Oracle® Data Integrator

Knowledge Modules Developer's Guide

10g Release 3 (10.1.3)

December 2006

Oracle Data Integrator Knowledge Modules Developer's Guide, 10g Release 3 (10.1.3)

Introduction to Knowledge Modules

What is a Knowledge Module?

Knowledge Modules (KMs) are code templates. Each KM is dedicated to an individual task in the overall data integration process. The code in the KMs appears in nearly the form that it will be executed except that it includes Oracle Data Integrator (ODI) substitution methods enabling it to be used generically by many different integration jobs. The code that is generated and executed is derived from the declarative rules and metadata defined in the ODI Designer module.

- A KM will be reused across several interfaces or models. To modify the behavior of hundreds of jobs using hand-coded scripts and procedures, developers would need to modify each script or procedure. In contrast, the benefit of Knowledge Modules is that you make a change once and it is instantly propagated to hundreds of transformations. KMs are based on logical tasks that will be performed. They don't contain references to physical objects (datastores, columns, physical paths, etc.)
- KMs can be analyzed for impact analysis.
- KMs can't be executed standalone. They require metadata from interfaces, datastores and models.

KMs fall into 6 different categories as summarized in the table below:

Knowledge Module	Description	Where used
Reverse-engineering KM	Retrieves metadata to the Oracle Data Integrator work repository	Used in models to perform a customized reverse-engineering
Check KM	Checks consistency of data against constraints	- Used in models, sub models and datastores for data integrity audit - Used in interfaces for flow control or static control
Loading KM	Loads heterogeneous data to a staging area	Used in interfaces with heterogeneous sources
Integration KM	Integrates data from the staging area to a target	Used in interfaces
Journalizing KM	Creates the Change Data Capture framework objects in the source staging area	Used in models, sub models and datastores to create, start and stop journals and to register subscribers.
Service KM	Generates data manipulation web services	Used in models and datastores

Oracle Data Integrator supplies more than 100 Knowledge Modules out-of-the-box.

The following sections describe each type of Knowledge Module.

Reverse-engineering Knowledge Modules (RKM)

The RKM's main role is to perform customized reverse engineering for a model. The RKM is in charge of connecting to the application or metadata provider then transforming and writing the resulting metadata into Oracle Data Integrator's repository. The metadata is written temporarily into the SNP_REV_xx tables. The RKM then calls the Oracle Data Integrator API to read from these tables and write to Oracle Data Integrator's metadata tables of the work repository in incremental update mode. This is illustrated below:

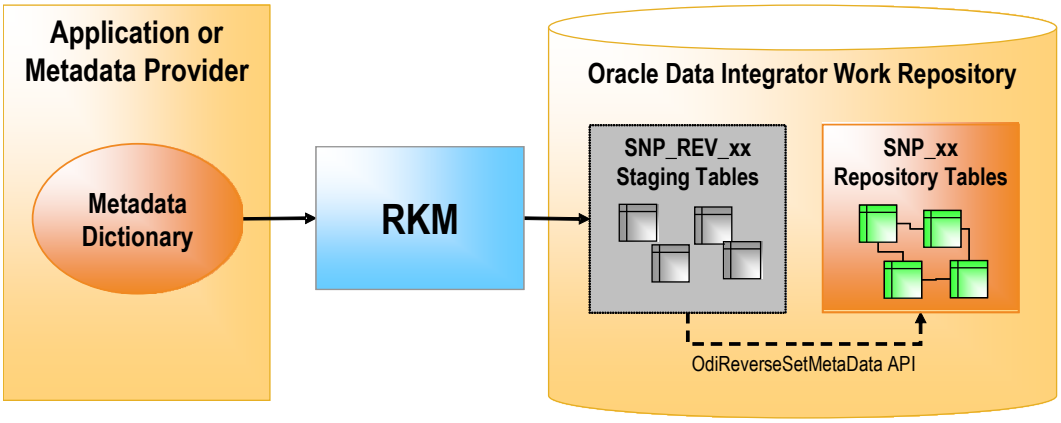

Figure 1: Reverse-engineering Knowledge Module

A typical RKM follows these steps:

1. Cleans up the SNP_REV_xx tables from previous executions using the OdiReverseResetTable command
2. Retrieves sub models, datastores, columns, unique keys, foreign keys, conditions from the metadata provider to SNP_REV_SUB_MODEL, SNP_REV_TABLE, SNP_REV_COL, SNP_REV_KEY, SNP_REV_KEY_COL, SNP_REV_JOIN, SNP_REV_JOIN_COL, SNP_REV_COND tables.
3. Updates the model in the work repository by calling the OdiReverseSetMetaData API.

Check Knowledge Modules (CKM)

The CKM is in charge of checking that records of a data set are consistent with defined constraints. The CKM is used to maintain data integrity and participates in the overall data quality initiative. The CKM can be used in 2 ways:

- To check the consistency of existing data. This can be done on any datastore or within interfaces, by setting the STATIC_CONTROL option to "Yes". In the first case, the data checked is the data currently in the datastore. In the second case, data in the target datastore is checked after it is loaded.
- To check consistency of the incoming data before loading the records to a target datastore. This is done by using the FLOW_CONTROL option. In this case, the CKM simulates the constraints of the target datastore on the resulting flow prior to writing to the target.

In summary: the CKM can check either an existing table or the temporary "I$" table created by an IKM.

The CKM accepts a set of constraints and the name of the table to check. It creates an "E$" error table which it writes all the rejected records to. The CKM can also remove the erroneous records from the checked result set.

The following figures show how a CKM operates in both STATIC_CONTROL and FLOW_CONTROL modes.

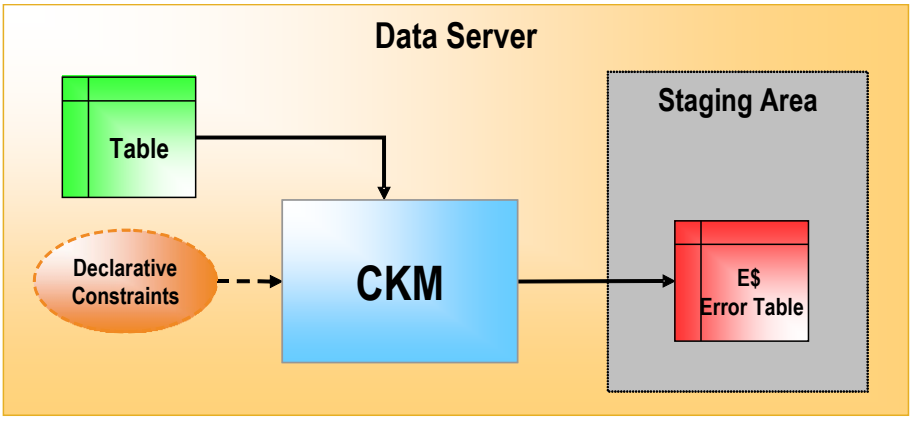

Figure 2: Check Knowledge Module (STATIC_CONTROL)

In STATIC_CONTROL mode, the CKM reads the constraints of the table and checks them against the data of the table. Records that don't match the constraints are written to the "E$" error table in the staging area.

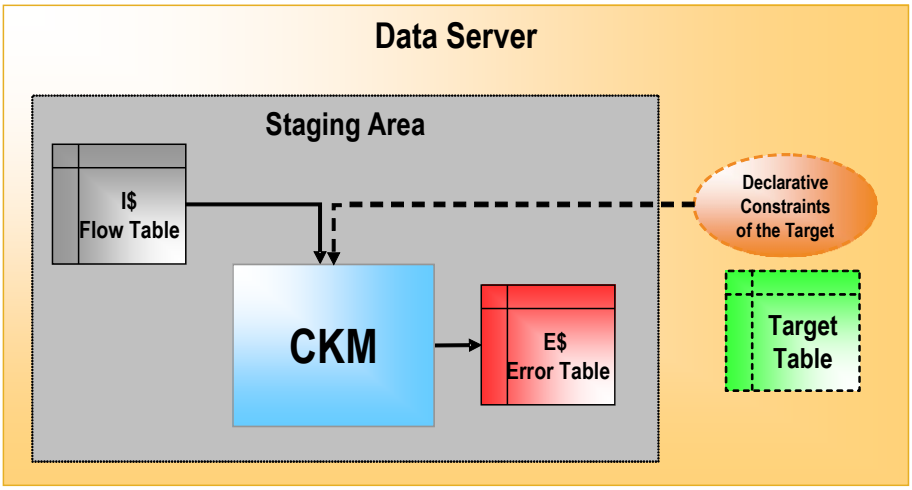

Figure 3: Check Knowledge Module (FLOW_CONTROL)

In FLOW_CONTROL mode, the CKM reads the constraints of the target table of the Interface. It checks these constraints against the data contained in the "I$" flow table of the staging area. Records that violate these constraints are written to the "E$" table of the staging area.

In both cases, a CKM usually performs the following tasks:

1. Create the "E$" error table on the staging area. The error table should contain the same columns as the datastore as well as additional columns to trace error messages, check origin, check date etc.
2. Isolate the erroneous records in the "E$" table for each primary key, alternate key, foreign key, condition, mandatory column that needs to be checked.

3. If required, remove erroneous records from the table that has been checked.

Loading Knowledge Modules (LKM)

An LKM is in charge of loading source data from a remote server to the staging area. It is used by interfaces when some of the source datastores are not on the same data server as the staging area. The LKM implements the declarative rules that need to be executed on the source server and retrieves a single result set that it stores in a "C$" table in the staging area, as illustrated below.

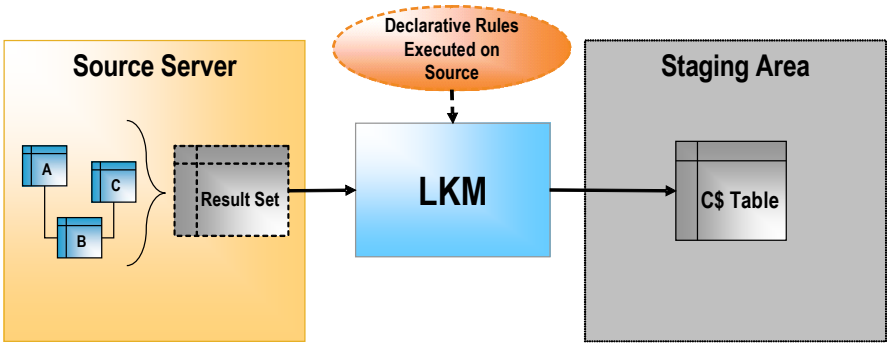

Figure 4: Loading Knowledge Module

1. The LKM creates the "C$" temporary table in the staging area. This table will hold records loaded from the source server.
2. The LKM obtains a set of pre-transformed records from the source server by executing the appropriate transformations on the source. Usually, this is done by a single SQL SELECT query when the source server is an RDBMS. When the source doesn't have SQL capacities (such as flat files or applications), the LKM simply reads the source data with the appropriate method (read file or execute API).
3. The LKM loads the records into the "C$" table of the staging area.

An interface may require several LKMs when it uses datastores from different sources. When all source datastores are on the same data server as the staging area, no LKM is required.

Integration Knowledge Modules (IKM)

The IKM is in charge of writing the final, transformed data to the target table. Every interface uses a single IKM. When the IKM is started, it assumes that all loading phases for the remote servers have already carried out their tasks. This means that all remote source data sets have been loaded by LKMs into "C$" temporary tables in the staging area, or the source datastores are on the same data server as the staging area. Therefore, the IKM simply needs to execute the "Staging and Target" transformations, joins and filters on the "C$" tables, and tables located on the same data server as the staging area. The resulting set is usually processed by the IKM and written into the "I$" temporary table before loading it to the target. These final transformed records can be written in several ways depending on the IKM selected in your interface. They may be simply appended to the target, or compared for incremental updates or for slowly changing dimensions. There are 2 types of IKMs: those that assume that the staging area is on the same server as the target datastore, and those that can be used when it is not. These are illustrated below:

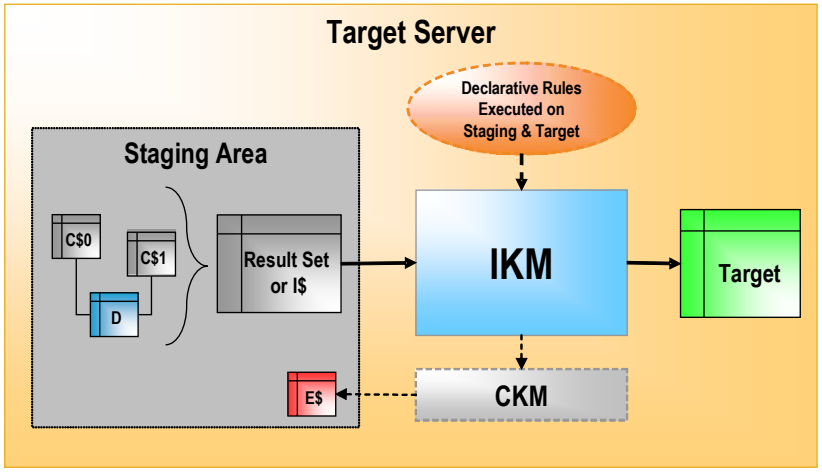

Figure 5: Integration Knowledge Module (Staging Area on Target)

When the staging area is on the target server, the IKM usually follows these steps:

1. The IKM executes a single set-oriented SELECT statement to carry out staging area and target declarative rules on all "C$" tables and local tables (such as D in the figure). This generates a result set.
2. Simple "append" IKMs directly write this result set into the target table. More complex IKMs create an "I$" table to store this result set.
3. If the data flow needs to be checked against target constraints, the IKM calls a CKM to isolate erroneous records and cleanse the "I$" table.
4. The IKM writes records from the "I$" table to the target following the defined strategy (incremental update, slowly changing dimension, etc.).
5. The IKM drops the "I$" temporary table.
6. Optionally, the IKM can call the CKM again to check the consistency of the target datastore.

These types of KMs do not manipulate data outside of the target server. Data processing is set-oriented for maximum efficiency when performing jobs on large volumes.

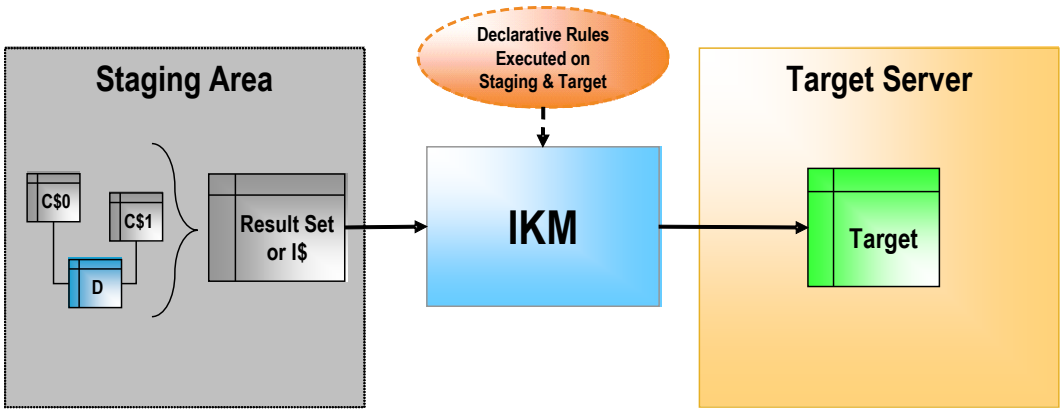

Figure 6: Integration Knowledge Module (Staging Area Different from Target)

When the staging area is different from the target server, as shown in Figure 6, the IKM usually follows these steps:

1. The IKM executes a single set-oriented SELECT statement to carry out declarative rules on all "C$" tables and tables located on the staging area (such as D in the figure). This generates a result set.
2. The IKM loads this result set into the target datastore, following the defined strategy (append or incremental update).

This architecture has certain limitations, such as:

- A CKM cannot be used to perform a data integrity audit on the data being processed.
- Data needs to be extracted from the staging area before being loaded to the target, which may lead to performance issues.

Journalizing Knowledge Modules (JKM)

JKMs create the infrastructure for Change Data Capture on a model, a sub model or a datastore. JKMs are not used in interfaces, but rather within a model to define how the CDC infrastructure is initialized. This infrastructure is composed of a subscribers table, a table of changes, views on this table and one or more triggers or log capture programs as illustrated below.

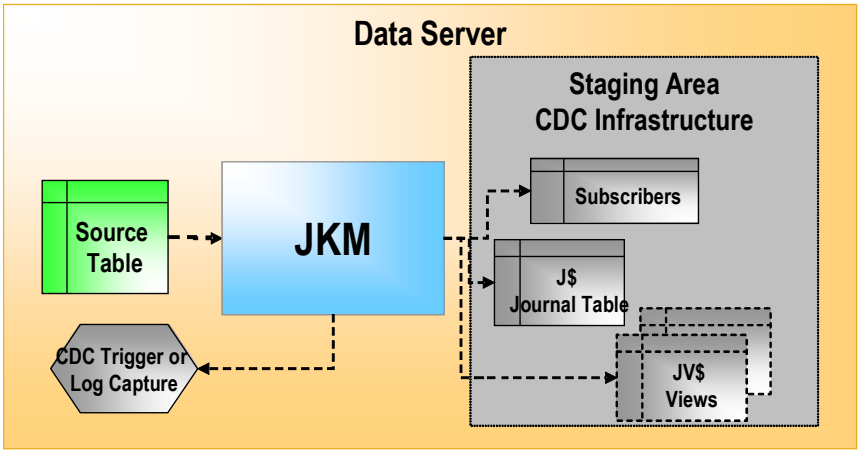

Figure 7: Journalizing Knowledge Module

Service Knowledge Modules (SKM)

SKMs are in charge of creating and deploying data manipulation Web Services to your Service Oriented Architecture (SOA) infrastructure. SKMs are set on a Model. They define the different operations to generate for each datastore's web service. Unlike other KMs, SKMs do no generate an executable code but rather the Web Services deployment archive files. SKMs are designed to generate Java code using Oracle Data Integrator's framework for Web Services. The code is then compiled and eventually deployed on the Application Server's containers.

Oracle Data Integrator Substitution API

KMs are written as templates by using the Oracle Data Integrator substitution API. A detailed reference for this API is provided in the online documentation. The API methods are java methods that return a string value. They all belong to a single object instance named "odiRef". The same method may return different values depending on the type of KM that invokes it. That's why they are classified by type of KM.

> **Note:** For backward compatibility, the "**odiRef**" API can also be referred to as "**snpRef**" API. "snpRef" and "odiRef" object instances are synonyms. Some examples in this section are still using the old snpRef notation rather than the new odiRef notation.

To understand how this API works, the following example illustrates how you would write a create table statement in a KM and what it would generate depending on the datastores it would deal with:

Code inside a KM	`Create table <%=odiRef.getTable("L", "INT_NAME", "A")%>`
	`(`
	`<%=odiRef.getColList("", "\t[COL_NAME] [DEST_CRE_DT]", ",\n", "", "")%>`
	`)`

Generated code for the PRODUCT datastore	`Create table db_staging.I$_PRODUCT`
	`(`
	`PRODUCT_ID numeric(10),`
	`PRODUCT_NAME varchar(250),`
	`FAMILY_ID numeric(4),`
	`SKU varchar(13),`
	`LAST_DATE timestamp`
	`)`

Generated code for the CUSTOMER datastore	`Create table db_staging.I$_CUSTOMER`
	`(`
	`CUST_ID numeric(10),`
	`CUST_NAME varchar(250),`
	`ADDRESS varchar(250),`
	`CITY varchar(50),`
	`ZIP_CODE varchar(12),`
	`COUNTRY_ID varchar(3)`
	`)`

As you can see, once executed with appropriate metadata, the KM has generated a different code for the product and customer tables.

The following topics cover some of the main substitution APIs and their use within KMs. Note that for better readability the tags "<%" and "%>" as well as the "odiRef" object reference are omitted in the examples.

Working with Datastores and Object Names

When working in Designer, you should almost never specify physical information such as the database name or schema name as they may change depending on the execution context. The correct physical information will be provided by Oracle Data Integrator at execution time.

The substitution API has methods that calculate the fully qualified name of an object or datastore taking into account the context at runtime. These methods are listed in the table below:

To Obtain the Full Qualified Name of	Use method	Where Applicable
Any object named MY_OBJECT	`getObjectName("L", "MY_OBJECT", "D")`	All KMs and procedures
The target datastore	`getTable("L", "TARG_NAME", "A")`	LKM, CKM, IKM, JKM
The "I$" datastore	`getTable("L", "INT_NAME", "A")`	LKM, IKM
The "C$" datastore	`getTable("L", "COLL_NAME", "A")`	LKM
The "E$" datastore	`getTable("L", "ERR_NAME", "A")`	LKM, CKM, IKM
The checked datastore	`getTable("L", "CT_NAME", "A")`	CKM
The datastore referenced by a foreign key	`getTable("L", "FK_PK_TABLE_NAME", "A")`	CKM

Working with Lists of Tables, Columns and Expressions

Generating code from a list of items often requires a "while" or "for" loop. Oracle Data Integrator addresses this issue by providing powerful methods that help you generate code based on lists. These methods act as "iterators" to which you provide a substitution mask or pattern and a separator and they return a single string with all patterns resolved separated by the separator.

All of them return a string and accept at least these 4 parameters:

- Start: a string used to start the resulting string.
- Pattern: a substitution mask with attributes that will be bound to the values of each item of the list.
- Separator: a string used to separate each substituted pattern from the following one.
- End: a string appended to the end of the resulting string

Some of them accept an additional parameter (Selector) that acts as a filter to retrieve only part of the items of the list.

Some of these methods are summarized in the table below:

Method	Description	Where Applicable
`getColList()`	The most frequently-used method in Oracle Data Integrator. It returns a list of columns and expressions that need to be executed in the context where used. You can use it, for example, to generate lists like	LKM, CKM, IKM, JKM, SKM

Method	Description	Where Applicable
	these: - Columns in a CREATE TABLE statement - Columns of the update key - Expressions for a SELECT statement in a LKM, CKM or IKM - Field definitions for a loading script This method accepts a "selector" as a 5th parameter to let you filter items as desired.	
`getTargetColList()`	Returns the list of columns in the target datastore. This method accepts a selector as a 5th parameter to let you filter items as desired.	LKM, CKM, IKM, JKM,SKM
`getAKColList()`	Returns the list of columns defined for an alternate key.	CKM, SKM
`getPKColList()`	Returns the list of columns in a primary key. You can alternatively use `getColList` with the selector parameter set to "PK" .	CKM,SKM
`getFKColList()`	Returns the list of referencing columns and referenced columns of the current foreign key.	CKM,SKM
`getSrcTablesList()`	Returns the list of source tables of an interface. Whenever possible, use the `getFrom` method instead. The `getFrom` method is discussed below.	LKM, IKM
`getFilterList()`	Returns the list of filter expressions in an interface. The `getFilter` method is usually more appropriate.	LKM, IKM
`getJoinList()`	Returns the list of join expressions in an interface. The `getJoin` method is usually more appropriate.	LKM, IKM
`getGrpByList()`	Returns the list of expressions that should appear in the group by clause when aggregate functions are detected in the mappings of an interface. The `getGrpBy` method is usually more appropriate.	LKM, IKM
`getHavingList()`	Returns the list of expressions that should appear in the having clause when aggregate functions are detected in the filters of an interface. The `getHaving` method is usually more appropriate.	LKM, IKM
`getSubscriberList()`	Returns a list of subscribers.	JKM

The following examples illustrate how these methods work for generating code:

Using getTargetColList to create a table

Code in	`Create table MYTABLE`

your KM	<%=odiRef.getTargetColList("(\n", "\t[COL_NAME] [DEST_WRI_DT]", ",\n", "\n)")%>
Code Generated	Create table MYTABLE (CUST_ID numeric(10), CUST_NAME varchar(250), ADDRESS varchar(250), CITY varchar(50), ZIP_CODE varchar(12), COUNTRY_ID varchar(3))

- Start is set to `"(\n"`: The generated code will start with a parenthesis followed by a carriage return (\n).
- Pattern is set to `"\t[COL_NAME] [DEST_WRI_DT]"`: The generated code will loop over every target column and generate a tab character (\t) followed by the column name ([COL_NAME]), a white space and the destination writable data type ([DEST_WRI_DT]).
- The Separator is set to `",\n"`: Each generated pattern will be separated from the next one with a comma (,) and a carriage return (\n)
- End is set to `"\n)"`: The generated code will end with a carriage return (\n) followed by a parenthesis.

Using getColList in an Insert values statement

Code in your KM	insert into MYTABLE (<%=odiRef.getColList("", "[COL_NAME]", ", ", "\n", "INS AND NOT TARG")%> <%=odiRef.getColList(",", "[COL_NAME]", ", ", "", "INS AND TARG")%>) Values (<%=odiRef.getColList("", ":[COL_NAME]", ", ", "\n", "INS AND NOT TARG")%> <%=odiRef.getColList(", ", "[EXPRESSION]", ", ", "", "INS AND TARG")%>)
Code Generated	insert into MYTABLE (CUST_ID, CUST_NAME, ADDRESS, CITY, COUNTRY_ID , ZIP_CODE, LAST_UPDATE) Values (:CUST_ID, :CUST_NAME, :ADDRESS, :CITY, :COUNTRY_ID , 'ZZ2345', current_timestamp)

In this example, the values that need to be inserted into MYTABLE are either bind variables with the same name as the target columns or constant expressions if they are executed on the target. To obtain these 2 distinct set of items, the list is split using the selector parameter:

- "INS AND NOT TARG": first, generate a comma-separated list of columns ([COL_NAME]) mapped to bind variables in the "value" part of the statement (:[COL_NAME]). Filter them to get only the ones that are flagged to be part of the INSERT statement and that are **not executed on the target.**
- "INS AND TARG": then generate a comma separated list of columns ([COL_NAME]) corresponding to expression ([EXPRESSION]) that are flagged to be part of the INSERT statement and that are **executed on the target**. The list should start with a comma if any items are found.

Using getSrcTableList

Code in your KM	`insert into MYLOGTABLE`
	`(`
	`INTERFACE_NAME,`
	`DATE_LOADED,`
	`SOURCE_TABLES`
	`)`
	`values`
	`(`
	`'<%=odiRef.getPop("POP_NAME")%>',`
	`current_date,`
	`'' <%=odiRef.getSrcTablesList("|| ", "'[RES_NAME]'", " || ',' || ", "")%>`
	`)`

Code Generated	`insert into MYLOGTABLE`
	`(`
	`INTERFACE_NAME,`
	`DATE_LOADED,`
	`SOURCE_TABLES`
	`)`
	`values`
	`(`
	`'Int. CUSTOMER',`
	`current_date,`
	`'' || 'SRC_CUST' || ',' || 'AGE_RANGE_FILE' || ',' || 'C$0_CUSTOMER'`
	`)`

In this example, `getSrcTableList` generates a message containing the list of resource names used as sources in the interface to append to MYLOGTABLE. The separator used is composed of a concatenation operator (||) followed by a comma enclosed by quotes (',') followed by the same operator again. When the table list is empty, the SOURCE_TABLES column of MYLOGTABLE will be mapped to an empty string ('').

Generating the Source Select Statement

LKMs and IKMs both manipulate a source result set. For the LKM, this result set represents the pre-transformed records according to the mappings, filters and joins that need to be executed on the source. For the IKM, however, the result set represents the transformed records matching the mappings, filters and joins executed on the staging area.

To build these result sets, you will usually use a SELECT statement in your KMs. Oracle Data Integrator has some advanced substitution methods, including `getColList`, that help you generate this code:

Method	Description	Where Applicable
`getFrom()`	Returns the FROM clause of a SELECT statement with the appropriate source tables, left, right and full outer joins. This method uses information from the topology to determine the SQL capabilities of the source or target technology. The FROM clause is built accordingly with the appropriate keywords (INNER, LEFT etc.) and parentheses when supported by the technology. - When used in an LKM, it returns the FROM clause as it should be executed by the source server. - When used in an IKM, it returns the FROM clause as it should be executed by the staging area server.	LKM, IKM
`getFilter()`	Returns filter expressions separated by an "AND" operator. - When used in an LKM, it returns the filter clause as it should be executed by the source server. - When used in an IKM, it returns the filter clause as it should be executed by the staging area server.	LKM, IKM
`getJrnFilter()`	Returns the special journal filter expressions for the journalized source datastore. This method should be used with the CDC framework.	LKM, IKM
`getGrpBy()`	Returns the GROUP BY clause when aggregation functions are detected in the mappings. The GROUP BY clause includes all mapping expressions referencing columns that do not contain aggregation functions. The list of aggregation functions are defined by the language of the technology in the topology.	LKM, IKM
`getHaving()`	Returns the HAVING clause when aggregation functions are detected in filters. The having clause includes all filters expressions containing aggregation functions. The list of aggregation functions are defined by the language of the technology in the topology.	LKM, IKM

To obtain the result set from any SQL RDBMS source server, you would use the following SELECT statement in your LKM:

```
select  <%=odiRef.getPop("DISTINCT_ROWS")%>
        <%=odiRef.getColList("", "[EXPRESSION]\t[ALIAS_SEP] [CX_COL_NAME]", ",\n\t", "", "")%>
from    <%=odiRef.getFrom()%>
where   (1=1)
<%=odiRef.getFilter()%>
<%=odiRef.getJrnFilter()%>
<%=odiRef.getJoin()%>
<%=odiRef.getGrpBy()%>
<%=odiRef.getHaving()%>
```

To obtain the result set from any SQL RDBMS staging area server to build your final flow data, you would use the following SELECT statement in your IKM. Note that the `getColList` is filtered to retrieve only expressions that are not executed on the target and that are mapped to writable columns.

```
select  <%=odiRef.getPop("DISTINCT_ROWS")%>
        <%=odiRef.getColList("", "[EXPRESSION]", ",\n\t", "", "(not TRG) and REW")%>
from    <%=odiRef.getFrom()%>
where   (1=1)
<%=odiRef.getJoin()%>
<%=odiRef.getFilter()%>
<%=odiRef.getJrnFilter()%>
<%=odiRef.getGrpBy()%>
<%=odiRef.getHaving()%>
```

As all filters and joins start with an AND, the WHERE clause of the SELECT statement starts with a condition that is always true (1=1).

Obtaining Other Information with the API

The following methods provide additional information which may be useful:

Method	Description	Where Applicable
`getPop()`	Returns information about the current interface.	LKM, IKM
`getInfo()`	Returns information about the source or target server.	Any procedure or KM
`getSession()`	Returns information about the current running session	Any procedure or KM
`getOption()`	Returns the value of a particular option	Any procedure or KM
`getFlexFieldValue()`	Returns information about a flex field value. Not that with the "List" methods, flex field values can be specified as part of the pattern parameter.	Any procedure or KM

Method	Description	Where Applicable
`getJrnInfo()`	Returns information about the CDC framework	JKM, LKM, IKM
`getTargetTable()`	Returns information about the target table of an interface	LKM, IKM, CKM
`getModel()`	Returns information about the current model during a reverse-engineering process.	RKM

Advanced Techniques for Code Generation

You can use conditional branching and advanced programming techniques to generate code. The code generation in Oracle Data Integrator is able to interpret any Java code enclosed between "<%" and "%>" tags. Refer to http://java.sun.com for a complete reference for the Java language.

The following examples illustrate how you can use these advanced techniques:

Code in the KM or procedure	Generated code

`<%`
`String myTableName;`
`myTableName = "ABCDEF";`
`%>`
`drop table`
`<%=odiRef.getObjectName(myTableName.toLowerCase())%>`
	`drop table SCOTT.abcdef`

`<%`
`String myOptionValue=odiRef.getOption("Test");`
 	
`if (myOption.equals("TRUE"))`
`{`
`out.print("/* Option Test is set to TRUE */");`
`}`
`else`
`{`
`%>`
`/* The Test option is not properly set */`
`<%`
`}`
`%>`
`...`
	When option Test is set to TRUE:
	`/* Option Test is set to TRUE */`
	`...`
	
	Otherwise:
	`/* The Test option is not properly set */`
	`...`

`Create table <%=odiRef.getObjectName("XYZ")%>`
`(`
`<%`
`String s;`
`s = "ABCDEF";`
`for (int i=0; i < s.length(); i++)`
	`Create table ADAMS.XYZ`
	`(`
	`A char(1),`
	`B char(1),`
	`C char(1),`
	`D char(1),`

```
{
%>
<%=s.charAt(i)%> char(1),
<%
}
%>
G char(1)
)
```

```
E char(1),
F char(1),
G char(1)
)
```

Reverse-engineering Knowledge Modules (RKM)

RKM Process

Customizing a reverse-engineering strategy using an RKM is normally straightforward. The steps are usually the same from one RKM to another:

1. Call the OdiReverseResetTable command to reset the SNP_REV_xx tables from previous executions.
2. Retrieve sub models, datastores, columns, unique keys, foreign keys, conditions from the metadata provider to SNP_REV_SUB_MODEL, SNP_REV_TABLE, SNP_REV_COL, SNP_REV_KEY, SNP_REV_KEY_COL, SNP_REV_JOIN, SNP_REV_JOIN_COL, SNP_REV_COND tables. Refer to section SNP_REV_xx Tables Reference for more information about the SNP_REVxx tables.
3. Call the OdiReverseSetMetaData command to apply the changes to the current Oracle Data Integrator model.

As an example, the steps below are extracted from the RKM for Oracle. Refer to the Knowledge Modules Reference Guide for additional information on this RKM:

Step	Example of code
Reset SNP_REV tables	`OdiReverseResetTable -MODEL=<%=odiRef.getModel("ID")%>`

Get Tables and views	`/*=======================*/`
	`/* Command on the source */`
	`/*=======================*/`
	`Select t.TABLE_NAME TABLE_NAME,`
	`t.TABLE_NAME RES_NAME,`
	
	`replace(t.TABLE_NAME,'<%=snpRef.getModel("REV_ALIAS_LTRIM")%>','')`
	`TABLE_ALIAS,`
	`substr(tc.COMMENTS,1,250) TABLE_DESC,`
	`t.NUM_ROWS R_COUNT`
	`From ALL_TABLES t,`
	`ALL_TAB_COMMENTS tc`
	`Where t.OWNER = '<%=snpRef.getModel("SCHEMA_NAME")%>'`
	`and t.TABLE_NAME like '<%=snpRef.getModel("REV_OBJ_PATT")%>'`
	`and tc.OWNER(+) = t.OWNER`
	`and tc.TABLE_NAME(+) = t.TABLE_NAME`
	
	`/*=======================*/`
	`/* Command on the target */`
	`/*=======================*/`
	`insert into SNP_REV_TABLE`

Step	Example of code
	(I_MOD, TABLE_NAME,RES_NAME,TABLE_ALIAS,TABLE_TYPE,TABLE_DESC, IND_SHOW,R_COUNT) values (<%=odiRef.getModel("ID")%>,:TABLE_NAME,:RES_NAME,:TABLE_ALIAS,'T', :TABLE_DESC,'1',:R_COUNT)
Get Table Columns	/*=======================*/ /* Command on the source */ /*=======================*/ select c.TABLE_NAME TABLE_NAME, c.COLUMN_NAME COL_NAME, c.DATA_TYPE DT_DRIVER, substr(cc.COMMENTS,1,250) COL_DESC, c.COLUMN_ID POS, decode(C.DATA_TYPE, 'NUMBER', c.DATA_PRECISION, nvl(c.DATA_PRECISION,c.DATA_LENGTH)) LONGC, c.DATA_SCALE SCALEC, decode(c.NULLABLE,'Y','0','1') COL_MANDATORY from ALL_TAB_COLUMNS c, ALL_COL_COMMENTS cc, ALL_OBJECTS o Where o.OWNER = '<%=snpRef.getModel("SCHEMA_NAME")%>' and (o.OBJECT_TYPE = 'TABLE' or o.OBJECT_TYPE = 'VIEW') and o.OBJECT_NAME like '<%=snpRef.getModel("REV_OBJ_PATT")%>' and cc.OWNER(+) = c.OWNER and cc.TABLE_NAME(+) = c.TABLE_NAME and cc.COLUMN_NAME(+) = c.COLUMN_NAME and o.OWNER = c.OWNER and o.OBJECT_NAME = c.TABLE_NAME /*=======================*/ /* Command on the target */ /*=======================*/ insert into SNP_REV_COL (I_MOD,TABLE_NAME,COL_NAME,DT_DRIVER,COL_DESC,POS,LONGC,SCALEC,COL_MANDATORY,CHECK_STAT,CHECK_FLOW) values (<%=odiRef.getModel("ID")%>,:TABLE_NAME,:COL_NAME,:DT_DRIVER,:COL_DESC,:POS,:LONGC,:SCALEC,:COL_MANDATORY,'1','1')
Etc.	
Set Metadata	OdiReverseSetMetaData -MODEL=<%=odiRef.getModel("ID")%>

Refer to the following RKMs for further details:

RKM	Description
RKM Oracle	Reverse-engineering Knowledge Module for Oracle

RKM	Description
RKM Teradata	Reverse-engineering Knowledge Module for Teradata
RKM DB2 400	Reverse-engineering Knowledge Module for DB2/400. Retrieves the short name of tables rather than the long name.
RKM File (FROM EXCEL)	Reverse-engineering Knowledge Module for Files, based on a description of files in a Microsoft Excel spreadsheet.
RKM Informix SE	Reverse-engineering Knowledge Module for Informix Standard Edition
RKM Informix	Reverse-engineering Knowledge Module for Informix
RKM SQL (JYTHON)	Reverse-engineering Knowledge Module for any JDBC compliant database. Uses Jython to call the JDBC API.

SNP_REV_xx Tables Reference

SNP_REV_SUB_MODEL

Description: Reverse-engineering temporary table for sub-models.

Column	Type	Mandatory	Description
I_MOD	numeric(10)	Yes	Internal ID of the model
SMOD_CODE	varchar(35)	Yes	Code of the sub-model
SMOD_NAME	varchar(100)		Name of the sub-model
SMOD_PARENT_CODE	varchar(35)		Code of the parent sub-model
IND_INTEGRATION	varchar(1)		Used internally by the OdiReverserSetMetadata API
TABLE_NAME_PATTERN	varchar(35)		Automatic assignment mask used to distribute datastores in this sub-model
REV_APPY_PATTERN	varchar(1)		Datastores distribution rule: 0: No distribution 1: Automatic distribution of all datastores not already in a sub-model 2: Automatic distribution of all datastores

SNP_REV_TABLE

Description: The temporary table for reverse-engineering datastores.

Column	Type	Mandatory	Description
I_MOD	numeric(10)	Yes	Internal ID of the model
TABLE_NAME	varchar(100)	Yes	Name of the datastore
RES_NAME	varchar(250)		Physical name of the datastore
TABLE_ALIAS	varchar(35)		Default alias for this datastore
TABLE_TYPE	varchar(2)		Type of datastore: T: Table or file V: View Q: Queue or Topic ST: System table AT: Table alias SY: Synonym
TABLE_DESC	varchar(250)		Datastore description
IND_SHOW	varchar(1)		Indicates whether this datastore is displayed or hidden: 0: Hidden 1: Displayed
R_COUNT	numeric(10)		Estimated row count
FILE_FORMAT	varchar(1)		Record format (applies only to files and JMS messages): F: Fixed length file D: Delimited file
FILE_SEP_FIELD	varchar(8)		Field separator (only applies to files and JMS messages)
FILE_ENC_FIELD	varchar(2)		Text delimiter (only applies to files and JMS messages)
FILE_SEP_ROW	varchar(8)		Row separator (only applies to files and JMS messages)
FILE_FIRST_ROW	numeric(10)		Numeric or records to skip in the file (only applies to files and JMS messages)
FILE_DEC_SEP	varchar(1)		Default decimal separator for numeric fields of the file (only applies to files and JMS messages)
SMOD_CODE	varchar(35)		Code of the sub-model this table should be place in. If null, the table will be placed in the main model.

SNP_REV_COL

Description: Reverse-engineering temporary table for columns.

Column	Type	Mandatory	Description
I_MOD	numeric(10)	Yes	Internal ID of the model
TABLE_NAME	varchar(100)	Yes	Name of the table
COL_NAME	varchar(100)	Yes	Name of the column
COL_HEADING	varchar(35)		Short description of the column
COL_DESC	varchar(250)		Long description of the column
DT_DRIVER	varchar(35)		Data type of the column. This data type should match the data type code as defined in Oracle Data Integrator Topology for this technology
POS	numeric(10)		Ordinal position of the column in the table
LONGC	numeric(10)		Character length or numeric precision radix of the column
SCALEC	numeric(10)		Decimal digits of the column
FILE_POS	numeric(10)		Start byte position of the column in a fixed length file (applies only to files and JMS messages)
BYTES	numeric(10)		Numeric of bytes of the column (applies only to files and JMS messages)
IND_WRITE	varchar(1)		Indicates whether the column is writable: 0 → No, 1 → Yes
COL_MANDATORY	varchar(1)		Indicates whether the column is mandatory: 0 → No, 1 →Yes
CHECK_FLOW	varchar(1)		Indicates whether to include the mandatory constraint by default in the flow control: 0 →No, 1 →yes
CHECK_STAT	varchar(1)		Indicates whether to include the mandatory constraint by default in the static control: 0 → No, 1 →yes
COL_FORMAT	varchar(35)		Column format. Usually this field applies only to files and JMS messages to explain the date format.
COL_DEC_SEP	varchar(1)		Decimal separator for the column (applies only to files and JMS messages)
REC_CODE_LIST	varchar(250)		Record code to filter multiple record files (applies only to files and JMS messages)
COL_NULL_IF_ERR	varchar(1)		Indicates whether to set this column to null in case of error (applies only to files and JMS messages)

SNP_REV_KEY

Description: Temporary table for reverse-engineering primary keys, alternate keys and indexes.

Column	Type	Mandatory	Description
I_MOD	numeric(10)	Yes	Internal ID of the model
TABLE_NAME	varchar(100)	Yes	Name of the table
KEY_NAME	varchar(100)	Yes	Name of the key or index
CONS_TYPE	varchar(2)	Yes	Type of key: PK: Primary key AK: Alternate key I: Index
IND_ACTIVE	varchar(1)		Indicates whether this constraint is active: 0 → No, 1 →yes
CHECK_FLOW	varchar(1)		Indicates whether to include this constraint by default in flow control: 0 →No, 1 →yes
CHECK_STAT	varchar(1)		Indicates whether to include constraint by default in static control: 0 →No, 1 →yes

SNP_REV_KEY_COL

Description: Temporary table for reverse-engineering columns that form part of a primary key, alternate key or index.

Column	Type	Mandatory	Description
I_MOD	numeric(10)	Yes	Internal ID of the model
TABLE_NAME	varchar(100)	Yes	Name of the table
KEY_NAME	varchar(100)	Yes	Name of the key or index
COL_NAME	varchar(100)	Yes	Name of the column belonging to the key
POS	numeric(10)		Ordinal position of the column in the key

SNP_REV_JOIN

Description: Temporary table for reverse-engineering references (foreign keys).

Column	Type	Mandatory	Description
I_MOD	numeric(10)	Yes	Internal ID of the model

Column	Type	Mandatory	Description
FK_NAME	varchar(100)	Yes	Name of the reference or foreign key
TABLE_NAME	varchar(100)	Yes	Name of the referencing table
FK_TYPE	varchar(1)		Type of foreign key: D: Database foreign key U: User-defined foreign key C: Complex user-defined foreign key
PK_CATALOG	varchar(35)		Catalog of the referenced table
PK_SCHEMA	varchar(35)		Schema of the referenced table
PK_TABLE_NAME	varchar(100)		Name of the referenced table
CHECK_STAT	varchar(1)		Indicates whether to include constraint by default in the static control: 0 → No, 1 → yes
CHECK_FLOW	varchar(1)		Indicates whether to include constraint by default in the flow control: 0 →No, 1 →yes
IND_ACTIVE	varchar(1)		Indicates whether this constraint is active: 0 →No, 1 →yes
DEFER	varchar(1)		Reserved for future use
UPD_RULE	varchar(1)		Reserved for future use
DEL_RULE	varchar(1)		Reserved for future use

SNP_REV_JOIN_COL

Description: Temporary table for reverse-engineering columns that form part of a reference (or foreign key).

Column	Type	Mandatory	Description
I_MOD	numeric(10)	Yes	Internal ID of the model
FK_NAME	varchar(100)	Yes	Name of the reference or foreign key
FK_COL_NAME	varchar(100)	Yes	Column name of the referencing table
FK_TABLE_NAME	varchar(100)		Name of the referencing table
PK_COL_NAME	varchar(100)	Yes	Column name of the referenced table
PK_TABLE_NAME	varchar(100)		Name of the referenced table
POS	numeric(10)		Ordinal position of the column in the foreign key

SNP_REV_COND

Description: Temporary table for reverse-engineering conditions and filters (check constraints).

Column	Type	Mandatory	Description
I_MOD	numeric(10)	Yes	Internal ID of the model
TABLE_NAME	varchar(100)	Yes	Name of the table
COND_NAME	varchar(35)	Yes	Name of the condition or check constraint
COND_TYPE	varchar(1)	Yes	Type of condition: C: Oracle Data Integrator condition D: Database condition F: Permanent filter
COND_SQL	varchar(250)		SQL expression for applying this condition or filter
COND_MESS	varchar(250)		Error message for this condition
IND_ACTIVE	varchar(1)		Indicates whether this constraint is active: 0 -> No, 1 -> yes
CHECK_STAT	varchar(1)		Indicates whether to include constraint by default in the static control: 0 -> No, 1 -> yes
CHECK_FLOW	varchar(1)		Indicates whether to include constraint by default in the flow control: 0 -> No, 1 -> yes

Data Integrity Strategies (CKM)

Standard Check Knowledge Modules

A CKM is in charge of checking the data quality of a datastore according to a predefined set of constraints. The CKM can be used either to check existing data when used in a "static control" or to check flow data when used in a "flow control" invoked from an IKM. It is also in charge of removing the erroneous records from the checked table if specified.

Standard CKMs maintain 2 different types of tables:

- A single summary table named SNP_CHECK_TAB for every data server, created in the work schema of the default physical schema of the data server. This table contains a summary of the errors for every table and constraint. It can be used, for example, to analyze the overall data quality of the data warehouse.
- An error table named E$_<DatastoreName> for every datastore that was checked. The error table contains the actual records rejected by the data quality process.

The recommended columns for these tables are listed below:

Table	Column	Description
SNP_CHECK_TAB	CATALOG_NAME	Catalog name of the checked table, where applicable
	SCHEMA_NAME	Schema name of the checked table, where applicable
	RESOURCE_NAME	Resource name of the checked table
	FULL_RES_NAME	Fully qualified name of the checked table. For example <catalog>.<schema>.<table>
	ERR_TYPE	Type of error: - 'F' when the datastore is checked during flow control - 'S' when the datastore is checked using static control
	ERR_MESS	Error message
	CHECK_DATE	Date and time when the datastore was checked
	ORIGIN	Origin of the check operation. This column is set either to the datastore name or to an interface name and ID depending on how the check was performed.
	CONS_NAME	Name of the violated constraint.
	CONS_TYPE	Type of constraint: - 'PK': Primary Key - 'AK': Alternate Key - 'FK': Foreign Key - 'CK': Check condition - 'NN': Mandatory column
	ERR_COUNT	Total number of records rejected by this constraint during the check process

Table	Column	Description
"E$" Error table	[Columns of the checked table]	The error table contains all the columns of the checked datastore.
	ERR_TYPE	Type of error: - 'F' when the datastore is checked during flow control - 'S' when the datastore is checked using static control
	ERR_MESS	Error message related to the violated constraint
	CHECK_DATE	Date and time when the datastore was checked
	ORIGIN	Origin of the check operation. This column is set either to the datastore name or to an interface name and ID depending on how the check was performed.
	CONS_NAME	Name of the violated constraint.
	CONS_TYPE	Type of the constraint: - 'PK': Primary Key - 'AK': Alternate Key - 'FK': Foreign Key - 'CK': Check condition - 'NN': Mandatory column

A standard CKM is composed of the following steps:

- Drop and create the summary table. The DROP statement is executed only if the designer requires it for resetting the summary table. The CREATE statement is always executed but the error is tolerated if the table already exists.
- Remove the summary records from the previous run from the summary table
- Drop and create the error table. The DROP statement is executed only if the designer requires it for recreating the error table. The CREATE statement is always executed but error is tolerated if the table already exists.
- Remove rejected records from the previous run from the error table
- Reject records that violate the primary key constraint.
- Reject records that violate any alternate key constraint
- Reject records that violate any foreign key constraint
- Reject records that violate any check condition constraint
- Reject records that violate any mandatory column constraint
- Remove rejected records from the checked table if required
- Insert the summary of detected errors in the summary table.

CKM commands should be tagged to indicate how the code should be generated. The tags can be:

- "Primary Key": The command defines the code needed to check the primary key constraint
- "Alternate Key": The command defines the code needed to check an alternate key constraint.

During code generation, Oracle Data Integrator will use this command for every alternate key

- "Join": The command defines the code needed to check a foreign key constraint. During code generation, Oracle Data Integrator will use this command for every foreign key
- "Condition": The command defines the code needed to check a condition constraint. During code generation, Oracle Data Integrator will use this command for every check condition
- "Mandatory": The command defines the code needed to check a mandatory column constraint. During code generation, Oracle Data Integrator will use this command for mandatory column
- "Remove Errors": The command defines the code needed to remove the rejected records from the checked table.

Extracts from the CKM Oracle are provided below:

Step	Example of code	Conditions to Execute
drop check table	`drop table` `<%=snpRef.getTable("L","CHECK_NAME","W")%>`	Always. Error Tolerated

create check table	`create table`
	`<%=snpRef.getTable("L","CHECK_NAME","W")%>`
	`(`
	`CATALOG_NAME`
	`<%=snpRef.getDataType("DEST_VARCHAR", "100", "")%> <%=snpRef.getInfo("DEST_DDL_NULL")%> ,`
	`SCHEMA_NAME`
	`<%=snpRef.getDataType("DEST_VARCHAR", "100", "")%> <%=snpRef.getInfo("DEST_DDL_NULL")%> ,`
	`RESOURCE_NAME`
	`<%=snpRef.getDataType("DEST_VARCHAR", "100", "")%> <%=snpRef.getInfo("DEST_DDL_NULL")%>,`
	`FULL_RES_NAME`
	`<%=snpRef.getDataType("DEST_VARCHAR", "100", "")%> <%=snpRef.getInfo("DEST_DDL_NULL")%>,`
	`ERR_TYPE`
	`<%=snpRef.getDataType("DEST_VARCHAR", "1", "")%> <%=snpRef.getInfo("DEST_DDL_NULL")%>,`
	`ERR_MESS`
	`<%=snpRef.getDataType("DEST_VARCHAR", "250", "")%> <%=snpRef.getInfo("DEST_DDL_NULL")%> ,`
	`CHECK_DATE`
	`<%=snpRef.getDataType("DEST_DATE", "", "")%> <%=snpRef.getInfo("DEST_DDL_NULL")%>,`
	`ORIGIN`
	`<%=snpRef.getDataType("DEST_VARCHAR", "100", "")%> <%=snpRef.getInfo("DEST_DDL_NULL")%>,`
	`CONS_NAME`
	`<%=snpRef.getDataType("DEST_VARCHAR", "35", "")%> <%=snpRef.getInfo("DEST_DDL_NULL")%>,`
	`CONS_TYPE`
	`<%=snpRef.getDataType("DEST_VARCHAR", "2", "")%> <%=snpRef.getInfo("DEST_DDL_NULL")%>,`
	`ERR_COUNT`
	`<%=snpRef.getDataType("DEST_NUMERIC", "10", "")%> <%=snpRef.getInfo("DEST_DDL_NULL")%>`
	`)`
		Always. Error Tolerated

Create the error table	`create table <%=snpRef.getTable("L","ERR_NAME", "W")%>`	Always. Error Tolerated

Step	Example of code	Conditions to Execute
	(ROW_ID ROWID, ERR_TYPE <%=snpRef.getDataType("DEST_VARCHAR", "1", "")%> <%=snpRef.getInfo("DEST_DDL_NULL")%>, ERR_MESS <%=snpRef.getDataType("DEST_VARCHAR", "250", "")%> <%=snpRef.getInfo("DEST_DDL_NULL")%>, CHECK_DATE <%=snpRef.getDataType("DEST_DATE", "", "")%> <%=snpRef.getInfo("DEST_DDL_NULL")%>, <%=snpRef.getColList("", "[COL_NAME]\t[DEST_WRI_DT] " + snpRef.getInfo("DEST_DDL_NULL"), ",\n\t", "", "")%>, ORIGIN <%=snpRef.getDataType("DEST_VARCHAR", "100", "")%> <%=snpRef.getInfo("DEST_DDL_NULL")%>, CONS_NAME <%=snpRef.getDataType("DEST_VARCHAR", "35", "")%> <%=snpRef.getInfo("DEST_DDL_NULL")%>, CONS_TYPE <%=snpRef.getDataType("DEST_VARCHAR", "2", "")%> <%=snpRef.getInfo("DEST_DDL_NULL")%>)	
Isolate PK errors	insert into <%=snpRef.getTable("L","ERR_NAME", "W")%> (ROW_ID, ERR_TYPE, ERR_MESS, ORIGIN, CHECK_DATE, CONS_NAME, CONS_TYPE, <%=snpRef.getColList("", "[COL_NAME]", ",\n\t", "", "MAP")%>) select rowid, '<%=snpRef.getInfo("CT_ERR_TYPE")%>', '<%=snpRef.getPK("MESS")%>', '<%=snpRef.getInfo("CT_ORIGIN")%>', <%=snpRef.getInfo("DEST_DATE_FCT")%>, '<%=snpRef.getPK("KEY_NAME")%>', 'PK', <%=snpRef.getColList("", snpRef.getTargetTable("TABLE_ALIAS")+".[COL_NAME]", ",\n\t", "", "MAP")%> from <%=snpRef.getTable("L", "CT_NAME", "A")%> <%=snpRef.getTargetTable("TABLE_ALIAS")%> where (	Primary Key

Step	Example of code	Conditions to Execute
	<%=snpRef.getColList("",snpRef.getTargetTable("TABLE_ALIAS")+".[COL_NAME]", ",\n\t\t", "", "PK")%>) in (select <%=snpRef.getColList("", "[COL_NAME]", ",\n\t\t\t\t", "", "PK")%> from <%=snpRef.getTable("L","CT_NAME","A")%> group by <%=snpRef.getColList("", "[COL_NAME]", ",\n\t\t\t\t", "", "PK")%> having count(1) > 1) <%=snpRef.getFilter()%>	
Remove errors from checked table	delete from <%=snpRef.getTable("L", "CT_NAME", "A")%> T where T.rowid in (select ROW_ID from <%=snpRef.getTable("L","ERR_NAME", "W")%>)	Remove Errors

Note:
When using a CKM to perform flow control from an interface, you can define the maximum number of errors allowed. This number is compared to the total number of records returned by every command in the CKM of which the "Log Counter" is set to "Error".

Case Study: Customizing a CKM to Dynamically Create Non-Existing References

In some cases, when loading a data warehouse for example, you may receive records that should reference data from other tables, but those referenced records do not yet exist.

Suppose, for example, that you receive daily sales transactions records that reference product SKUs. When a product does not exist in the products table, the default behavior of the standard CKM is to reject the sales transaction record into the error table instead of loading it into the data warehouse. However, to meet the requirements of your project you wish to load this sales record into the data warehouse and create an empty product on the fly to ensure data consistency. The data analysts would then simply analyze the error tables and complete the missing information for products that were automatically added to the products table.

The following figure illustrates this example.

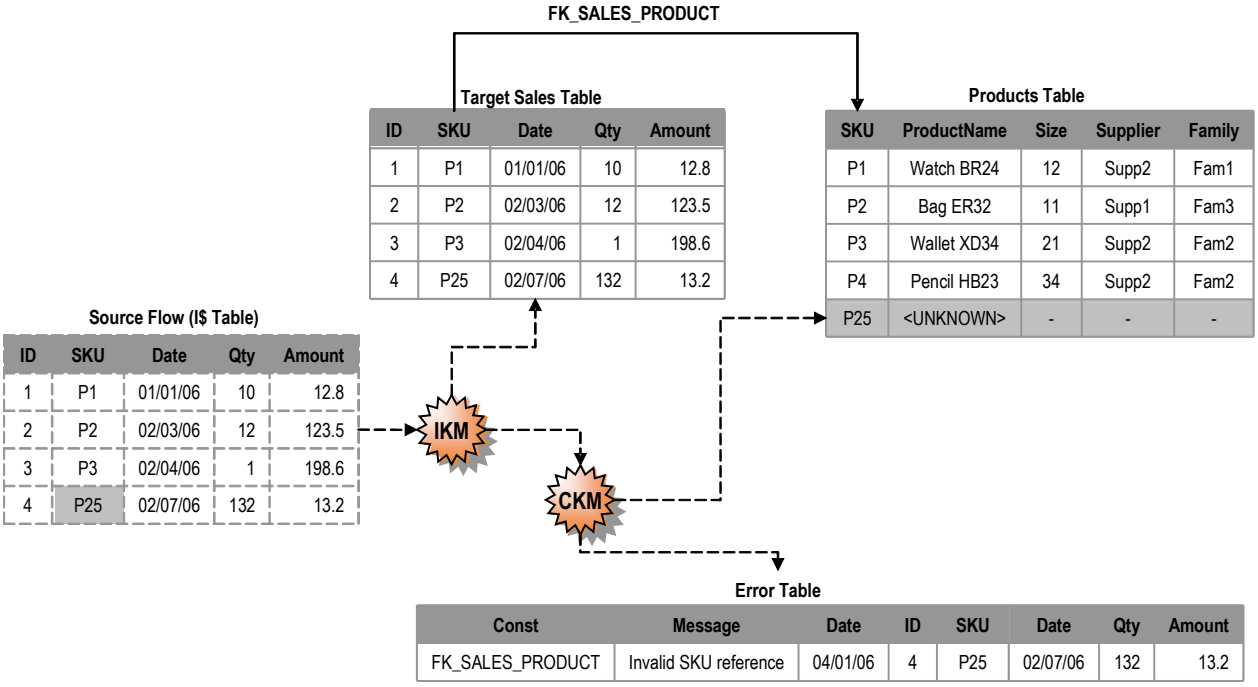

Figure 8: Creating References on the Fly

- The source flow data is staged by the IKM in the "I$" table. The IKM calls the CKM to have it check the data quality.
- The CKM checks every constraint including the FK_SALES_PRODUCT foreign key defined between the target Sales table and the Products Table. It rejects record ID 4 in the error table as product P25 doesn't exist in the products table.
- The CKM inserts the missing P25 reference in the products table and assigns an '<UNKNOWN>' value to the product name. All other columns are set to null or default values
- The CKM does not remove the rejected record from the source flow I$ table, as it became consistent
- The IKM writes the flow data to the target

To implement such a CKM, you will notice that some information is missing in the Oracle Data Integrator default metadata. For example, it could be useful to define for each foreign key, the name of the column of the referenced table that should hold the '<UNKNOWN>' value (ProductName in our case). As not all the foreign keys will behave the same, it could also be useful to have an indicator for every foreign key that explains whether this constraint needs to automatically create the missing reference or not. This additional information can be obtained simply by adding Flex Fields on the "Reference" object in the Oracle Data Integrator Security. The FK_SALES_PRODUCT constraint will allow you to enter this metadata as described in the figure below. For more information about Flex Fields, refer Oracle Data Integrator documentation.

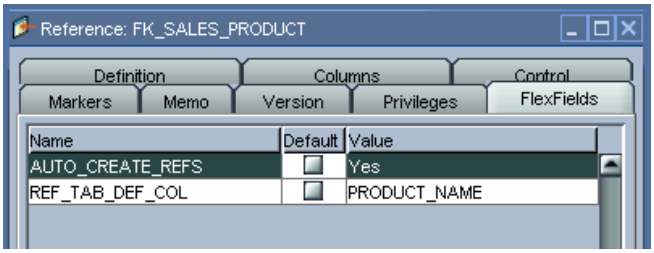

Figure 9: Adding Flex Fields to the FK_SALES_PRODUCT Foreign Key

Now that we have all the required metadata, we can start enhancing the default CKM to meet our requirements. The steps of the CKM will therefore be:

- Drop and create the summary table.
- Remove the summary records of the previous run from the summary table
- Drop and create the error table. **Add an extra column to the error table to store constraint behavior.**
- Remove rejected records from the previous run from the error table
- Reject records that violate the primary key constraint.
- Reject records that violate each alternate key constraint
- Reject records that violate each foreign key constraint
- **For every foreign key, if the AUTO_CREATE_REFS is set to "yes", insert missing references in the referenced table**
- Reject records that violate each check condition constraint
- Reject records that violate each mandatory column constraint
- Remove rejected records from the checked table if required. **Do not remove records for which the constraint behavior is set to Yes**
- Insert the summary of detected errors in the summary table.

Details of the implementation of such a CKM are listed below:

Step	Example of code for Teradata

Create the error table	`create multiset table <%=odiRef.getTable("L","ERR_NAME", "A")%>,`
	`no fallback,`
	`no before journal,`
	`no after journal`
	`(`
	`***AUTO_CREATE_REFS*** varchar(3),`
	`ERR_TYPE varchar(1) ,`
	`ERR_MESS varchar(250) ,`
	`CHECK_DATE timestamp ,`
	`<%=odiRef.getColList("", "[COL_NAME]\t[DEST_WRI_DT] " + odiRef.getInfo("DEST_DDL_NULL"), ",\n\t", "", "")%>,`
	`ORIGIN varchar(100) ,`
	`CONS_NAME varchar(35) ,`
	`CONS_TYPE varchar(2) ,`
	`)`

Isolate FK errors	`insert into <%=odiRef.getTable("L","ERR_NAME", "A")%>` `(`

Step	Example of code for Teradata
	AUTO_CREATE_REFS, ERR_TYPE, ERR_MESS, CHECK_DATE, ORIGIN, CONS_NAME, CONS_TYPE, <%=odiRef.getColList("", "[COL_NAME]", ",\n\t", "", "MAP")%>) select ***'<%=odiRef.getFK("AUTO_CREATE_REFS")%>'***, '<%=odiRef.getInfo("CT_ERR_TYPE")%>', '<%=odiRef.getFK("MESS")%>', <%=odiRef.getInfo("DEST_DATE_FCT")%>, '<%=odiRef.getInfo("CT_ORIGIN")%>', '<%=odiRef.getFK("FK_NAME")%>', 'FK', [... etc.]
Insert missing references	<% if (odiRef.getFK("AUTO_CREATE_REFS").equals("Yes")) { %> insert into <%=odiRef.getTable("L", "FK_PK_TABLE_NAME", "A")%> (<%=odiRef.getFKColList("", "[PK_COL_NAME]", ",", "")%> , <%=odiRef.getFK("REF_TAB_DEF_COL")%>) select distinct <%=odiRef.getFKColList("", "[COL_NAME]", ",", "")%> ,'<UNKNOWN>' from <%=odiRef.getTable("L","ERR_NAME", "A")%> where CONS_NAME = '<%=odiRef.getFK("FK_NAME")%>' And CONS_TYPE = 'FK' And ORIGIN = '<%=odiRef.getInfo("CT_ORIGIN")%>' And AUTO_CREATE_REFS = 'Yes' <%}%>
Remove the rejected records from the checked table	delete from <%=odiRef.getTable("L", "CT_NAME", "A")%> where exists (select 'X' from <%=odiRef.getTable("L","ERR_NAME", "A")%> as E where <%=odiRef.getColList("","(("+odiRef.getTable("L", "CT_NAME", "A")+".[COL_NAME]\t= E.[COL_NAME]) or ("+odiRef.getTable("L", "CT_NAME", "A")+".[COL_NAME] is null and E.[COL_NAME] is null))", "\n\t\tand\t", "", "UK")%> and E.AUTO_CREATE_REFS <> 'Yes') [...etc.]

Using the Agent

The Agent is able to read a result set using JDBC on a source server and write this result set using JDBC to the "C$" table of the target staging area server. To use this method, your Knowledge Module needs to include a SELECT/INSERT statement as described Oracle Data Integrator documentation. This method may not be suited for large volumes as data is read row-by-row in arrays, using the array fetch feature, and written row-by-row, using the batch update feature.

A typical LKM using this strategy contains the following steps:

Step	Example of code
Drop the "C$" table from the staging area. If the table doesn't exist, ignore the error.	`drop table <%=odiRef.getTable("L", "COLL_NAME", "A")%>`

Create the "C$" table in the staging area	`create table <%=odiRef.getTable("L", "COLL_NAME", "A")%>`
	`(`
	`<%=odiRef.getColList("", "[CX_COL_NAME]\t[DEST_WRI_DT] " + odiRef.getInfo("DEST_DDL_NULL"), ",\n\t", "","")%>`
	`)`

Load the source result set to the "C$" table using a SELECT/INSERT command.
The SELECT is executed on the source and the INSERT on the staging area. The agent performs data type translations in memory using the JDBC API.
	Code on the Source tab executed by the source server:
	`select <%=odiRef.getPop("DISTINCT_ROWS")%>`
	`<%=odiRef.getColList("", "[EXPRESSION]\t[ALIAS_SEP] [CX_COL_NAME]", ",\n\t", "", "")%>`
	`from <%=odiRef.getFrom()%>`
	`where (1=1)`
	`<%=odiRef.getFilter()%>`
	`<%=odiRef.getJrnFilter()%>`
	`<%=odiRef.getJoin()%>`
	`<%=odiRef.getGrpBy()%>`
	`<%=odiRef.getHaving()%>`
	
	Code on the Target tab executed in the staging area:
	`insert into <%=odiRef.getTable("L", "COLL_NAME", "A")%>`
	`(`
	`<%=odiRef.getColList("", "[CX_COL_NAME]", ",\n\t", "","")%>`
	`)`
	`values`
	`(`
	`<%=odiRef.getColList("", ":[CX_COL_NAME]", ",\n\t", "","")%>`
	`)`

After the IKM has finished	`drop table <%=odiRef.getTable("L", "COLL_NAME", "A")%>`

integration in the target, drop the "C$" table. This step can be made dependent on the value of an option to give the developer the option of keeping the "C$" table for debugging purposes.	

Using Loaders

Using Loaders for Flat Files

When your interface contains a flat file as a source, you may want to use a strategy that leverages the most efficient loading utility available for the staging area technology, rather than the standard "LKM File to SQL". Almost all RDBMS have a fast loading utility to load flat files into tables:

When working with Oracle we can use either SQL*LOADER or EXTERNAL TABLE.

Teradata suggests 3 different utilities: FastLoad for loading large files to empty tables, MultiLoad for complex loads of large files, including incremental loads and TPump for continuous loads of small files.

For LKMs, you simply need to load the file into the "C$" staging area. All transformations will be done by the IKM in the RDBMS. Therefore, a typical LKM using a loading utility will usually follow these steps:

- Drop and create the "C$" table in the staging area
- Generate the script required by the loading utility to load the file to the "C$" staging table.
- Execute the appropriate operating system command to start the load and check its return code.
- Possibly analyze any log files produced by the utility for error handling.
- Drop the "C$" table once the integration KM has terminated.

The following table gives you extracts from the "LKM File to Oracle (EXTERNAL TABLE)" that uses this strategy. Refer to the KM for the complete code:

Step	Example of code
Create Oracle directory	`create or replace directory dat_dir AS '<%=snpRef.getSrcTablesList("", "[SCHEMA]", "", "")%>'`

Create external table	`create table <%=snpRef.getTable("L", "COLL_NAME", "W")%>`
	`(`
	`<%=snpRef.getColList("", "[CX_COL_NAME]\t[DEST_WRI_DT]", ",\n\t", "","")%>`
	`)`
	`ORGANIZATION EXTERNAL`
	`(`
	`TYPE ORACLE_LOADER`
	`DEFAULT DIRECTORY dat_dir`
	`ACCESS PARAMETERS`
	`<% if (snpRef.getSrcTablesList("", "[FILE_FORMAT]", "", "").equals("F")) {%> (`

```
                RECORDS DELIMITED BY NEWLINE
                <%=snpRef.getUserExit("EXT_CHARACTERSET")%>
                <%=snpRef.getUserExit("EXT_STRING_SIZE")%>
                BADFILE        '<%=snpRef.getSrcTablesList("",
"[RES_NAME]", "", "")%>_%a.bad'
                LOGFILE        '<%=snpRef.getSrcTablesList("",
"[RES_NAME]", "", "")%>_%a.log'
                DISCARDFILE    '<%=snpRef.getSrcTablesList("",
"[RES_NAME]", "", "")%>_%a.dsc'
                SKIP           <%=snpRef.getSrcTablesList("",
"[FILE_FIRST_ROW]", "", "")%>
                FIELDS
                <%=snpRef.getUserExit("EXT_MISSING_FIELD")%>
                (
                        <%=snpRef.getColList("",
"[CX_COL_NAME]\tPOSITION([FILE_POS]\\:[FILE_END_POS])", ",\n\t\t\t",
"","")%>
                )
        )
<%} else {%>   (
                RECORDS DELIMITED BY NEWLINE
                <%=snpRef.getUserExit("EXT_CHARACTERSET")%>
                <%=snpRef.getUserExit("EXT_STRING_SIZE")%>
                BADFILE        '<%=snpRef.getSrcTablesList("",
"[RES_NAME]", "", "")%>_%a.bad'
                LOGFILE        '<%=snpRef.getSrcTablesList("",
"[RES_NAME]", "", "")%>_%a.log'
                DISCARDFILE    '<%=snpRef.getSrcTablesList("",
"[RES_NAME]", "", "")%>_%a.dsc'
                SKIP           <%=snpRef.getSrcTablesList("",
"[FILE_FIRST_ROW]", "", "")%>
                FIELDS  TERMINATED BY  '<%=snpRef.getSrcTablesList("",
"[SFILE_SEP_FIELD]", "", "")%>'
                <% if(snpRef.getSrcTablesList("", "[FILE_ENC_FIELD]", "",
"").equals("")){%>
                <%} else {%>OPTIONALLY ENCLOSED BY
'<%=snpRef.getSrcTablesList("", "[FILE_ENC_FIELD]", "",
"").substring(0,1)%>' AND '<%=snpRef.getSrcTablesList("",
"[FILE_ENC_FIELD]", "", "").substring(1,2)%>' <%}%>
                <%=snpRef.getUserExit("EXT_MISSING_FIELD")%>
                (
                        <%=snpRef.getColList("", "[CX_COL_NAME]",
",\n\t\t\t", "","")%>
                )
        )
<%}%>  LOCATION (<%=snpRef.getSrcTablesList("", "'[RES_NAME]'", "",
"")%>)
)
<%=snpRef.getUserExit("EXT_PARALLEL")%>
REJECT LIMIT <%=snpRef.getUserExit("EXT_REJECT_LIMIT")%>
NOLOGGING
```

Using Unload/Load for Remote Servers

When the source result set is on a remote database server, an alternative to using the agent to transfer the data would be to unload it to a file and then load that into the staging area. This is usually the most efficient method when dealing with large volumes. The steps of LKMs that follow this strategy are often as follows:

- Drop and create the "C$" table in the staging area
- Unload the data from the source to a temporary flat file using either a source unload utility (such as MSSQL bcp or DB2 unload) or the OdiSqlUnload tool.
- Generate the script required by the loading utility to load the temporary file to the "C$" staging table.
- Execute the appropriate operating system command to start the load and check its return code.
- Optionally, analyze any log files produced by the utility for error handling.
- Drop the "C$" table once the integration KM has terminated.

The "LKM SQL to Teradata (TPUMP-FASTLOAD-MULTILOAD)" follows these steps and uses the generic OdiSqlUnload tool to unload data from any remote RDBMS. Of course, this KM can be optimized if the source RDBMS is known to have a fast unload utility.

The following table shows some extracts of code from this LKM:

Step	Example of code
Unload data from source using OdiSqlUnload	`OdiSqlUnload "-DRIVER=<%=snpRef.getInfo("SRC_JAVA_DRIVER")%>" "-URL=<%=snpRef.getInfo("SRC_JAVA_URL")%>" "-USER=<%=snpRef.getInfo("SRC_USER_NAME")%>" "-PASS=<%=snpRef.getInfo("SRC_ENCODED_PASS")%>" "-FILE_FORMAT=variable" "-FIELD_SEP=<%=snpRef.getOption("FIELD_SEP")%>" "-FETCH_SIZE=<%=snpRef.getInfo("SRC_FETCH_ARRAY")%>" "-DATE_FORMAT=<%=snpRef.getOption("UNLOAD_DATE_FMT")%>" "-FILE=<%=snpRef.getOption("TEMP_DIR")%>/<%=snpRef.getTable("L", "COLL_NAME", "W")%>"`  `select <%=snpRef.getPop("DISTINCT_ROWS")%>` `<%=snpRef.getColList("", "\t[EXPRESSION]", ",\n", "", "")%>` `from <%=snpRef.getFrom()%>` `where (1=1)` `<%=snpRef.getJoin()%>` `<%=snpRef.getFilter()%>` `<%=snpRef.getJrnFilter() %>` `<%=snpRef.getGrpBy()%>` `<%=snpRef.getHaving()%>`

Oracle Data Integrator delivers the following Knowledge Modules that use this strategy:

Using Piped Unload/Load

When using an unload/load strategy, data needs to be staged twice: once in the temporary file and a second time in the "C$" table, resulting in extra disk space usage and potential efficiency issues. A more efficient alternative would be to use pipelines between the "unload" and the "load" utility. Unfortunately, not all the operating systems support file-based pipelines (FIFOs).

When the agent is installed on Unix, you can decide to use a piped unload/load strategy. The steps followed by your LKM would be:

- Drop and create the "C$" table in the staging area
- Create a pipeline file on the operating system (for example using the `mkfifo` command on Unix)
- Generate the script required by the loading utility to load the temporary file to the "C$" staging table.
- Execute the appropriate operating system command to start the load as a detached process (using "&" at the end of the command). The load starts and immediately waits for data in the FIFO.
- Start unloading the data from the source RDBMS to the FIFO using either a source unload utility (such as MSSQL bcp or DB2 unload) or the OdiSqlUnload tool.
- Join the load process and wait until it finishes. Check for processing errors.
- Optionally, analyze any log files produced by utilities for additional error handling.
- Drop the "C$" table once the integration KM has finished.

Oracle Data Integrator provides the "LKM SQL to Teradata (piped TPUMP-FAST-MULTILOAD)" that uses this strategy. To have a better control on the behavior of every detached process (or thread), this KM was written using Jython. The OdiSqlUnload tool is also available as a callable object in Jython. The following table gives extracts of code from this LKM. Refer to the actual KM for the complete code:

Step	Example of code

Jython function used to trigger the OdiSqlUnload command	`import com.sunopsis.dwg.tools.OdiSqlUnload as JOdiSqlUnload`
	`import java.util.Vector as JVector`
	`import java.lang.String`
	`from jarray import array`
	`...`
	`srcdriver = "<%=snpRef.getInfo("SRC_JAVA_DRIVER")%>"`
	`srcurl = "<%=snpRef.getInfo("SRC_JAVA_URL")%>"`
	`srcuser = "<%=snpRef.getInfo("SRC_USER_NAME")%>"`
	`srcpass = "<%=snpRef.getInfo("SRC_ENCODED_PASS")%>"`
	`fetchsize = "<%=snpRef.getInfo("SRC_FETCH_ARRAY")%>"`
	`...`
	
	`query = """select <%=snpRef.getPop("DISTINCT_ROWS")%>`
	`<%=snpRef.getColList("", "\t[EXPRESSION]", ",\n", "", "")%>`
	`from <%=snpRef.getFrom()%>`
	`where (1=1)`
	`<%=snpRef.getJoin()%>`
	`<%=snpRef.getFilter()%>`
	`<%=snpRef.getJrnFilter() %>`
	`<%=snpRef.getGrpBy()%>`
	`<%=snpRef.getHaving()%>`
	`"""`
	`...`
	`def odisqlunload():`
	`  odiunload = JOdiSqlUnload()`
	
	`  # Set the parameters`
	`  cmdline = JVector()`

Step	Example of code

	cmdline.add(array(["-DRIVER", srcdriver], java.lang.String))
	cmdline.add(array(["-URL", srcurl], java.lang.String))
	cmdline.add(array(["-USER", srcuser], java.lang.String))
	cmdline.add(array(["-PASS", srcpass], java.lang.String))
	cmdline.add(array(["-FILE_FORMAT", "variable"], java.lang.String))
	cmdline.add(array(["-FIELD_SEP", fieldsep], java.lang.String))
	cmdline.add(array(["-FETCH_SIZE", fetchsize], java.lang.String))
	cmdline.add(array(["-FILE", pipename], java.lang.String))
	cmdline.add(array(["-DATE_FORMAT", datefmt], java.lang.String))
	cmdline.add(array(["-QUERY", query], java.lang.String))
	
	odiunload.setParameters(cmdline)
	
	# Start the unload process
	odiunload.execute()

Main function that runs the piped load	...
	
	utility= "<%=snpRef.getOption("TERADATA UTILITY")%>"
	if utility == "multiload":
	utilitycmd="mload"
	else:
	utilitycmd=utility
	
	# when using Unix pipes, it is important to get the pid
	# command example : load < myfile.script > myfile.log & echo $! > mypid.txt ; wait $!
	# Note: the PID is stored in a file to be able to kill the fastload in case of crash
	
	loadcmd= '%s < %s > %s & echo $! > %s ; wait $!' % (utilitycmd,scriptname, logname, outname)
	
	...
	
	def pipedload():
	# Create or Replace a Unix FIFO
	os.system("rm %s" % pipename)
	if os.system("mkfifo %s" % pipename) <> 0:
	raise "mkfifo error", "Unable to create FIFO %s" % pipename
	
	# Start the load command in a dedicated thread
	loadthread = threading.Thread(target=os.system,
	args=(loadcmd,),
	name="snptdataload")

Step	Example of code

	loadthread.start()
	\# now that the fastload thead has started, wait
	\# 3 seconds to see if it is running
	time.sleep(3)
	if not loadthread.isAlive():
	os.system("rm %s" % pipename)
	raise "Load error", "(%s) load process not started" % loadcmd
	
	\# Start the SQLUnload process
	try:
	OdiSqlUnload()
	except:
	\# if the unload process fails, we have to kill
	\# the load process on the OS.
	\# Several methods are used to get sure the process is killed
	
	\# get the pid of the process
	f = open(outname, 'r')
	pid = f.readline().replace('\n', '').replace('\r', '')
	f.close()
	\# close the pipe by writing something fake in it
	os.system("echo dummy > %s" % pipename)
	\# attempt to kill the process
	os.system("kill %s" % pid)
	\# remove the pipe
	os.system("rm %s" % pipename)
	raise
	
	\# At this point, the unload() process has finished, so we need to wait
	\# for the load process to finish (join the thread)
	loadthread.join()

Using RDBMS-Specific Strategies

Some RDBMSs have a mechanism for sharing data across servers of the same technology. For example:

- Oracle has database links for loading data between 2 remote oracle servers
- Microsoft SQL Server has linked servers
- IBM DB2 400 has DRDA file transfer

IKMs with Staging Area on Target

Simple Replace or Append

The simplest strategy for integrating data in an existing target table, provided that all source data is already in the staging area is to replace and insert the records in the target. Therefore, the simplest IKM would be composed of 2 steps:

- Remove all records from the target table. This step can be made dependent on an option set by the designer of the interface
- Transform and insert source records from all source sets. When dealing with remote source data, LKMs will have already prepared "C$" tables with pre-transformed result sets. If the interface uses source data sets on the same server as the target (and the staging area as well), they will be joined to the other "C$" tables. Therefore the integration operation will be a straight INSERT/SELECT statement leveraging all the transformation power of the target Teradata box.

The following example gives you the details of these steps:

Step	Example of code
Remove data from target table. This step can be made dependent on a "check box" option: "Delete all rows?"	`delete from <%=odiRef.getTable("L","INT_NAME","A")%>`

Append the flow records to the target	`insert into <%=odiRef.getTable("L","INT_NAME","A")%>`
	`(`
	`<%=odiRef.getColList("", "[COL_NAME]", ",\n\t", "", "((INS and !TRG) and REW)")%>`
	`)`
	`select <%=odiRef.getPop("DISTINCT_ROWS")%>`
	`<%=odiRef.getColList("", "[EXPRESSION]", ",\n\t", "", "((INS and !TRG) and REW)")%>`
	`from <%=odiRef.getFrom()%>`
	`where (1=1)`
	`<%=odiRef.getJoin()%>`
	`<%=odiRef.getFilter()%>`
	`<%=odiRef.getJrnFilter()%>`
	`<%=odiRef.getGrpBy()%>`
	`<%=odiRef.getHaving()%>`

This very simple strategy is not provided as is in the default Oracle Data Integrator KMs. It can be obtained as a special case of the "Control Append" IKMs when choosing not to control the flow data.

The next paragraph further discusses these types of KMs.

Append with Data Quality Check

In the preceding example, flow data was simply inserted in the target table without any data quality checking. This approach can be improved by adding extra steps that will store the flow data in a temporary table called the integration table ("I$") before calling the CKM to isolate erroneous records in the error table ("E$"). The steps of such an IKM could be:

- Drop and create the flow table in the staging area. The "I$" table is created with the same columns as the target table so that it can be passed to the CKM for data quality check.
- Insert flow data in the "I$" table. Source records from all source sets are transformed and inserted in the "I$" table in a single INSERT/SELECT statement.
- Call the CKM for the data quality check. The CKM will simulate every constraint defined for the target table on the flow data. It will create the error table and insert the erroneous records. It will also remove all erroneous records from the controlled table. Therefore, after the CKM completes, the "I$" table will only contain valid records. Inserting them in the target table can then be done safely.
- Remove all records from the target table. This step can be made dependent on an option value set by the designer of the interface
- Append the records from the "I$" table to the target table in a single "inset/select" statement.
- Drop the temporary "I$" table.

In some cases, it may also be useful to recycle previous errors so that they are added to the flow and applied again to the target. This method can be useful for example when receiving daily sales transactions that reference product IDs that may not exist. Suppose that a sales record is rejected in the error table because the referenced product ID does not exist in the product table. This happens during the first run of the interface. In the meantime the missing product ID is created by the data administrator. Therefore the rejected record becomes valid and should be re-applied to the target during the next execution of the interface.

This mechanism is fairly easy to implement in the IKM by simply adding an extra step that would insert all the rejected records of the previous run into the flow table ("I$") prior to calling the CKM to check the data quality.

This IKM can also be enhanced to support a simple replace-append strategy. The flow control steps would become optional. The data would be applied to the target either from the "I$" table, if the designer chose to check the data quality, or from the source sets, e.g. staging "C$" tables.

Some of the steps of such an IKM are described below:

Step	Example of code	Execute if

Create the flow table in the staging area	`create table <%=odiRef.getTable("L", "INT_NAME", "A")%>`
	`(`
	`<%=odiRef.getColList("", "[COL_NAME]\t[DEST_WRI_DT] " + odiRef.getInfo("DEST_DDL_NULL"), ",\n\t", "", "INS")%>`
	`)`
		FLOW_CONTROL is set to YES

Insert flow data in the "I$" table	`insert into <%=odiRef.getTable("L","INT_NAME","A")%>`
	`(`
	`<%=odiRef.getColList("", "[COL_NAME]",`
		FLOW_CONTROL is set to YES

Step	Example of code	Execute if
	",\n\t", "", "((INS and !TRG) and REW)")%>) select <%=odiRef.getPop("DISTINCT_ROWS")%> <%=odiRef.getColList("", "[EXPRESSION]", ",\n\t", "", "((INS and !TRG) and REW)")%> from <%=odiRef.getFrom()%> where (1=1) <%=odiRef.getJoin()%> <%=odiRef.getFilter()%> <%=odiRef.getJrnFilter()%> <%=odiRef.getGrpBy()%> <%=odiRef.getHaving()%>	

Recycle previous rejected records	insert into <%=odiRef.getTable("L","INT_NAME","A")%>
	(
	  <%=odiRef.getColList("", "[COL_NAME]", ",\n\t", "", "INS and REW")%>
	)
	select <%=odiRef.getColList("", "[COL_NAME]", ",\n\t", "", "INS and REW")%>
	from <%=odiRef.getTable("L","ERR_NAME", "A")%> <%=odiRef.getInfo("DEST_TAB_ALIAS_WORD")%> E
	where not exists (
	  select 'X'
	  from <%=odiRef.getTable("L","INT_NAME","A")%> <%=odiRef.getInfo("DEST_TAB_ALIAS_WORD")%> T
	  where <%=odiRef.getColList("", "T.[COL_NAME]\t= E.[COL_NAME]", "\n\t\tand\t", "", "UK")%>
	  )
	and E.ORIGIN = '<%=odiRef.getInfo("CT_ORIGIN")%>'
	and E.ERR_TYPE = '<%=odiRef.getInfo("CT_ERR_TYPE")%>'
		RECYCLE_ERRORS is set to Yes

Call the CKM to perform data quality check	<%@ INCLUDE CKM_FLOW DELETE_ERRORS%>	FLOW_CONTROL is set to YES
Remove all records from the target table	delete from <%=odiRef.getTable("L","TARG_NAME","A")%>	DELETE_ALL is set to Yes
Insert records. If flow control is set to Yes, then the data will be inserted from the "I$" table. Otherwise it will be inserted from the source sets.	<%if (odiRef.getOption("FLOW_CONTROL").equals("1")) { %> insert into <%=odiRef.getTable("L","TARG_NAME","A")%> (<%=odiRef.getColList("", "[COL_NAME]", ",\n\t", "", "((INS and !TRG) and REW)")%> <%=odiRef.getColList(",", "[COL_NAME]",	INSERT is set to Yes

Step	Example of code	Execute if
	",\n\t", "", "((INS and TRG) and REW)")%>) select <%=odiRef.getColList("", "[COL_NAME]", ",\n\t", "", "((INS and !TRG) and REW)")%> <%=odiRef.getColList(",", "[EXPRESSION]", ",\n\t", "", "((INS and TRG) and REW)")%> from <%=odiRef.getTable("L","INT_NAME","A")%> <% } else { %> insert into <%=odiRef.getTable("L","TARG_NAME","A")%> (<%=odiRef.getColList("", "[COL_NAME]", ",\n\t", "", "(INS and REW)")%>) select <%=odiRef.getPop("DISTINCT_ROWS")%> <%=odiRef.getColList("", "[EXPRESSION]", ",\n\t", "", "(INS and REW)")%> from <%=odiRef.getFrom()%> where <% if (odiRef.getPop("HAS_JRN").equals("0")) { %> (1=1) <%} else {%> JRN_FLAG <> 'D' <% } %> <%=odiRef.getJoin()%> <%=odiRef.getFilter()%> <%=odiRef.getJrnFilter()%> <%=odiRef.getGrpBy()%> <%=odiRef.getHaving()%> <% } %>	

Incremental Update

The Incremental Update strategy is used to integrate data in the target table by comparing the records of the flow with existing records in the target according to a set of columns called the "update key". Records that have the same update key are updated when their associated data is not the same. Those that don't yet exist in the target are inserted. This strategy is often used for dimension tables when there is no need to keep track of the records that have changed.

The challenge with such IKMs is to use set-oriented SQL based programming to perform all operations rather than using a row-by-row approach that often leads to performance issues. The most common method to build such strategies often relies on a temporary integration table ("I$") which stores the transformed source sets. This method is described below:

- Drop and create the flow table in the staging area. The "I$" table is created with the same columns as the target table so that it can be passed to the CKM for the data quality check. It also contains an IND_UPDATE column that is used to flag the records that should be inserted ("I") and those that should be updated ("U").
- Insert flow data in the "I$" table. Source records from all source sets are transformed and inserted in the "I$" table in a single INSERT/SELECT statement. The IND_UPDATE column is set by

default to "I".

- Add the rejected records from the previous run to the "I$" table if the designer chooses to recycle errors.
- Call the CKM for the data quality check. The CKM simulates every constraint defined for the target table on the flow data. It creates an error table and inserts any erroneous records. It also removes all erroneous records from the checked table. Therefore, after the CKM completes, the "I$" table will only contain valid records.
- Update the "I$" table to set the IND_UPDATE column to "U" for all the records that have the same update key values as the target ones. Therefore, records that already exist in the target will have a "U" flag. This step is usually an UPDATE/SELECT statement
- Update the "I$" table again to set the IND_UPDATE column to "N" for all records that are already flagged as "U" and for which the column values are exactly the same as the target ones. As these flow records match exactly the target records, they don't need to be used to update the target data. After this step, the "I$" table is ready for applying the changes to the target as it contains records that are flagged:
    - "I": these records should be inserted into the target
    - "U": these records should be used to update the target
    - "N": these records already exist in the target and should be ignored
- Update the target with records from the "I$" table that are flagged "U". Note that the update statement should be executed prior to the INSERT statement to minimize the volume of data manipulated.
- Insert records in the "I$" table that are flagged "I" into the target
- Drop the temporary "I$" table.

Of course, this approach can be optimized depending on the underlying database. For example, in Teradata, it may be more efficient in some cases to use a left outer join between the flow data and the target table to populate the "I$" table with the IND_UPDATE column already set properly.

Note:
The update key should always be unique. In most cases, the primary key will be used as an update key. The primary key cannot be used, however, when it is automatically calculated using an increment such as an identity column, a rank function, or a sequence. In this case an update key based on columns present in the source must be used.

Some of the steps of such an IKM are described below:

Step	Example of code	Execute if

Create the flow table in the staging area	`create <%=odiRef.getOption("FLOW_TABLE_TYPE")%> table <%=odiRef.getTable("L", "INT_NAME", "A")%>,`
	`(`
	`<%=odiRef.getColList("", "[COL_NAME]\t[DEST_WRI_DT] " + odiRef.getInfo("DEST_DDL_NULL"), ",\n\t", "", "")%>,`
	`IND_UPDATE char(1)`
	`)`	

Determine what to update (using the update key)	`update <%=odiRef.getTable("L", "INT_NAME", "A")%>` `from <%=odiRef.getTable("L", "TARG_NAME", "A")%> T`	INSERT or UPDATE are set to Yes

Step	Example of code	Execute if
	`set IND_UPDATE = 'U'` `where <%=odiRef.getColList("", odiRef.getTable("L", "INT_NAME", "A") + ".[COL_NAME]\t= T.[COL_NAME]", "\nand\t", "", "UK")%>`	

Determine what shouldn't be updated by comparing the data	`update <%=odiRef.getTable("L", "INT_NAME", "A")%>`
	`from <%=odiRef.getTable("L", "TARG_NAME", "A")%> T`
	`set IND_UPDATE = 'N'`
	`where <%=odiRef.getColList("", odiRef.getTable("L", "INT_NAME", "A") + ".[COL_NAME]\t= T.[COL_NAME]", "\nand\t", "", "UK")%>`
	`and <%=odiRef.getColList("","((" + odiRef.getTable("L", "INT_NAME", "A") + ".[COL_NAME] = T.[COL_NAME]) or (" + odiRef.getTable("L", "INT_NAME", "A") + ".[COL_NAME] IS NULL and T.[COL_NAME] IS NULL))", " \nand\t", "", "((UPD and !TRG) and !UK) ")%>`
		UPDATE is set to Yes

Update the target with the existing records	`update <%=odiRef.getTable("L", "TARG_NAME", "A")%>`
	`from <%=odiRef.getTable("L", "INT_NAME", "A")%> S`
	`set <%=odiRef.getColList("", "[COL_NAME]\t= S.[COL_NAME]", ",\n\t", "", "(((UPD and !UK) and ! TRG) and REW)")%>`
	`<%=odiRef.getColList(",", "[COL_NAME]=[EXPRESSION]", ",\n\t", "", "(((UPD and !UK) and TRG) and REW)")%>`
	`where <%=odiRef.getColList("", odiRef.getTable("L", "TARG_NAME", "A") + ".[COL_NAME]\t= S.[COL_NAME]", "\nand\t", "", "(UK)")%>`
	`and S.IND_UPDATE = 'U'`
		UPDATE is set to Yes

Insert new records	`insert into <%=odiRef.getTable("L","TARG_NAME","A")%>`
	`(`
	`<%=odiRef.getColList("", "[COL_NAME]", ",\n\t", "", "((INS and !TRG) and REW)")%>`
	`<%=odiRef.getColList(",", "[COL_NAME]", ",\n\t", "", "((INS and TRG) and REW)")%>`
	`)`
	`select <%=odiRef.getColList("", "[COL_NAME]", ",\n\t", "", "((INS and !TRG) and REW)")%>`
	`<%=odiRef.getColList(",", "[EXPRESSION]", ",\n\t", "", "((INS and TRG) and REW)")%>`
	`from <%=odiRef.getTable("L","INT_NAME","A")%>`
	`where IND_UPDATE = 'I'`
		INSERT is set to Yes

When comparing data values to determine what should not be updated, the join between the "I$" table and the target table is expressed on each column as follow:

```
Target.ColumnN = I$.ColumnN or (Target.ColumnN is null and I$.ColumnN is null)
```

This is done to allow comparison between null values, so that a null value matches another null value. A more elegant way of writing it would be to use the coalesce function. Therefore the WHERE predicate could be written this way:

```
<%=odiRef.getColList("","coalesce(" + odiRef.getTable("L", "INT_NAME", "A") + ".[COL_NAME], 0) =
coalesce(T.[COL_NAME], 0)", " \nand\t", "", "((UPD and !TRG) and !UK) ")%>
```

Notes:
Columns updated by the UPDATE statement are not the same as the ones used in the INSERT statement. The UPDATE statement uses selector "UPD and not UK" to filter only mappings marked as "Update" in the interface and that do not belong to the update key. The INSERT statement uses selector "INS" to retrieve mappings marked as "insert" in the interface.

It is important that the UPDATE statement and the INSERT statement for the target belong to the same transaction (Transaction 1). Should any of them fail, no data will be inserted or updated in the target.

Slowly Changing Dimensions

Type 2 Slowly Changing Dimension is one of the most well known data warehouse loading strategies. It is often used for loading dimension tables, in order to keep track of changes that occurred on some of the columns. A typical slowly changing dimension table would contain the flowing columns:

- A surrogate key calculated automatically. This is usually a numeric column containing an auto-number such as an identity column, a rank function or a sequence.
- A natural key. List of columns that represent the actual primary key of the operational system.
- Columns that may be overwritten on change
- Columns that require the creation of a new record on change
- A start date column indicating when the record was created in the data warehouse
- An end date column indicating when the record became obsolete (closing date)
- A current record flag indicating whether the record is the actual one (1) or an old one (0)

The figure below gives an example of the behavior of the product slowly changing dimension. In the operational system, a product is defined by its ID that acts as a primary key. Every product has a name, a size, a supplier and a family. In the Data Warehouse, we want to store a new version of this product whenever the supplier or the family is updated in the operational system.

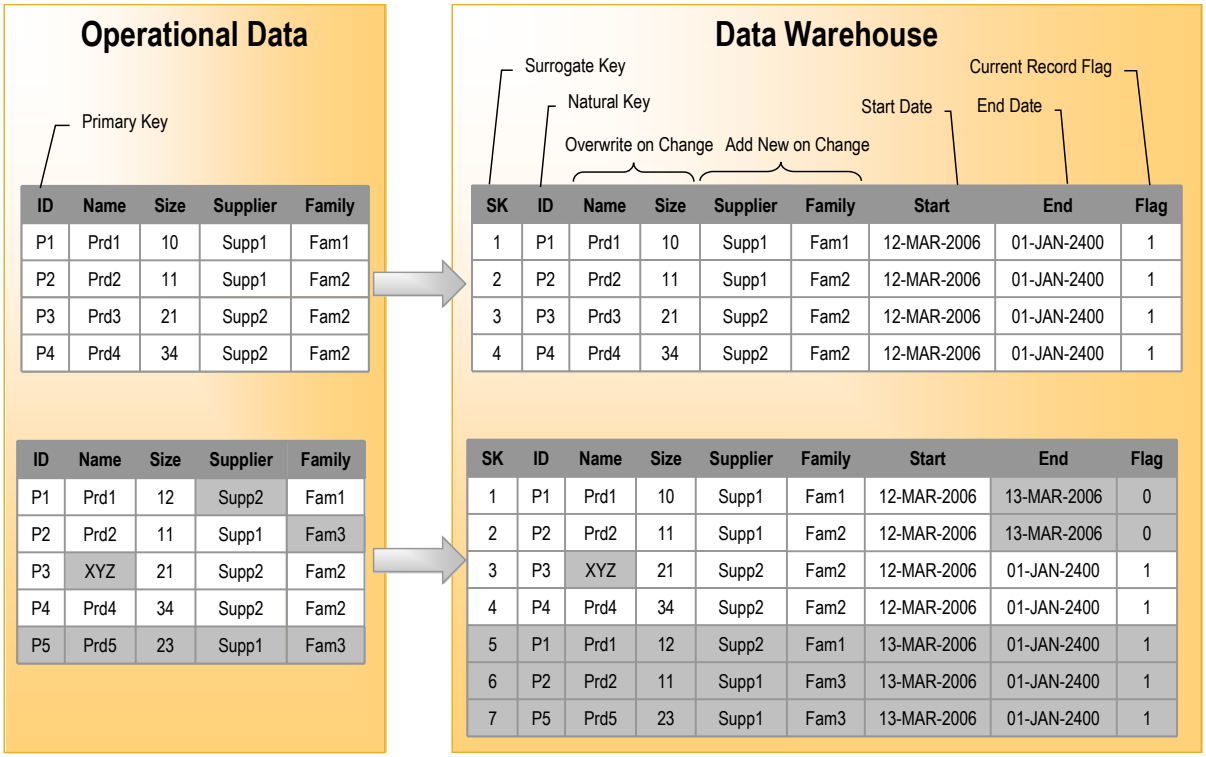

Figure 10: Slowly Changing Dimension Example

In this example, the product dimension is first initialized in the Data Warehouse on March 12, 2006. All the records are inserted and are assigned a calculated surrogate key as well as a fake ending date set to January 1, 2400. As these records represent the current state of the operational system, their current record flag is set to 1. After the first load, the following changes happen in the operational system:

1. The supplier is updated for product P1
2. The family is updated for product P2
3. The name is updated for product P3
4. Product P5 is added

These updates have the following impact on the data warehouse dimension:

1. The update of the supplier of P1 is translated into the creation of a new current record (Surrogate Key 5) and the closing of the previous record (Surrogate Key 1)
2. The update of the family of P2 is translated into the creation of a new current record (Surrogate Key 6) and the closing of the previous record (Surrogate Key 2)
3. The update of the name of P3 simply updates the target record with Surrogate Key 3
4. The new product P5 is translated into the creation of a new current record (Surrogate Key 7).

To create a Knowledge Module that implements this behavior, you need to know which columns act as a surrogate key, a natural key, a start date etc. Oracle Data Integrator can set this information in additional metadata fields for every column of the target slowly changing dimension datastore as described in the figure below.

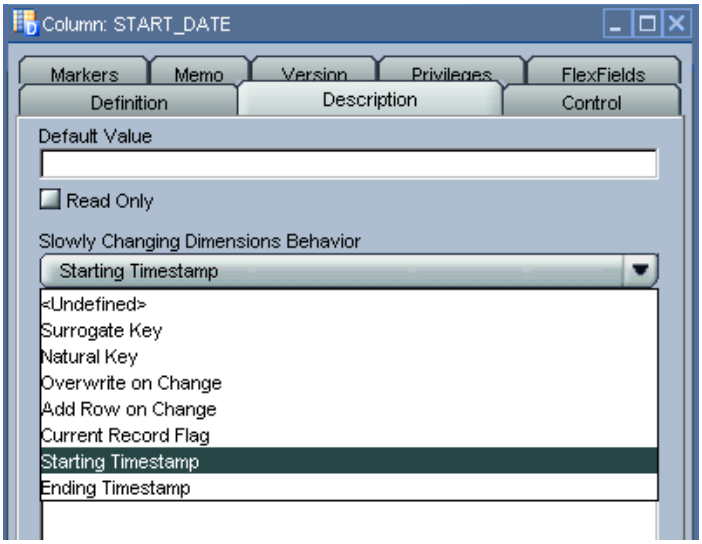

Figure 11: Slowly Changing Dimension Column Behavior

When populating such a datastore in an interface, the IKM has access to this metadata using the SCD_xx selectors on the getColList() substitution method.

The way Oracle Data Integrator implements Type 2 Slowly Changing Dimensions is described below:

- Drop and create the "I$" flow table to hold the flow data from the different source sets.
- Insert the flow data in the "I$" table using only mappings that apply to the "natural key", "overwrite on change" and "add row on change" columns. Set the start date to the current date and the end date to a constant.
- Recycle previous rejected records
- Call the CKM to perform a data quality check on the flow
- Flag the records in the "I$" table to 'U' when the "natural key" and the "add row on change" columns have not changed compared to the current records of the target
- Update the target with the columns that can be "overwritten on change" by using the "I$" flow filtered on the 'U' flag.
- Close old records – those for which the natural key exists in the "I$" table, and set their current record flag to 0 and their end date to the current date
- Insert the new changing records with their current record flag set to 1
- Drop the "I$" temporary table

Again, this approach can be adapted to your project's specific needs. There may be some cases where the SQL produced requires further tuning and optimization.

Some of the steps of the Teradata Slowly Changing Dimension IKM are listed below:

Step	Example of code

Insert flow data in the I$ table using an MINUS statement	`insert /*+ APPEND */ into <%=snpRef.getTable("L","INT_NAME","W")%>`
	`(`
	`  <%=snpRef.getColList("", "[COL_NAME]", ",\n\t", "", "(((INS OR UPD) AND NOT TRG) AND REW)")%>,`
	`  IND_UPDATE`
	`)`
	`select  <%=snpRef.getUserExit("OPTIMIZER_HINT")%>`

Step	Example of code

	<%=snpRef.getPop("DISTINCT_ROWS")%>
	  <%=snpRef.getColList("", "[EXPRESSION]", ",\n\t", "", "(((INS OR UPD) AND NOT TRG) AND REW)")%>,
	<%if (snpRef.getPop("HAS_JRN").equals("0")) {%>
	  'I' IND_UPDATE
	<%}else{%>
	  JRN_FLAG
	<%}%>
	from <%=snpRef.getFrom()%>
	where (1=1)
	<%=snpRef.getJoin()%>
	<%=snpRef.getFilter()%>
	<%=snpRef.getJrnFilter()%>
	<%=snpRef.getGrpBy()%>
	<%=snpRef.getHaving()%>
	minus
	select
	  <%=snpRef.getColList("","[COL_NAME]", ",\n\t", "", "(((INS OR UPD) AND NOT TRG) AND REW)")%>,
	  'I' IND_UPDATE
	from <%=snpRef.getTable("L", "TARG_NAME", "A")%>

Flag records that require an update on the target	update <%=snpRef.getTable("L", "INT_NAME", "W")%>
	set IND_UPDATE = 'U'
	where (<%=snpRef.getColList("","[COL_NAME]", ", ", "", "UK")%>)
	  in (
	  select <%=snpRef.getColList("","[COL_NAME]", ",\n\t\t\t", "", "UK")%>
	  from <%=snpRef.getTable("L", "TARG_NAME", "A")%>
	  )

Update the updatable columns on the target	update <%=snpRef.getTable("L", "TARG_NAME", "A")%> T
	set (
	  <%=snpRef.getColList("", "T.[COL_NAME]", ",\n\t", "", "((UPD AND (NOT UK) AND (NOT TRG)) AND REW)")%>
	  <%=snpRef.getColList(",", "T.[COL_NAME]", ",\n\t", "", "((UPD AND (NOT UK) AND TRG) AND REW)")%>
	  ) =
	  (
	  select <%=snpRef.getColList("", "S.[COL_NAME]", ",\n\t\t\t", "", "((UPD AND (NOT UK) AND (NOT TRG)) AND REW)")%>
	  <%=snpRef.getColList(",", "[EXPRESSION]", ",\n\t\t\t", "", "((UPD AND (NOT UK) AND TRG) AND REW)")%>
	  from <%=snpRef.getTable("L", "INT_NAME", "W")%> S
	  where <%=snpRef.getColList("", "T.[COL_NAME] =S.[COL_NAME]", "\n\t\tand\t", "", "UK")%>
	  )
	where (<%=snpRef.getColList("", "[COL_NAME]", ", ", "", "UK")%>)
	  in (
	  select <%=snpRef.getColList("", "[COL_NAME]",

Step	Example of code
	",\n\t\t\t", "", "UK")%> from <%=snpRef.getTable("L", "INT_NAME", "W")%> where IND_UPDATE = 'U')
Insert new records	insert into <%=snpRef.getTable("L","TARG_NAME","A")%> (<%=snpRef.getColList("", "[COL_NAME]", ",\n\t", "", "((INS AND (NOT TRG)) AND REW)")%> <%=snpRef.getColList(",", "[COL_NAME]", ",\n\t", "", "((INS AND TRG) AND REW)")%>) select <%=snpRef.getColList("", "[COL_NAME]", ",\n\t", "", "((INS AND (NOT TRG)) AND REW)")%> <%=snpRef.getColList(",", "[EXPRESSION]", ",\n\t", "", "((INS AND TRG) AND REW)")%> from <%=snpRef.getTable("L","INT_NAME","W")%> where IND_UPDATE = 'I'

Case Study: Backup Target Table before Load

Suppose that one of your project's requirements is to backup every data warehouse table prior to loading the current data. This requirement could, for example, help restore the data warehouse to its previous state in case of a major problem.

A first solution to this requirement would be to develop interfaces that would duplicate data from every target datastore to its corresponding backup one. These interfaces would be triggered prior to the ones that would populate the data warehouse. Unfortunately, this solution would lead to significant development and maintenance effort as it requires the creation of an additional interface for every target datastore. The number of interfaces to develop and maintain would be at least doubled!

A more elegant solution would be to implement this behavior in the IKM used to populate the target datastores. This would be done using a single INSERT/SELECT statement that writes to the backup table right before the steps that write to the target. Therefore, the backup of the data would become automatic and the developers of the interfaces would no longer need to worry about it.

This example shows how this behavior could be implemented in the IKM Incremental Update:

- Drop and create the "I$" flow table in the staging area.
- Insert flow data in the "I$" table.
- Recycle previous rejected records.
- Call the CKM for data quality check.
- Update the "I$" table to set the IND_UPDATE column to "U".
- Update the "I$" table again to set the IND_UPDATE column to "N".
- **Backup target table before load.**
- Update the target with the records of the "I$" table that are flagged "U".
- Insert into the target the records of the "I$" table that are flagged "I"
- Drop the temporary "I$" table.

Assuming that the name of the backup table is the same as the target table followed by "_BCK", the code of the backup step could be expressed as follows:

Step	Example of code
Drop the backup table	`Drop table <%=odiRef.getTable("L","TARG_NAME","A")%>_BCK`

Create and populate the backup table	`Create table <%=odiRef.getTable("L","TARG_NAME","A")%>_BCK as`
	`select <%=odiRef.getTargetColList("", "[COL_NAME]", ",", "")%>`
	`from <%=odiRef.getTable("L","TARG_NAME","A")%>`

Case Study: Tracking Records for Regulatory Compliance

Some data warehousing projects could require keeping track of every insert or update operation done to target tables for regulatory compliance. This could help business analysts understand what happened to their data during a certain period of time.

Even if you can achieve this behavior by using the slowly changing dimension Knowledge Modules, it can also be done by simply creating a copy of the flow data before applying it to the target table.

Suppose that every target table has a corresponding table with the same columns and additional regulatory compliance columns such as:

- The Job Id
- The Job Name
- Date and time of the operation
- The type of operation ("Insert" or "Update")

You would then populate this table directly from the "I$" table after applying the inserts and updates to the target, and right before the end of the IKM. For example, in the case of the Incremental Update IKM, your steps would be:

- Drop and create the "I$" flow table in the staging area.
- Insert flow data in the "I$" table.
- Recycle previous rejected records.
- Call the CKM for data quality check.
- Update the "I$" table to set the IND_UPDATE column to "U" or "N".
- Update the target with records from the "I$" table that are flagged "U".
- Insert into the target records from the "I$" table that are flagged "I"
- **Backup the I$ table for regulatory compliance**
- Drop the temporary "I$" table.

Assuming that the name of the regulatory compliance table is the same as the target table followed by "_RGC", the code for this step could be expressed as follows:

Step	Example of code
Backup the I$	`insert  into <%=odiRef.getTable("L","TARG_NAME","A")%>_RGC`

Step	Example of code
table for regulatory compliance	(JOBID, JOBNAME, OPERATIONDATE, OPERATIONTYPE, <%=odiRef.getColList("", "[COL_NAME]", ",\n\t", "")%>) select <%=odiRef.getSession("SESS_NO")%> /* JOBID */, <%=odiRef.getSession("SESS_NAME")%> /* JOBNAME */, Current_timestamp /* OPERATIONDATE */, Case when IND_UPDATE = 'I' then 'Insert' else 'Update' end <%=odiRef.getColList("", "[COL_NAME]", ",\n\t", "")%> from <%=odiRef.getTable("L","INT_NAME","A")%> where IND_UPDATE <> 'N'

This example demonstrates how easy and flexible it is to adapt existing Knowledge Modules to have them match even complex requirements, with a very low cost of implementation.

IKMs with Staging Area Different from Target

File to Server Append

There are some cases when your source is composed of a single file that you want to load directly into the target table using the most efficient method. By default, Oracle Data Integrator will suggest putting the staging area on the target server and performing such a job using an LKM to stage the file in a "C$" table and an IKM to apply the source data of the "C$" table to the target table. Obviously, if your source data is not transformed, you don't need to have the file loaded in the staging "C$" table before being loaded to the target.

A way of addressing this issue would be to use an IKM that can directly load the file data to the target. This requires setting the staging area to the source file logical schema. By doing so, Oracle Data Integrator will automatically suggest to use a "Multi-Connection" IKM that knows how to move data between a remote staging area and the target.

An IKM from a File to a target table using a loader would have the following steps:

- Generate the appropriate load utility script
- Run the load utility

Server to Server Append

When using a staging area different from the target and when setting this staging area to an RDBMS, you can use an IKM that will move the transformed data from the staging area to the remote target. This kind of IKM is very close to an LKM and follows almost the same rules.

Some IKMs use the agent to capture data from the staging area using arrays and write it to the target using batch updates. Others unload from the staging area to a file or FIFO and load the target using bulk load utilities.

The steps when using the agent are usually straightforward:

- Delete target data made dependent on the value of an option
- Insert the data from the staging area to the target. This step has a SELECT statement in the "Command on Source" tab that will be executed on the staging area. The INSERT statement is written using bind variables in the "Command on Target" tab and will be executed for every batch on the target table.

The steps when using an unload/load strategy usually depend on the type of IKM you choose. However most of them will have these general steps:

- Use OdiSqlUnload to unload data from the staging area to a file or FIFO pipeline.
- Generate the load utility script
- Call the load utility

Server to File or JMS Append

When the target datastore is a file or JMS queue or topic you will need to set the staging area to a different place than the target. Therefore, if you want to target a file or queue datastore you will have to use a "Multi-Connection" IKM that will export the transformed data from your staging area to this target. The way that the data is exported to the file or queue will depend on the IKM. For example, you can choose to use the agent to have it select records from the staging area and write them to the file or queue using standard Oracle Data Integrator features. Or you can use specific unload utilities such as Teradata FastExport if the target is not JMS based.

Typical steps of such an IKM might be:

- Reset the target file or queue made dependent on an option
- Unload the data from the staging area to the file or queue

Guidelines for Developing your own Knowledge Module

One of the main guidelines when developing your own KM is to never start from scratch. Oracle Data Integrator provides more than 100 KMs out-of-the-box. It is therefore recommended that you have a look at these existing KMs, even if they are not written for your technology. The more examples you have, the faster you develop your own code. You can, for example, duplicate an existing KM and start enhancing it by changing its technology, or copying lines of code from another one.

When developing your own KM, keep in mind that it is targeted to a particular stage of the integration process. As a reminder,

- LKMs are designed to load remote source data sets to the staging area (into "C$" tables)
- IKMs apply the source flow from the staging area to the target. They start from the "C$" tables, may transform and join them into a single "I$" table, may call a CKM to perform data quality checks on this "I$" table, and finally write the flow data to the target
- CKMs check data quality in a datastore or a flow table ("I$") against data quality rules expressed as constraints. The rejected records are stored in the error table ("E$")
- RKMs are in charge of extracting metadata from a metadata provider to the Oracle Data Integrator repository by using the SNP_REV_xx temporary tables.
- JKMs are in charge of creating the Change Data Capture infrastructure.

Be aware of these common pitfalls:

- Too many KMs: A typical project requires less than 5 KMs!
- Using hard-coded values including catalog or schema names in KMs: You should instead use the substitution methods getTable(), getTargetTable(), getObjectName() or others as appropriate.
- Using variables in KMs: You should instead use options or flex fields to gather information from the designer.
- Writing the KM completely in Jython or Java: You should do that if it is the only solution. SQL is often easier to read and maintain.
- Using <%if%> statements rather than a check box option to make code generation conditional.

Other common code writing recommendations that apply to KMs:

- The code should be correctly indented
- The generated code should also be indented in order to be readable
- SQL keywords such as "select", "insert", etc. should be in lowercase for better readability