provider generates an email address with a user name based on either destination data or a literal concatenated with a sequential number. The domain name can be based on an email address in the source data, a literal, or randomly selected from a list of large email service providers. The email address can also be converted to upper or lower case.

The Email Service Provider can generate a user name based on the values in one or two columns (usually containing the name of a user). Processing options allow you to use only the first

character of the value in the first column (for example, the initial letter of a first name) and separate the values from both columns using either a period or an underscore.

If the user name is based on a single column value or a literal, the name will be concatenated with a sequential number. If a user name is based on values in two columns and a separating period or underscore is not used, the values are concatenated. If a parameter is not provided for the user name, the name will be formed by the literal "email" concatenated with a sequential number. Sequential numbers for user names are suffixes that begin with 1 and are incremented by 1.

Using the Hash option (e.g. ODPP_EML_FLAG_HASH_ENABLED) the Email SP can be used to generate repeatable email addresses.

5.5.1 Validations

The following basic validations are performed as a part of the EML service provider:
1. The input email does not contain '@' symbol.
2. The input email domain length exceeds the maximum length.
3. The conversion of source is not possible due to invalid source value.

5.5.2 Supported data types

The EML SP supports the following data types:
- ODPPDATATYPE_CHAR
- ODPPDATATYPE_WCHAR
- ODPPDATATYPE_VARCHAR
- ODPPDATATYPE_WVARCHAR
- ODPPDATATYPE_VARCHAR_SZ
- ODPPDATATYPE_WVARCHAR_SZ

5.6 Service Provider – Hash

The Service Provider is used is to return a numeric Hash value based on an input source value. The source value can be in various forms like character string, integers, floats, date/time etc. Hash Service Provider also supports multiple source values, of the same or different data type. When multiple source values are used they are first converted to a UTF-8 string and then hashed.

The output hash value may not be unique even if the input is unique but is repeatable for a given input. (i.e. the same hash value is generated for the same input). To generate repeatable hash values the seed value must be constant for a given input. The Hash value for the same input will vary if the seed is changed for that input.

The caller into the Hash SP also has the capability to:
1. Trim characters from the source
2. Add a separator while concatenating multiple source values
3. Specify the format for input date and time values

5.6.1 Supported data types

The Hash SP supports the following data types for the source column:

- ODPPDATATYPE_CHAR
- ODPPDATATYPE_WCHAR
- ODPPDATATYPE_VARCHAR
- ODPPDATATYPE_WVARCHAR
- ODPPDATATYPE_VARCHAR_SZ
- ODPPDATATYPE_WVARCHAR_SZ
- ODPPDATATYPE_ODBC_DATE
- ODPPDATATYPE_ODBC_TIME
- ODPPDATATYPE_ODBC_TIMESTAMP

- ODPPDATATYPE_DATETIME_SZ
- ODPPDATATYPE_DATETIME_WSZ
- ODPPDATATYPE_DOUBLE
- ODPPDATATYPE_FLOAT
- ODPPDATATYPE_LONG_LONG
- ODPPDATATYPE_U_LONG_LONG
- ODPPDATATYPE_INTEGER
- ODPPDATATYPE_U_INTEGER
- ODPPDATATYPE_SMALLINT
- ODPPDATATYPE_USMALLINT
- ODPPDATATYPE_TINYINT
- ODPPDATATYPE_UTINYINT
- ODPPDATATYPE_ORA_VARNUM
- ODPPDATATYPE_DECIMAL_370

The Hash SP support the following data types for the destination column if the parameter ODPP_OPR_HASH_MAXVALUE is specified:

- ODPPDATATYPE_CHAR
- ODPPDATATYPE_WCHAR
- ODPPDATATYPE_VARCHAR
- ODPPDATATYPE_WVARCHAR
- ODPPDATATYPE_VARCHAR_SZ
- ODPPDATATYPE_WVARCHAR_SZ
- ODPPDATATYPE_DOUBLE
- ODPPDATATYPE_FLOAT
- ODPPDATATYPE_LONG_LONG
- ODPPDATATYPE_INTEGER
- ODPPDATATYPE_SMALLINT
- ODPPDATATYPE_TINYINT
- ODPPDATATYPE_ORA_VARNUM
- ODPPDATATYPE_DECIMAL_370

The SP supports the following data types for the destination column if the parameter ODPP_OPR_HASH_MAXVALUE is NOT specified:

- ODPPDATATYPE_INTEGER
- ODPPDATATYPE_SMALLINT
- ODPPDATATYPE_TINYINT

5.7 Service Provider – Lookup (LKP)

The primary purpose of the Lookup Service Provider is to “lookup” data, using a key from the source data, and then subsequently replacing the source data with the replacement data.

Certain type of generic data like names, addresses etc., can not be generated using arithmetical logic as that found with many of the other ODPP service providers. When this generic-type of data needs to be masked, a similar set of replacement data is required. The process of looking up this replacement data using a key from source data is called “lookup”

The primary requirement for doing lookup is to have a set of replacement data which is similar in type to the original data. As a rule, a name is replaced by a name, address is replaced by a address and so on. Replacement data is normally stored as a set of rows in a database with a key column. Value(s) in the key column(s) of original data or a Hash value generated from original data columns are used to lookup in the key column of replacement table to fetch a replacement record. Typically the replacement records are selected that they are unrelated to the original records but contextually similar.

Lookup is performed via the key column(s) of a replacement type-table and is based upon:

- Values in the key columns of the original input data
- or -
- Hash-type value generated from the original input data column(s)

Lookup Service Provider provides lookup functionality with three types of key lookups:
- Lookup
- Hash Lookup
- Random Lookup

5.7.1 Lookup

In Lookup, key source field(s) are used to find matching records in lookup data. There need to be one to one mapping between the source key data and replacement key data.

There are two forms of Lookup:

5.7.1.1 Single column

The single column search form uses a single key source column to lookup data in replacement table and can replace a single column or multiple columns with replacement data.

5.7.1.2 Multiple column

The multiple column search form uses multiple key source columns to lookup data in replacement table and can replace a single column or multiple columns with replacement data. All the columns can be joined together by using logical operators AND or OR.

The following logical operations will be possible between all the columns:

Logical AND
This will compare the source values with the values in the replacement table search columns and if all the values match then replacement record is returned.
Logical OR
This will compare the source values with the values in the replacement table search columns and if any of the values match then replacement record is returned.

5.7.2 Hash Lookup

In Hash Lookup, a hash value is generated from a single or multiple source columns which is then used as a key value to lookup in a special sequence column in a replacement table. The Hash Lookup is case-sensitive. For example, the source values John and JOHN will be hashed to different values. You can use the ODPP_OPR_LOOKUP_FLAGS parameter with ODPP_FLAG_LOOKUP_SRC_UPR value to convert the source value to uppercase before it is hashed.

There are two forms of the Hash Lookup:

5.7.2.1 Single Hash Column

The single column hash form hashes a single source column to generate a hash value to lookup data in replacement table and can replace a single column or multiple columns with replacement data.

5.7.2.2 Multiple Hash Columns

The multiple column hash form hashes multiple source columns to generate a hash value to lookup data in replacement table and can replace a single column or multiple columns with replacement data.

5.7.3 Random Lookup

The Random Lookup selects a value at random from a specified replacement table to use as a replacement record. The function generates a random number between 1 and the limit supplied using ODPP_OPR_LOOKUP_LIMIT parameter or number of rows in the replacement table to use as a subscript into the replacement table. The column value or values from the row that correspond to the subscript are used as replacement values.

There are two forms of the Random Lookup:

5.7.3.1 Single column

Replaces a single column with replacement data.

5.7.3.2 Multiple column

Replaces multiple columns with replacement data.

5.7.4 Supported data types

The Lookup Service Provider supports the following data types for the source and destination column:

- ODPPDATATYPE_CHAR
- ODPPDATATYPE_VARCHAR
- ODPPDATATYPE_VARCHAR_SZ
- ODPPDATATYPE_WCHAR
- ODPPDATATYPE_WVARCHAR
- ODPPDATATYPE_WVARCHAR_SZ
- ODPPDATATYPE_LONG_LONG
- ODPPDATATYPE_U_LONG_LONG
- ODPPDATATYPE_INTEGER
- ODPPDATATYPE_U_INTEGER
- ODPPDATATYPE_SMALLINT
- ODPPDATATYPE_USMALLINT
- ODPPDATATYPE_TINYINT
- ODPPDATATYPE_UTINYINT
- ODPPDATATYPE_DOUBLE
- ODPPDATATYPE_FLOAT
- ODPPDATATYPE_ODBC_DATE
- ODPPDATATYPE_ODBC_TIME
- ODPPDATATYPE_ODBC_TIMESTAMP
- ODPPDATATYPE_DATETIME_CHAR
- ODPPDATATYPE_DATETIME_VARCHAR
- ODPPDATATYPE_DATETIME_WCHAR
- ODPPDATATYPE_DATETIME_WVARCHAR
- ODPPDATATYPE_DATETIME_SZ
- ODPPDATATYPE_DATETIME_WSZ

5.8 Service Provider - National Identifier Masking (NID)

The NID Service Provider is used to generate valid and unique National Identifiers (NIDs) for:

U.S. = Social Security Number (SSN)
U.K. = National Insurance Number (NINO)
Canada = Social Insurance Number (SIN)
France = Institute for Statistics and Economic Studies (INSEE)
Italy = Fiscal Code (CF)
Spain = Fiscal Identification (NIF) / Foreign Identification Number (NIE)

The NID Service Provider supports two different methods of masking:
- mask

- random

5.8.1.1 Mask method

This method uses an algorithm-based generation to create the destination NID based upon the source NID.

5.8.1.2 Random method

This method randomly generates the destination NID when the source does not have a NID value or where there is no need to transform the source NID in a consistent manner.

5.8.2 U.S. NID – (SSN)

When the NID Service Provider is configured for the U.S., it is used to generate a valid and unique U.S. Social Security Number (SSN). By default, the United States National ID Service Provider algorithmically generates a consistently altered destination SSN based on the source SSN. This Service Provider can also generate a random SSN when the source data does not have an SSN value or when there is no need for transforming the source SSN in a consistent manner.

An SSN is made-up of three subfields. The first three digits (area) represent an area generally determined by the state in which the SSN is issued. The next two digits (group) define a group number corresponding to the area number. The last four digits (serial) are a sequential serial number. Regardless of the type of processing, mask or random, this service provider will generate an SSN with a group number appropriate to the area number.

The mask method generates an SSN that includes the source area number as well as altered group and serial numbers based on the source SSN.

The random method generates an SSN that has a random area number and an appropriate group number and serial number. The group number assigned will be the most recent group used by the Social Security Administration for the area. Serial numbers begin with 0001 and are incremented by 1 for each additional SSN generated for the area number. When the serial number exceeds 9999, the serial number will be reset to 0001 and the group number preceding the number most recently issued for the area number will be used.

5.8.2.1 Format

A U.S. SSN is a 9 digit number having the following format:
RRR-GG-SSSS

RRR	Area number
GG	Group number
SSSS	Sequential serial number

The three subfields can be separated as shown below:
123-45-6789
123 45 6789
123.45.6789

5.8.2.2 Validations

The following validations are part of the basic validations for the United States National ID:
1. The input national ID is invalid in length.
2. The conversion of source is not possible due to invalid source value.
3. The input national ID has invalid separator.
4. The input national ID is invalid.
5. The input area number exceeds the maximum value.
6. The input area has not been used by the Social Security Administration.

The below validations are done only when parameter ODPP_OPR_FLAGS is specified with value ODPP_FLAG_VALIDATE_INPUT during Provider_Init():
1. The input SSN group number has not been used in the specified area.
2. The input serial number is 0 or the SSN is a reserved value
   (i.e. 078-05-1120 and 457-55-5462)

5.8.2.3 Supported data types

The United States NID SP supports the following data types for the source and destination columns:
- ODPPDATATYPE_CHAR
- ODPPDATATYPE_WCHAR
- ODPPDATATYPE_VARCHAR
- ODPPDATATYPE_WVARCHAR
- ODPPDATATYPE_VARCHAR_SZ
- ODPPDATATYPE_WVARCHAR_SZ
- ODPPDATATYPE_U_LONG_LONG
- ODPPDATATYPE_U_INTEGER
- ODPPDATATYPE_ORA_VARNUM
- ODPPDATATYPE_DECIMAL_370

5.8.3 U.K. NID - (NINO)

When the NID Service Provider is configured for the U.K., it is used to generate a valid and unique United Kingdom National Insurance Number (NINO). By default, the United Kingdom National ID Service Provider algorithmically generates a consistently altered destination NINO based on the source NINO. This service provider can also generate a random NINO when the source data does not have an insurance number or when there is no need for transforming the source NINO in a consistent manner.

5.8.3.1 Format

A National Insurance Number (NINO) is of the form *AB123456C:*
- 2-letter prefix
- 6-digit serial number
- optional suffix.

A NINO is often separated into parts by dashes, dots or spaces. Possible sample NINO formats are shown below:
AB-12-34-56-C
AB-12-34-56
AB-123456-C
AB-123456
AB123456C
AB123456

5.8.3.2 Validations

The following validations are part of the basic validations for the United Kingdom National ID:
1. The input national ID is invalid in length.
2. The conversion of source is not possible due to invalid source value.
3. The input national ID has invalid separator.
4. The input national ID is invalid.
5. The invalid prefix in national ID.
6. The invalid suffix in national ID.
7. The invalid number in national ID.

5.8.3.3 Supported data types

The U.K. NID SP supports the following data types for the source and destination columns:
- ODPPDATATYPE_CHAR
- ODPPDATATYPE_WCHAR
- ODPPDATATYPE_VARCHAR
- ODPPDATATYPE_WVARCHAR
- ODPPDATATYPE_VARCHAR_SZ
- ODPPDATATYPE_WVARCHAR_SZ

5.8.4 Canada NID - (SIN)

When the NID Service Provider is configured for Canada, it is used to generate a valid and unique Canadian Social Insurance Number (SIN). By default, the Canada National ID Service Provider algorithmically generates a consistently altered destination SIN that includes the first three digits (header) of the source SIN. This Service Provider can also generate a random SIN when the source data does not have an Insurance Number or when there is no need for transforming the source SIN in a consistent manner.

5.8.4.1 Format

The Canadian Social Insurance Number is a 9 digit number having the following format:
RSSSSSSSS

R	Region code
SSSSSSSS	Serial number

A Social Insurance Number is separated into three subfields, each of three digits as shown below:
123-456-789
123 456 789
123.456.789

5.8.4.2 Validations

The following validations are a part of the basic validations for the Canada National ID:
1. The input national ID is invalid in length.
2. The conversion of source is not possible due to invalid source value.
3. The input national ID has invalid separator.
4. The input national ID is invalid.
5. The invalid region in national ID.

The following validations are performed only when parameter ODPP_OPR_FLAGS is specified with value ODPP_FLAG_VALIDATE_INPUT during Provider_Init():
1. The first digit must not be 8.
2. There must not be three consecutive zeros ("000") at positions 1-3, 4-6 or 7-9.
3. Additional validation is done using the Luhn algorithm.

5.8.4.3 Supported data types

The Canada National ID Service Provider supports the following data types for the source and destination columns:

- ODPPDATATYPE_CHAR
- ODPPDATATYPE_WCHAR
- ODPPDATATYPE_VARCHAR
- ODPPDATATYPE_WVARCHAR
- ODPPDATATYPE_VARCHAR_SZ
- ODPPDATATYPE_WVARCHAR_SZ
- ODPPDATATYPE_U_LONG_LONG
- ODPPDATATYPE_U_INTEGER
- ODPPDATATYPE_ORA_VARNUM
- ODPPDATATYPE_DECIMAL_370

5.8.5 France NID – (INSEE)

When the NID Service Provider is configured for France, it is used to generate a valid and unique French National Institute for Statistics and Economic Studies (INSEE) number. By default, the France National ID Service Provider algorithmically generates a consistently altered destination INSEE number based on the source INSEE number. This Service Provider can also generate a random INSEE number when the source data does not have an INSEE number or when there is no need for transforming the source INSEE number in a consistent manner.

5.8.5.1 Format

The France National ID is a 15 digit number having the following format:
SYYMMDDCCCOOOKK

S	Sex and Citizenship information
YY	Last two digits of the Year of Birth
MM	Month of Birth (January has a value 1)
DD	Department of origin
CCC	Commune of origin (Valid values are defined by the department value)
OOO	Order number
KK	Control key or Check Digit

5.8.5.2 Validations

The following validations are part of the basic validations for the France National ID:

1. The input national ID is invalid in length.
2. The conversion of source is not possible due to invalid source value.
3. The input national ID has invalid separator.
4. The input national ID is invalid.
5. The input national ID has invalid name.
6. The replacement character is invalid for national ID.
7. The national ID has invalid character.
8. The family name is invalid for national ID.
9. The first name is invalid for national ID.
10. The birth date is invalid for national ID.
11. The region code is invalid for national ID.

The below validations are performed only when parameter ODPP_OPR_FLAGS is specified with value ODPP_FLAG_VALIDATE_INPUT during Provider_Init():

1. Check for an invalid commune code.
2. Check for an invalid control key (check digit).

5.8.5.3 Supported data types

The France National ID Service Provider supports the following data types for the source and destination columns:

- ODPPDATATYPE_CHAR
- ODPPDATATYPE_WCHAR
- ODPPDATATYPE_VARCHAR
- ODPPDATATYPE_WVARCHAR
- ODPPDATATYPE_VARCHAR_SZ
- ODPPDATATYPE_WVARCHAR_SZ

5.8.6 Italy NID – (CF)

When the NID Service Provider is configured for Italy, it generates a valid and unique Italian Fiscal Code number (CF). By default, the Italy National ID Service Provider algorithmically generates a consistently altered destination CF number based on the source CF number. This Service Provider can also generate a random CF number when the source data does not have a Fiscal Code number or when there is no need for transforming the source CF number in a consistent manner.

5.8.6.1 Format

The Italian Fiscal Code number is a 16 digit number having the following format:
FFF-NNN-YYMDD-RRRRC

FFF	Encoded Family Name
NNN	Encoded First Name
YY	Year of Birth
M	Month of Birth
DD	Day of Birth
RRRR	Region Code
C	Control Character

5.8.6.2 Validations

The following validations are part of the basic validations for the Italy National ID:

1. The input national ID is invalid in length.
2. The conversion of source is not possible due to invalid source value.
3. The input national ID has invalid separator.
4. The input national ID is invalid.
5. The input national ID has invalid name.
6. The replacement character is invalid for national ID.
7. The national ID has invalid character.
8. The family name is invalid for national ID.
9. The first name is invalid for national ID.
10. The birth date is invalid for national ID.
11. The region code is invalid for national ID.

The following validations are done only when parameter ODPP_OPR_FLAGS is specified with value ODPP_FLAG_VALIDATE_INPUT during Provider_Init():

1. Check for an invalid control character (check digit).

5.8.6.3 Supported data types

The Italy National ID Service Provider supports the following data types for the source and destination columns:

- ODPPDATATYPE_CHAR
- ODPPDATATYPE_WCHAR
- ODPPDATATYPE_VARCHAR

- ODPPDATATYPE_WVARCHAR
- ODPPDATATYPE_VARCHAR_SZ
- ODPPDATATYPE_WVARCHAR_SZ

5.8.7 Spain NID – (NIF/NIE)

When the NID Service Provider is configured for Spain, it is used to generate a valid and unique Spanish Fiscal Identification Number (NIF) / Foreign Identification Number (NIE). By default, the Spain National ID Service Provider algorithmically generates a consistently altered destination NIF/NIE number based on the source NIF/NIE number. This Service Provider can also generate a random NIF/NIE number when the source data does not have an Identification Number or when there is no need for transforming the source NIF/NIE number in a consistent manner.

5.8.7.1 Format

A Fiscal Identification Number (NIF) is of the form *0000000-A* where there is a seven digit serial and a suffix (A). The serial may contain numbers ranging from 0000000 to 9999999.

A Foreign Identification Number (NIE) is of the form *X-0000000-A* where X is a literal, followed by a seven digit serial and a suffix (A). Prefix X is ignored in the masking process (i.e., an NIE of the form X-0000000-Y retains the form X-1234567-L after masking).

5.8.7.2 Validations

The following validations are part of the basic validations for the Spain National ID:
1. The input national ID is invalid in length.
2. The conversion of source is not possible due to invalid source value.
3. The input national ID has invalid separator.
4. The input national ID is invalid.

The following validations are done only when parameter ODPP_OPR_FLAGS is specified with value ODPP_FLAG_VALIDATE_INPUT during Provider_Init():
1. Check for an invalid suffix.

5.8.7.3 Supported data types

The Spain National ID Service Provider supports the following data types for the source and destination columns:
- ODPPDATATYPE_CHAR
- ODPPDATATYPE_WCHAR
- ODPPDATATYPE_VARCHAR
- ODPPDATATYPE_WVARCHAR
- ODPPDATATYPE_VARCHAR_SZ
- ODPPDATATYPE_WVARCHAR_SZ

5.9 Data Swapping Service Providers - Introduction

There is an increasing exchange of large data sets within and between organizations. However, this data must often be protected such that it can be disclosed or published without revealing confidential information, be it business secrets or the capability to link sensitive information to individuals whose data is contained therein. The primary purpose of the Data Swapping Service Provider is to “swap” data within the selected column of the source data. Data swapping is a statistical disclosure limitation technique that works at the microdata level. Confidentiality protection is achieved by modifying a fraction of the records in the offered set of data by switching a subset of fields across selected pairs of rows (known as swap pairs). The goal is to make it impossible or at least very hard for any intruder to be certain of having identified a person or entity.

Removing attributes that uniquely identify individuals, such as names, employee numbers or social insurance numbers, is often not sufficient to protect the identity of individuals. Even if these direct identifiers are removed from the data set, often a unique combination of attribute values is sufficient to identify a person, at least with a high probability. Therefore, these key attributes - when considered together - must be determined. This decision depends on (national) laws, business practices, the sensitivity of the data, and the recipients (scientific or public use). With respect to these key variables, the disclosure risk must be determined; i.e., estimating the corresponding re-identification risk for each individual by taking for example the population frequencies into account. Among other techniques, data swapping can then be used to perturb the data with high risks. After the data is perturbed, the information loss and the (new) disclosure risk has to be estimated. The transformed data can be released if it has low risk of linking confidential information to individuals and still has high data utility.

Before release, the transformed data must be validated. Validation includes simple tasks as the examination of the (minimal) cluster size. But in some situations, there may be conditions on swapped fields that are physically infeasible (and hence detectable), such as males who have undergone hysterectomies.

Data swapping preserves the original values of the attribute (and their distribution), while protecting persons’ privacy. It achieves this by changing the correspondence between original values of the attribute and rows in the modified data. The preservation of the original data distribution in the modified data is an important property for a range of applications, in which the produced data can be of high utility. Moreover, by avoiding introducing new values to the modified data (that were non-existent in the original data), data swapping allows providing realistic values of the attribute. Application areas include customer data, census data, and medical data,

The following table identifies the data swapping ODPP service providers detailed within the following sections:

Provider	Description
Data Swapping	Swaps random pairs of fields
Class-based Data Swapping	Swaps random pairs of fields dependent on a control field
Distance-based Data Swapping	Swaps random pairs of fields within a given distance

Whereas the Data Swapping (DS) provider gives a basic protection of most microdata files, the other two providers offer parameters to influence the data swap so that there is some control over the amount of distortion introduced into the resulting file. By construction, these providers offer higher data utility than general data swapping.

5.9.1 Service Provider - Data Swapping (DS)

In plain data swapping, fields within a column are randomly swapped.

5.9.1.1 Multiple columns

In the multiple column form, fields are swapped within each selected column. Over the different columns, swapping can be performed together (locked) or independently (unlocked).

LOCKED = YES

This mode will swap all selected fields in a given row to the corresponding fields of the same new row. For example, if a field in row *i* should be copied into the same field of row *j* then all other selected fields of row *i* will also be copied into the corresponding fields in row *j*.

LOCKED = NO
This mode will swap fields independently of each other; i.e., fields of row *i* may be copied into rows *j*, *k*, etc. Note that swapping the selected fields, each in a separate run, is equivalent to performing the swapping in Unlocked mode.

In general, data swapping with an independent order on the fields achieves a higher protection as the swaps of columns differ; i.e., are independent of each other. Contrary, a locked swap maintains the correlation between the selected fields.

5.9.1.2 Random

If METHOD=REPEATABLE then the DS provider uses the value defined in parameter SEED as the starting point for computing the swap order. In this case, the order is predictable and could be used for testing purposes for example. If METHOD=RANDOM then the DS provider uses a random value computed by itself taking information from the context into account, the time at execution for example. A random seed provides a better protection of the individuals than using a predefined value.

5.9.1.3 Swap Rate

In some cases it might be sufficient to swap fields within a subset of the rows. The Swap Rate defines the amount of rows whose fields should be swapped, measured as a percentage. The idea behind swap rate is that the attacker has no means to distinguish between a row with swapped fields and a row that was left unaffected by the swap process.

5.9.2 Service Provider - Class-based Data Swapping (CDS)

Class-based Data Swapping allows you to increase the utility of the transformed data. The service provider randomly swaps pairs of fields in selected rows but only if a condition on the pairs holds. This condition (or constraint) is defined with respect to a particular field, called the control field.

Class-based data swapping offers a good balance between data protection and data utility. Together with distance-based swapping, this technique is widely adopted for the protection of census tables in microdata releases, and for the protection of the respondents' identity in released survey data. Class-based data swapping should be applied for protecting data where there is high correlation between the sensitive field (swap attribute) and another field in the data (control attribute) and this correlation should be maintained in the released data to facilitate meaningful analysis. As an example, consider a dataset that holds information about individual households, total family income in the household, and the residential area (zip code) where each household is located. Assume that the released data should be used to compute the average income per household in each residential area. To enable meaningful analysis, the sensitive field (household income) needs to be protected while maintaining the correlation of this field with the control field (residential area). Class-based data swapping is the preferred method for achieving this. With respect to the privacy offering, class-based swapping should be used in datasets where the frequency of occurrence of each distinct value of the control attribute is lower-bounded by *k*, thereby providing a privacy level of 1/*k* for the corresponding individuals/rows. Typically, *k* should be at least equal to 3.

Note that class-based data swapping is independent of the order in which the rows appear in the input row set.

5.9.2.1 Multiple column

Replaces multiple columns with replacement data (see Section 5.9.1.1).

5.9.2.2 Random

Generates a predictable output row set (see Section 5.9.1.2).

5.9.2.3 Classifier

A simple way to specify an equivalence class is to use the identity relation on a control field. It does not require user-input to define the groups. The Class-based Data Swapping provider supports IDENTIY classification on categorical data. The IDENTITY classifier checks for identity; i.e., all rows that have the same value in the control field constitute a category. Taking the City field as the base for classification, the identity classifier groups together for example all rows with value Zurich in that field - sensitive attributes are only swapped between rows with Zurich in the control field.

Classification depends on the kind of data in the given row set. Besides having specialized classification functions, there is also the possibility to pre-compute the classification and to provide the result in an additional field (column). In case every category has a unique name (value), the Identity classifier can be used to perform the corresponding data swapping. For example, the string "21-30" could be used to mark all rows that have a value between 21 and 30 in the Age field. Then selected fields would only be swapped within these marked rows.

5.9.3 Service Provider - Distance-based Data Swapping (DDS)

Distance-based data swapping offers the highest level of data utility among the data swapping providers implemented in ODPP. Distance-based swapping operates by swapping random pairs of fields within a given distance, thereby exchanging original data values in rows with swapped values that are close enough to the original. Due to its high utility offering, this technique is widely employed for the protection of census microdata.

The Distance-based Data Swapping (DDS) provider minimizes the distance between pairs of swapped rows based on a given control field. In case the values of the control field are in a particular order, distance-based data swapping offers a less computation-intensive procedure to achieve similar masking behavior than class-based data swapping.

5.9.3.1 Multiple column

Replaces multiple columns with replacement data (see Section 5.9.1.1).

5.9.3.2 Random

Generates a predictable output row set (see Section 5.9.1.2).

5.9.3.3 Block size

The block size divides the data set into a number of blocks. If the size of the data set is not a multiple of the block size, the last block will be smaller. To protect confidential information, it should have at least three rows.

For example assuming that the given data set is ordered according to the Age field, distance-based data swapping assures that values from the selected fields will be swapped to rows being similar in age.

5.9.4 Data Swapping Service Providers - Supported data types

The Data Swapping Service Providers DS, CDS and DDS support the following data types for the source and destination column:

- ODPPDATATYPE_CHAR
- ODPPDATATYPE_VARCHAR
- ODPPDATATYPE_VARCHAR_SZ
- ODPPDATATYPE_WCHAR
- ODPPDATATYPE_WVARCHAR
- ODPPDATATYPE_WVARCHAR_SZ
- ODPPDATATYPE_LONG_LONG
- ODPPDATATYPE_U_LONG_LONG
- ODPPDATATYPE_INTEGER
- ODPPDATATYPE_U_INTEGER
- ODPPDATATYPE_DECIMAL_370
- ODPPDATATYPE_ODBC_DATE
- ODPPDATATYPE_ODBC_TIME
- ODPPDATATYPE_ODBC_TIMESTAMP
- ODPPDATATYPE_DATETIME_SZ
- ODPPDATATYPE_DATETIME_WSZ
- ODPPDATATYPE_DOUBLE
- ODPPDATATYPE_FLOAT
- ODPPDATATYPE_SMALLINT
- ODPPDATATYPE_USMALLINT
- ODPPDATATYPE_TINYINT
- ODPPDATATYPE_UTINYINT
- ODPPDATATYPE_DATETIME_CHAR
- ODPPDATATYPE_DATETIME_VARCHAR
- ODPPDATATYPE_DATETIME_WCHAR
- ODPPDATATYPE_DATETIME_WVARCHAR
- ODPPDATATYPE_ORA_VARNUM
- ODPPDATATYPE_GRAPHIC
- ODPPDATATYPE_VARGRAPHIC

For CDS, the IDENTITY classifier supports string types:

- ODPPDATATYPE_VARCHAR_SZ
- ODPPDATATYPE_WVARCHAR_SZ

6. ODPP Loader

ODPP uses a number of shared libraries and uses the system-specific library search order to load these shared libraries. It is dependent on the PATH / LD_LIBRARY_PATH / LIBPATH / SHLIB_PATH environment variable to load its libraries.

This feature will enable ODPP to load its libraries without having a dependency on system environment variables such as PATH / LD_LIBRARY_PATH / LIBPATH / SHLIB_PATH. ODPP Loader must be invoked as the first activity even before ODPP framework initialization is done.

This is an optional feature controlled by the user by means of a Loader API independent of the ODPP core API's and to use this feature the application must load and invoke all ODPP API's dynamically.

6.1 Interfaces

The primary interface is a Loader API exported by an ODPP shared library called "ODPPLoader" which is a stand-alone shared library that does not depend upon any other external library except system libraries.

6.1.1 The Loader API

Provider_PreLoad(char *pBinPath**,**
int iBinPathBytes,
DP_LIB_LIST *pLibList,
int iArrCnt**)**

This API will dynamically load all the ODPP libraries, **EXCEPT the SERVICE PROVIDER libraries and their dependent libraries**, using the path supplied in pBinPath.

Optionally if any additional libraries are specified using argument 'pLibList' then that too will be loaded before loading the ODPP libraries.

Argument	Data Type		Description
pBinPath	char *	IN	Pointer to a fully qualified directory path containing ODPP binaries.
iBinPathBytes	int	IN	Size of the buffer, pointed to by pBinPath, in bytes.
pLibList	DP_LIB_LIST *	IN	**[OPTIONAL]** An array of additional libraries to load.
iArrCnt	int	IN	Count of elements in array pLibList

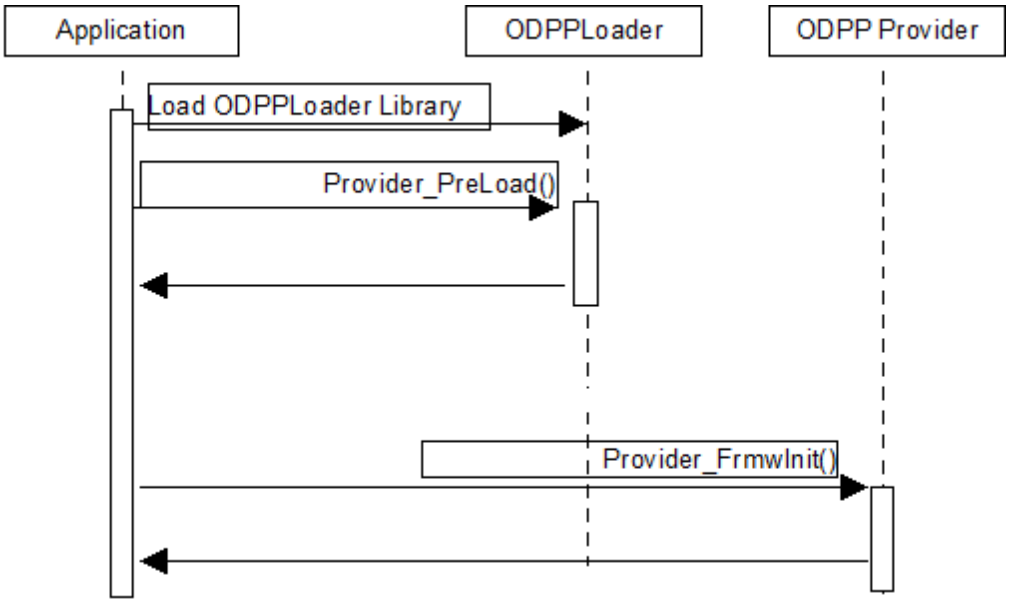
6.1.1.1 Argument 'pLibList'

This optional argument must be set to NULL unless there are other shared libraries to load in addition to ODPP libraries.

For example:
If you write a custom service provider and implement it as multiple shared libraries, you can then feed the custom service provider dependencies to the loader, using this argument, and pre-load the libraries before initializing the service provider.

6.2 Sequence Diagram

The following sequence diagram shows how an application would use the ODPP Loader along with ODPP core without setting the LD_LIBRARY_PATH or LIBPATH or SHLIB_PATH in a Unix environment.



6.3 Platform Support

This feature is supported on all ODPP supported platforms except HP PA-RISC

7. Samples

The Samples folder within the ODPP-type .zip contains ODPP-type sample programs for Windows, UNIX, Linux and z/OS.

7.1 ODPP stand-alone using CCN

The Samples\App_CCN folder within the ODPP-type .zip contains a stand-alone ODPP program that uses the ODPP Credit Card Number (CCN) service provider. Consult the file "Using this sample.txt" for complete details on building and running this stand-alone program.

7.2 The Optim solution column map exit using ODPP CCN

The Samples\CMExit_ODPP_CCN folder within the ODPP-type .zip contains an Optim solution Column Map Exit (CME) that uses the ODPP Credit Card Number (CCN) service provider. Consult the Optim_CMExit_ODPP_CCN.doc file for complete details on building and running this stand-alone program.

7.3 ODPP Affinity custom language exit

The Samples\ODPP_AFFLANGEXIT folder within the ODPP-type .zip contains a sample custom language exit for the Affinity Service Provider.

7.4 ODPP Hash HMAC exit

The Samples\ODPP_HASHEXIT folder within the ODPP-type .zip contains a sample exit for the Hash Service Provider.

7.5 Optim/z column map exits

The Samples\zOS_CMEXits folder within the ODPP-type .zip contains two Optim/z-type Column Map Exits.

The first one, is written in C and uses the ODPP Credit Card Number service provider. Consult the Optim_CMExit_ODPP_CCN_zOS.doc for complete details for building and running this Optim/z CME.

The second one, is written in COBOL and uses the ODPP National Identifier service provider. Consult the Optim_CMExit_ODPP_NID_zOS.doc for complete details for building and running this Optim/z CME.

7.6 ODPP Java Sample

The Samples\JavaAPI folder within the ODPP-type .zip contains a sample Java program that demonstrates the Java API. The folder also includes a Readme file with information about the API.

8. Limitations

The following are limitations within the current ODPP implementation:

8.1 User-Defined Functions (UDFs)

For ODPP v11.3.0.4, UDFs are supported for the following databases:
- DB2 z/OS
- DB2 LUW
- MS SQL Server
- Netezza
- Oracle
- Teradata

For ODPP v11.3.0.4, UDFs are supported for the following ODPP-type service providers:
- Affinity (column transformation)
- Age
- Credit Card Numbers
- Email
- Hash
- National Identifiers

For ODPP v11.3.0.4, UDFs are supported for the following data types:
- char
- date
- decimal/numeric
- double/float
- integer
- nchar
- nvarchar
- time
- timestamp
- varchar

8.1.1 DB2 z/OS

For ODPP-type UDFs for DB2/z, since the UDFs are created by default as non deterministic, any CREATE TABLE AS-type statements that you use that incorporate one or more ODPP-type UDFs will require the DISABLE QUERY OPTIMIZATION clause. For example:

```
CREATE TABLE XYZZY (CUST_ID, CUST_ID_MASKED)
        AS (SELECT    CUST_ID,
                      OPTIMMASK(CUST_ID, 'PROVIDER=AFFINITY,
                              FLDDEF1=(NAME=X,LENGTH=5,DT=CHAR),
                              METHOD=HASH' )
        FROM CUSTOMERS)
        DATA INITIALLY DEFERRED REFRESH DEFERRED
        DISABLE QUERY OPTIMIZATION ;
```

8.1.2 Netezza

8.1.2.1 Random masking using CCN, EML or NID providers may have duplicates.

During execution of a query, Netezza sends the query to all the data slices present, these data slices execute the query in separate parallel processes. At the end of the query, the output from all of the queries on all of the data slices are merged into a single result set.

From the ODPP perspective, the Random method generates the output using a combination of a random sequence and sequence numbers. This means it always starts from the same point for each query in each Netezza slice thus generating the same set of values in each slice. When the query output is merged together, the result is duplicate values with a rate of duplication equal to the number of data slices.

8.1.3 Oracle

When a UDF is run on an Oracle server, it is actually run within a separate process called EXTPROC.EXE on the Oracle server. At the beginning of the execution of an Oracle UDF (i.e. external process) the external program library (i.e. DLL or shared object) is loaded into memory within the EXTPROC.EXE process. When the Oracle SQL query hosting a UDF execution completes, the external program library (i.e. DLL or shared object) is not unloaded from memory. This means subsequent executions of the same UDF do not require the library to be reloaded.

An unfortunate by-product of this action is that if you are using an ODPP-type UDF that uses the Email-type service provider and you have specified a METHOD=REPEATABLE, the subsequent executions of the UDF with the same inputs will not reproduce the same repeatable outputs. This is because the sequence indicator that is maintained within the ODPP is not reset between executions since the ODPP libraries remain in memory.

You may overcome this Oracle-type limitation by simply stopping and restarting the Oracle Listener service as this will cause the EXTPROC.EXE to unload the libraries when the service is stopped and to reload the libraries when the ODPP-type UDF is subsequently run.

Another alternative might be to use the Email-type service provider with the METHOD=HASH. This may result in some duplicates but there would be consistency between different executions.

8.1.4 Teradata

By default, Teradata UDFs currently do not provide persistent storage across invocations. This means that the ODPP-type UDFs for Teradata are initialized and terminated upon invocation for each row. A side affect of this is that the ODPP-type service providers, Credit Card Number (CCN) and National Identifiers (NID) will not generate masked output data that is truly random as the METHOD=RANDOM algorithm is based upon a counter which is consistent with Optim.

Given the Teradata UDF lack of persistent storage, this counter will be reset to 1 each time thus producing the masked output that is not as random as you would expect. For the ODPP-type UDFs that utilize CCN or NID, you may want to specify the METHOD=REPEATABLE parameter.

8.2 Hash_Lookup Service Provider

The HASH_LOOKUP service provider supports three special values. They are: -1 for NULL, -2 for Space and -3 for Zero Length as a part of the sequence values in the replacement table. In addition to these special values, positive values from 1 to 'n' where 'n' is the maximum value are supported. Further, there must be no gaps in the positive sequence values.

If you happen to use negative values other than those detailed above or you have gaps in the positive sequence values, then the results of the HASH_LOOKUP function are unpredictable.

8.3 Lookup Service Provider

The ODPP Lookup service on Linux, UNIX and Windows supports DB2-LUW v9.1 and beyond, as well as Oracle v10.2 & v11.2. The ODPP Lookup service on z/OS supports DB2 z/OS v8.1 and beyond.

8.4 Lookup limitation with double and float data type

For Real and Double data type columns, in a DB2 replacement table, ODPP supports only fields with Float and Double ODPP data types respectively. Using any other data type for such columns might result in the following:

- If used in a search field:
  Failure to search the replacement row

- If used in a replacement field:
  Values in the output typically in the exponential form. (e.g. 1.3000000E01)

8.5 Others

The ODPP_METHOD_DEFAULT option is the only format that is currently supported for the “sMethod” argument in the Provider_Service() API.