

Oracle Data Integrator Best Practices for a Data Warehouse

Oracle Best Practices
March 2008

Oracle Data Integrator Best Practices for a Data Warehouse

PREFACE	7
PURPOSE	7
AUDIENCE	7
ADDITIONAL INFORMATION	7
INTRODUCTION TO ORACLE DATA INTEGRATOR (ODI)	8
OBJECTIVES	8
BUSINESS-RULES DRIVEN APPROACH.....	8
<i>Introduction to Business rules</i>	8
<i>Mappings</i>	9
<i>Joins</i>	9
<i>Filters</i>	9
<i>Constraints</i>	9
TRADITIONAL ETL VERSUS E-LT APPROACH	9
UNDERSTANDING ORACLE DATA INTEGRATOR (ODI) INTERFACES	10
A BUSINESS PROBLEM CASE STUDY	11
IMPLEMENTATION USING MANUAL CODING.....	13
IMPLEMENTATION USING TRADITIONAL ETL TOOLS	15
IMPLEMENTATION USING ODI'S E-LT AND THE BUSINESS-RULE DRIVEN APPROACH	17
<i>Specifying the Business Rules in the Interface</i>	17
<i>Business Rules are Converted into a Process</i>	18
BENEFITS OF E-LT COMBINED WITH A BUSINESS-RULE DRIVEN APPROACH	20
ARCHITECTURE OF ORACLE DATA INTEGRATOR (ODI)	23
ARCHITECTURE OVERVIEW.....	23
GRAPHICAL USER INTERFACES	24
REPOSITORY	24
SCHEDULER AGENT	26
METADATA NAVIGATOR.....	27

USING ORACLE DATA INTEGRATOR IN YOUR DATA WAREHOUSE PROJECT	28
ODI AND THE DATA WAREHOUSE PROJECT	28
ORGANIZING THE TEAMS.....	28
REVERSE-ENGINEERING, AUDITING AND PROFILING SOURCE APPLICATIONS.....	30
DESIGNING AND IMPLEMENTING THE DATA WAREHOUSE'S SCHEMA	32
SPECIFYING AND DESIGNING BUSINESS RULES	33
BUILDING A DATA QUALITY FRAMEWORK	38
DEVELOPING ADDITIONAL COMPONENTS	39
PACKAGING AND RELEASING DEVELOPMENT	40
VERSIONING DEVELOPMENT	40
SCHEDULING AND OPERATING SCENARIOS.....	41
MONITORING THE DATA QUALITY OF THE DATA WAREHOUSE.....	41
PUBLISHING METADATA TO BUSINESS USERS	41
PLANNING FOR NEXT RELEASES	42
DEFINING THE TOPOLOGY IN ORACLE DATA INTEGRATOR	44
INTRODUCTION TO TOPOLOGY.....	44
DATA SERVERS	45
<i>Understanding Data Servers and Connectivity</i>	<i>45</i>
<i>Defining User Accounts or Logins for ODI to Access your Data Servers</i>	<i>47</i>
<i>Defining Work Schemas for the Staging Area</i>	<i>47</i>
<i>Defining the Data Server.....</i>	<i>48</i>
<i>Examples of Data Servers Definitions</i>	<i>49</i>
Teradata.....	49
Oracle.....	50
Microsoft SQL Server	50
IBM DB2 UDB (v8 and higher)	50
IBM DB2 UDB (v6, v7) and IBM DB2 MVS	51
IBM DB2 400 (iSeries)	51
Flat Files (Using ODI Agent).....	52
XML	52
Microsoft Excel	53
PHYSICAL SCHEMAS.....	53
CONTEXTS	54
LOGICAL SCHEMAS.....	56
PHYSICAL AND LOGICAL AGENTS	56
THE TOPOLOGY MATRIX	56
OBJECT NAMING CONVENTIONS.....	59
DEFINING ODI MODELS	61
INTRODUCTION TO MODELS.....	61
MODEL CONTENTS.....	62
IMPORTING METADATA AND REVERSE-ENGINEERING	63
<i>Introduction to Reverse-engineering.....</i>	<i>63</i>
<i>Reverse-engineering Relational Databases.....</i>	<i>64</i>
<i>Non Relational Models.....</i>	<i>64</i>
Flat Files and JMS Queues and Topics.....	64
Fixed Files and Binary Files with COBOL Copy Books	65
XML	65
LDAP Directories	66
Other Non Relation Models	66
<i>Troubleshooting Reverse-engineering</i>	<i>67</i>

JDBC Reverse-engineering Failure	67
Missing Data Types	67
Missing Constraints	68
CREATING USER-DEFINED DATA QUALITY RULES	68
ADDING USER-DEFINED METADATA WITH FLEX FIELDS	70
DOCUMENTING MODELS FOR BUSINESS USERS AND DEVELOPERS	71
EXPORTING METADATA	72
IMPACT ANALYSIS, DATA LINEAGE AND CROSS-REFERENCES	74
OBJECT NAMING CONVENTIONS	76
IMPLEMENTING ODI PROJECTS AND MAPPINGS	78
INTRODUCTION TO PROJECTS	78
IMPORTING KNOWLEDGE MODULES	79
IMPLEMENTING BUSINESS RULES IN INTERFACES	79
<i>Definition of an Interface</i>	79
<i>Designing the Business Rules</i>	80
Mappings	80
Joins	82
Filters	83
Constraints or Data Quality Rules	84
Insert/Update Checkboxes on Mappings	84
UD1..UD5 Markers on the Mappings	85
Update Key and Incremental Update Strategy	85
<i>Flow Diagram, Execution Plan and Staging Area</i>	86
<i>Impact of Changing the Optimization Context</i>	94
<i>Enforcing a Context for Sources or Target Datastores</i>	94
USING PROCEDURES	95
<i>Introduction to Procedures</i>	95
<i>Writing Code in Procedures</i>	95
<i>Using the Substitution API</i>	98
<i>Using User Options</i>	99
<i>Handling RDBMS Transactions</i>	100
<i>Using Select on Source / Action on Target</i>	102
Example1: Loading Data from a Remote SQL Database	102
Example2: Dropping a List of Tables	103
Example3: Sending Multiple Emails	103
Example4: Starting Multiple Scenarios in Parallel	104
<i>Common Pitfalls</i>	105
USING VARIABLES	105
<i>Declaring a Variable</i>	106
<i>Assigning a Value to a Variable</i>	106
<i>Evaluating the Value of a Variable</i>	106
<i>Use Cases for Variables</i>	107
Using Variables in Interfaces	107
Using Variables in Procedures	108
Using Variables in Packages	109
Using Variables in the Resource Name of a Datastore	109
Using Variables in a Server URL	110
USING SEQUENCES	112
USING USER FUNCTIONS	113
BUILDING WORKFLOWS WITH PACKAGES	114
INTRODUCTION TO PACKAGES	114
ODI TOOLS OR API COMMANDS	115

PACKAGES EXAMPLES	118
Performing a Loop in a Package.....	118
Starting Scenarios from a Package.....	119
Waiting for incoming files.....	121
ORGANIZING OBJECTS WITH MARKERS	123
GLOBAL OBJECTS.....	123
OBJECTS NAMING CONVENTIONS	123
ODI KNOWLEDGE MODULES	126
INTRODUCTION TO KNOWLEDGE MODULES	126
<i>Loading Knowledge Modules (LKM)</i>	127
<i>Integration Knowledge Modules (IKM)</i>	127
<i>Check Knowledge Modules (CKM)</i>	129
<i>Reverse-engineering Knowledge Modules (RKM)</i>	130
<i>Journalizing Knowledge Modules (JKM)</i>	131
ODI SUBSTITUTION API.....	132
<i>Working with Datastores and Object Names</i>	132
<i>Working with Lists of Tables, Columns and Expressions</i>	133
<i>Generating the Source Select Statement</i>	137
<i>Obtaining Other Information with the API</i>	138
<i>Advanced Techniques for Code Generation</i>	139
LOADING STRATEGIES (LKM).....	140
<i>Using the Agent</i>	140
<i>Using Loaders</i>	142
Using Loaders for Flat Files	142
Using Unload/Load for Remote Servers.....	144
Using Piped Unload/Load	146
<i>Using RDBMS Specific Strategies</i>	149
INTEGRATION STRATEGIES (IKM).....	149
<i>IKMs with Staging Area on Target</i>	149
Simple Replace or Append	149
Append with Data Quality Check	150
Incremental Update	153
Slowly Changing Dimensions	157
Case Study: Backup Target Table before Load.....	162
Case Study: Tracking Records for Regulatory Compliance	163
<i>IKMs with Staging Area Different from Target</i>	165
File to Server Append.....	165
Server to Server Append	167
Server to File or JMS Append.....	168
DATA QUALITY STRATEGIES (CKM).....	169
<i>Standard Check Knowledge Modules</i>	169
Case Study: Using a CKM to Dynamically Create Non-Existing References.....	174
REVERSE-ENGINEERING KNOWLEDGE MODULES (RKM).....	178
<i>RKM Process</i>	178
<i>SNP_REV_xx Tables Reference</i>	181
SNP_REV_SUB_MODEL.....	181
SNP_REV_TABLE	181
SNP_REV_COL	183
SNP_REV_KEY	184
SNP_REV_KEY_COL	185
SNP_REV_JOIN	185
SNP_REV_JOIN_COL.....	187

SNP_REV_COND.....	187
JOURNALIZING KNOWLEDGE MODULES (JKM)	188
GUIDELINES FOR DEVELOPING YOUR OWN KNOWLEDGE MODULE	188
ARCHITECTURE CASE STUDIES	189
SETTING UP REPOSITORIES.....	189
<i>General Architecture for ODI Repositories</i>	189
<i>Creating a Separate Master Repository for Production</i>	191
<i>Understanding the Impact of Work Repository IDs</i>	192
USING ODI VERSION MANAGEMENT.....	193
<i>How Version Management Works</i>	193
<i>Creating and Restoring Versions of Objects</i>	194
<i>Using Solutions for Configuration Management</i>	194
GOING TO PRODUCTION	195
<i>Releasing Scenarios</i>	195
<i>Executing Scenarios</i>	195
Executing a Scenario Manually	195
Executing a Scenario Using an Operating System Command	196
Executing a Scenario Using an HTTP Request	196
Assigning Session Keywords for a Scenario.....	197
Retrieving ODI Logs after the Execution of a Scenario.....	197
<i>Scheduling Scenarios Using the Built-in Scheduler</i>	199
<i>Using Operator in Production</i>	199
<i>Using Metadata Navigator in Production</i>	199
SETTING UP AGENTS.....	200
<i>Where to Install the Agent(s)?</i>	200
<i>Using Load Balancing</i>	200
BACKING UP REPOSITORIES	202
APPENDICES	203
APPENDIX I. ODI FOR TERADATA BEST PRACTICES	203
ARCHITECTURE OF ODI REPOSITORIES	203
REVERSE-ENGINEERING A TERADATA SCHEMA	203
TERADATA LOADING STRATEGIES	204
<i>Using Loaders for Flat Files</i>	204
<i>Using Unload/Load for Remote Servers</i>	206
<i>Using Piped Unload/Load</i>	206
TERADATA INTEGRATION STRATEGIES	209
<i>IKMs with Staging Area Different from Target</i>	210
File to Server Append.....	210
Server to Server Append	211
Server to File or JMS Append.....	212
KNOWLEDGE MODULE OPTIONS.....	212
SETTING UP AGENTS ON TERADATA ENVIRONMENT	216
<i>Where to Install the Agent(s) on Teradata environment?</i>	216
APPENDIX II: ADDITIONAL INFORMATION	216
ACRONYMS USED IN THIS BOOK	216

Oracle Data Integrator for Best Practices for a Data Warehouse

PREFACE

Purpose

This book describes the best practices for implementing Oracle Data Integrator (ODI) for a data warehouse solution. It is designed to help setup a successful environment for data integration with Enterprise Data Warehouse projects and Active Data Warehouse projects.

This book applies to Oracle Data Integrator version 10.1.3.4.

Audience

This book is intended for Data Integration Professional Services, System Integrators and IT teams that plan to use Oracle Data Integrator (ODI) as the Extract, Load and Transform tool in their Enterprise or Active Data Warehouse projects.

Additional Information

The following resource contains additional information:

- Oracle web site: <http://www.oracle.com>
- Oracle Data Integrator on-line documentation:
http://www.oracle.com/technology/products/oracle-data-integrator/10.1.3/htdocs/1013_support.html#docs
- Java reference <http://java.sun.com>
- Jython reference <http://www.jython.org>

INTRODUCTION TO ORACLE DATA INTEGRATOR (ODI)

Objectives

The objective of this chapter is to

- Introduce the key concepts of a business-rule driven architecture
- Introduce the key concepts of E-LT
- Understand what a Oracle Data Integrator (ODI) interface is
- Through a business problem case study, understand and evaluate some different development approaches including:
 - Manual coding
 - Traditional ETL
 - ODI'S business-rule driven approach combined with E-LT

Business-Rules Driven Approach

Introduction to Business rules

Business rules specify mappings, filters, joins and constraints. They often apply to metadata to transform data and are usually described in natural language by business users. In a typical data integration project (such as a Data Warehouse project), these rules are defined during the specification phase in documents written by business analysts in conjunction with project managers.

Business Rules usually define “What” to do rather than “How” to do it.

They can very often be implemented using SQL expressions, provided that the metadata they refer to is known and qualified in a metadata repository.

Examples of business rules are given in the table below:

Business Rule	Type	SQL Expression
Sum of all amounts of items sold during October 2005 multiplied by the item price	Mapping	<pre>SUM(CASE WHEN SALES.YEARMONTH=200510 THEN SALES.AMOUNT * PRODUCT.ITEM_PRICE ELSE 0 END)</pre>
Products that start with 'CPU' and that belong to the hardware category	Filter	<pre>Upper (PRODUCT.PRODUCT_NAME) like 'CPU%' And PRODUCT.CATEGORY = 'HARDWARE'</pre>
Customers with their orders and order lines	Join	<pre>CUSTOMER.CUSTOMER_ID = ORDER.ORDER_ID And ORDER.ORDER_ID = ORDER_LINE.ORDER_ID</pre>

Business Rule	Type	SQL Expression
Reject duplicate customer names	Unique Key Constraint	Unique key (CUSTOMER_NAME)
Reject orders with a link to an non-existent customer	Reference Constraint	Foreign key on ORDERS (CUSTOMER_ID) references CUSTOMER (CUSTOMER_ID)

Mappings

A mapping is a business rule implemented as an **SQL expression**. It is a transformation rule that maps source columns (or fields) onto one of the target columns. It can be executed by a relational database server at run-time. This server can be the source server (when possible), a middle tier server or the target server.

Joins

A join operation links records in several data sets, such as tables or files. Joins are used to **link multiple sources**. A join is implemented as an **SQL expression** linking the columns (fields) of two or more data sets.

Joins can be defined regardless of the physical location of the source data sets involved. For example, a JMS queue can be joined to a relational table.

Depending on the technology performing the join, it can be expressed as an inner join, right outer join, left outer join and full outer join.

Filters

A filter is an **expression** applied to source data sets columns. Only the records matching this filter are processed by the data flow.

Constraints

A constraint is an object that defines the rules enforced on data sets' data. A constraint ensures the validity of the data in a given data set and the integrity of the data of a model. Constraints on the target are used to check the validity of the data before integration in the target.

Traditional ETL versus E-LT Approach

Traditional ETL tools operate by first *Extracting* the data from various sources, *Transforming* the data on a proprietary, middle-tier ETL engine, and then *Loading* the transformed data onto the target data warehouse or integration server. Hence the term "ETL" represents both the names and the order of the operations performed, as shown in Figure 1 below.

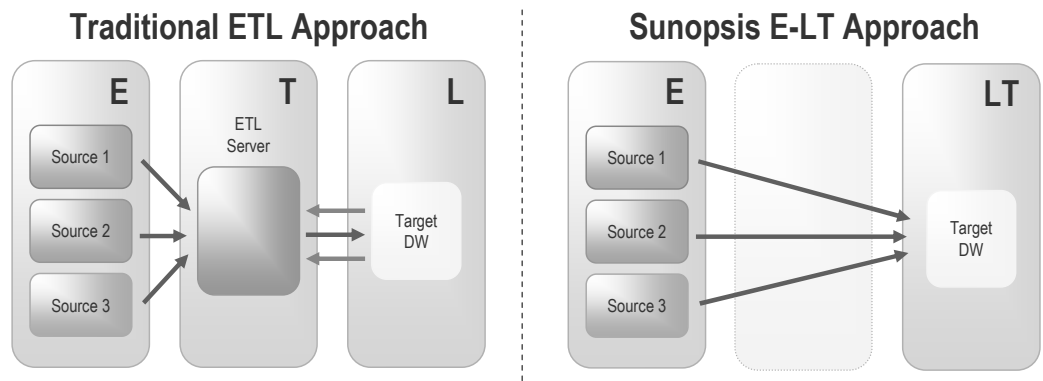


Figure 1: Traditional ETL approach compared to E-LT approach

In response to the issues raised by ETL architectures, a new architecture has emerged, which in many ways incorporates the best aspects of manual coding and automated code-generation approaches. Known as “E-LT”, this new approach changes where and how data transformation takes place, and leverages existing developer skills, RDBMS engines and server hardware to the greatest extent possible. In essence, E-LT moves the data transformation step to the target RDBMS, changing the order of operations to: *Extract* the data from the source tables, *Load* the tables into the destination server, and then *Transform* the data on the target RDBMS using native SQL operators. Note, with E-LT there is no need for a middle-tier engine or server as shown in Figure 1 above.

Understanding Oracle Data Integrator (ODI) Interfaces

An interface is an ODI object stored in the ODI Repository that enables the loading of **one target datastore** with data transformed from one or more **source datastores**, based on **business rules** implemented as **mappings, joins, filters** and **constraints**.

A **datastore** can be:

- a table stored in a relational database
- an ASCII or EBCDIC file (delimited, or fixed length)
- a node from a XML file
- a JMS topic or queue from a Message Oriented Middleware such as IBM Websphere MQ
- a node from a LDAP directory
- an API that returns data in the form of an array of records

Figure 3 shows a screenshot of an ODI interface that loads data into the FACT_SALES target table. Source Data is defined as a heterogeneous query on the SALES_FILE file, the DIM_PRODUCT and DIM_DAY tables.

Mappings, joins, filters and constraints are defined within this window.

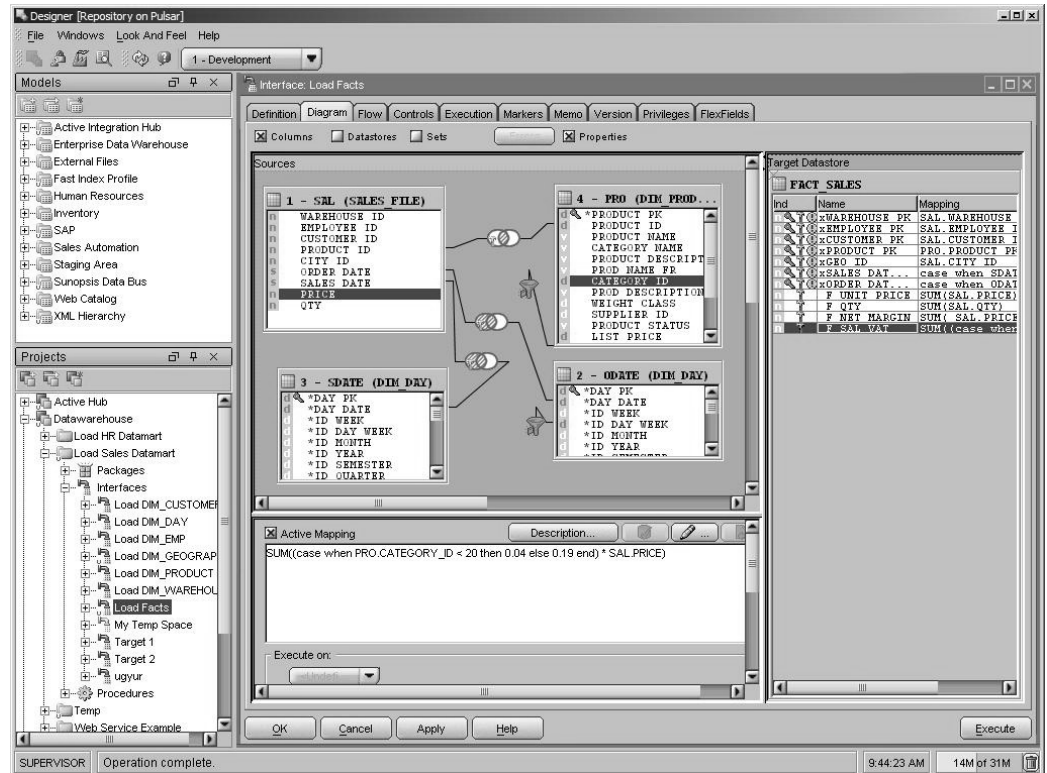


Figure 2: Example of an ODI Interface

Wherever possible, ODI interfaces generate E-LT operations that relegate transformations to target RDBMS servers.

A Business Problem Case Study

Figure 3 describes an example of a business problem to extract, transform and load data from an Oracle database and a file into a target Teradata table.

Data is coming from 2 Oracle tables (ORDERS joined to ORDER_LINES) and is combined with data from the CORRECTIONS file. The target SALES Teradata table must match some constraints such as the uniqueness of the ID column and valid reference to the SALES_REP table.

Data must be transformed and aggregated according to some mappings as shown in Figure 3.

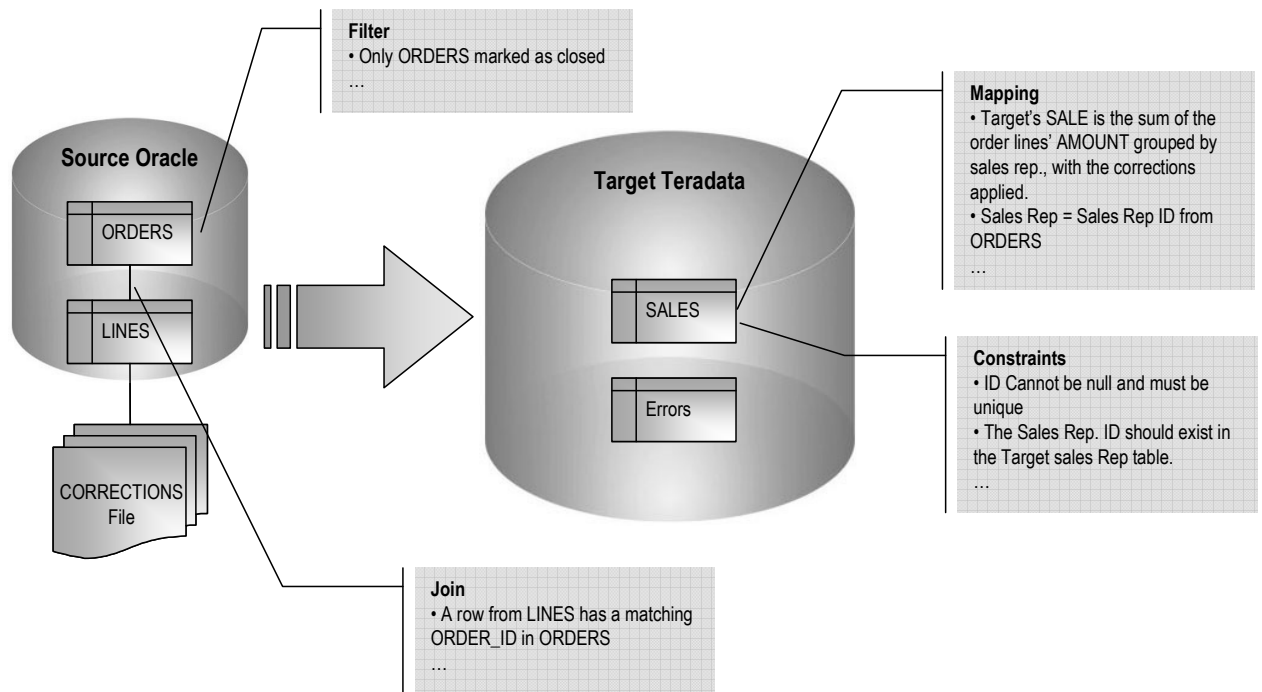


Figure 3: Example of a business problem

Translating these business rules from natural language to SQL expressions is usually straightforward. In our example, the rules that appear in the figure could be translated as follows:

Type	Rule	SQL Expression / constraint
Filter	Only ORDERS marked as closed	<code>ORDERS.STATUS = 'CLOSED'</code>
Join	A row from LINES has a matching ORDER_ID in ORDERS	<code>ORDERS.ORDER_ID = LINES.ORDER_ID</code>
Mapping	Target's SALE is the sum of the order lines' AMOUNT grouped by sales rep., with the corrections applied.	<code>SUM(LINES.AMOUNT + CORRECTIONS.VALUE)</code>
Mapping	Sales Rep = Sales Rep ID from ORDERS	<code>ORDERS.SALES_REP_ID</code>
Constraint	ID must not be null	ID is set to "not null" in the data model
Constraint	ID must be unique	A Unique Key Is added to the data model with (ID) as set of columns
Constraint	The Sales Rep. ID should exist in the Target sales Rep table	A Reference (Foreign Key) is added in the data model on <code>SALES.SALES_REP = SALES_REP.SALES_REP_ID</code>

Implementation using Manual Coding

When implementing such a data flow using, manual coding, one would probably use several steps, several languages, and several scripting tools or utilities.

Figure 4 gives an overview of the different steps needed to achieve such an extract, transform and load process.

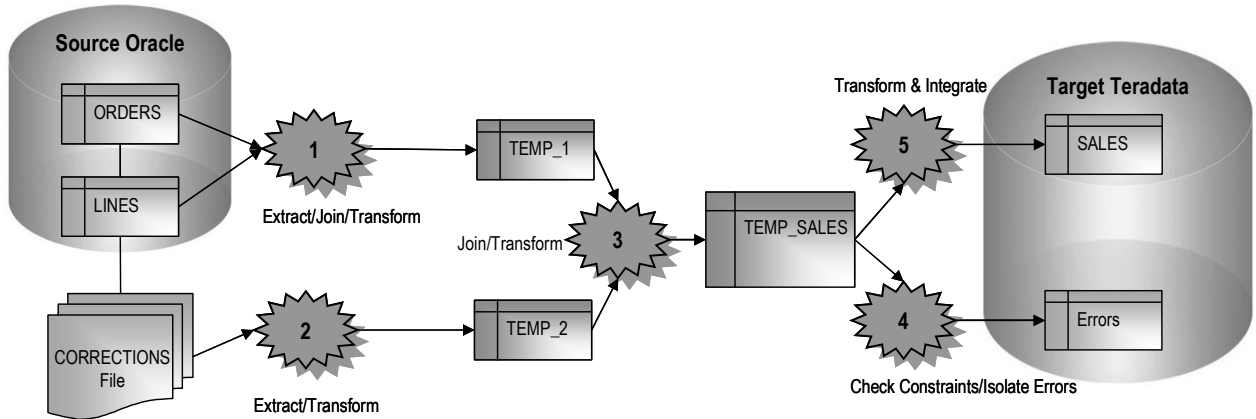


Figure 4: Sequence of Operations for the Process

There are, of course, several technical solutions for implementing such a process. One of them (probably the most efficient, as it uses Teradata as a transformation engine) is detailed in the following table:

Step	Description	Example of code
1	<ul style="list-style-type: none"> Execute the join between ORDERS and LINES as well as the filters on the source Oracle database using SQL. Extract the result of the select query into a temporary ASCII File using the Oracle SQL*PLUS utility. Use Teradata's FASTLOAD utility to load the temporary ASCII file into TEMP_1 Teradata table 	<pre> select ... from ORDERS, LINES where ORDERS.STATUS = 'CLOSED' and ORDERS.ORDER_ID = LINES.ORDER_ID \$ sqlplus user/passw@db @myscript.sql logon tdat/usr,pwd begin loading... define order_id (varchar(10)), ... \$ fastload < myfastload.script </pre>
2	<ul style="list-style-type: none"> Use Teradata's FASTLOAD utility to load the CORRECTIONS ASCII file into TEMP_2 Teradata table 	<pre> logon tdat/usr,pwd begin loading... define corr_id (varchar(12)), ... \$ fastload < myfastload.script </pre>

Step	Description	Example of code
3	<ul style="list-style-type: none"> Join, transform and aggregate the 2 temporary tables TEMP1 and TEMP2 and load the results into a 3rd table (TEMP_SALES) using SQL and BTEQ 	<pre> insert into TEMP_SALES (...) select SUM(TEMP_1.AMOUNT+TEMP_2.VALUE), TEMP1.SALES_REP_ID, ... from TEMP_1 left outer join TEMP2 on (TEMP1.LINEID = TEMP2.CORR_ID) where ...; \$ bteq < step3.sql </pre>
4	<ul style="list-style-type: none"> Check Unique constraints using SQL and insert the errors into the Errors table Check Reference constraints using SQL and insert the errors into the Error table 	<pre> insert into Errors(...) select ... from TEMP_SALES where ID in (select ID from TEMP_SALES group by ID having count(*) > 1) ; insert into Errors(...) select ... from TEMP_SALES where SALES_REP not in (select SALES_REP_ID from SALES_REP) \$ bteq < step4.sql </pre>
5	<ul style="list-style-type: none"> Finally, use SQL logic to insert / update into the target SALES table using a query on TEMP_SALES 	<pre> update SALES set ... from ... ; insert into SALES (...) select ... from TEMP_SALES where TEMP_SALES.ID not in (select ID from Errors) ... ; \$ bteq < step5.sql </pre>

The benefits of this approach are:

- High performance:
 - Uses pure set-oriented SQL to avoid row-by-row operations
 - Uses Teradata as a transformation engine to leverage the power of parallel processing
 - Uses in-place utilities such as Teradata FASTLOAD and BTEQ
- Code flexibility:
 - Leverages the latest features of Teradata such as new built-in transformation functions

However, this approach raises several issues that become painful as the Enterprise Data Warehouse projects grow, and more developers get involved. These issues are:

- Poor productivity
 - Every load process needs to be developed as a set of scripts and programs, within different environments and with several languages.
 - Business rules (“what happens to the data” –SUM(AMOUNT + VALUE)) are mixed with technical logic (“how to move / load the data” – bteq, fastload, insert etc.)
 - Moving to production is often difficult when developers haven’t designed environment variables, or variable qualified names for their objects.
- High cost of data quality implementation
 - Data cleansing / data quality according to predefined constraints is usually avoided due to the cost of its implementation
 - Every project has its own definition of the data quality without any centralized framework (default structure of the error tables, error recycling etc.)
- Hard maintenance
 - Even when project managers have set up a framework, every script may “reinvent the wheel” and contain specific routines that make it hard to understand
 - Developments are spread in several machines and folders, without a central repository
 - Impact analysis is impossible as there is no metadata management and no cross references mechanism
- No project flexibility
 - The cost of a change to the data models or to the business rules becomes such a constraint that IT teams refuse it, leading to frustration amongst the business users.

Implementation using Traditional ETL tools

Traditional ETL tools perform all the transformations in a proprietary engine. They often require additional hardware to stage the data for the transformations. None of them really leverages the power of the RDBMS.

A typical ETL architecture is shown in Figure 5.

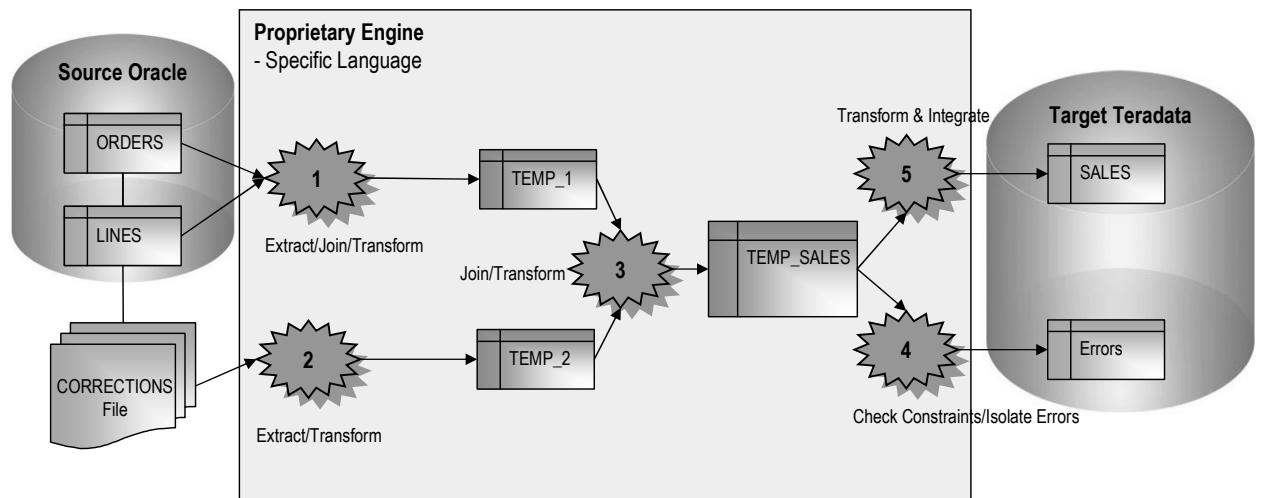


Figure 5: Implementation Using an ETL

Every transformation step requires a specific connector or transformer.

ETL tools are often known for the following advantages:

- Centralized development and administration
 - Single graphical user interface
 - Centralized repository
- Easier maintenance
 - Impact analysis (for certain tools)

Unfortunately this ETL approach presents several drawbacks:

- Poor performance
 - As the data needs to be processed in the engine, it is often processed row by row
 - When data from the target database is referenced - table lookups for example - it needs to be extracted from the database, into the engine and then moved back again to the target database.
 - Very few mappings, joins, aggregations and filters are given to the powerful engine of the RDBMS
- Bad productivity
 - Every load process needs to be developed as a set of steps that mix business rules (“what happens to the data” – SUM(AMOUNT + VALUE)) with technical logic (“how to move / load the data” – connector 1, connector 2 etc.)

- Moving to production is often difficult when developers haven't designed environment variables, or variable qualified names within their queries
- Some of them still require the use of heavy manual coding to achieve certain particular tasks, or to leverage the RDBMS' powerful transformation functions
- High Cost
 - ETL tools require additional hardware
 - ETL tools require specific skills

Implementation using ODI'S E-LT and the Business-rule Driven Approach

Implementing a business problem using ODI is a very easy and straightforward exercise. It is done by simply translating the business rules into an interface. Every business rule remains accessible from the Diagram panel of the interface's window.

Specifying the Business Rules in the Interface

Figure 6 gives an overview of how the business problem is translated into an ODI interface:

- The ORDERS, LINES and CORRECTION datastores are dragged and dropped into the "Source" panel of the interface
- The Target SALES datastore is dropped in the "Target Datastore" panel
- Joins and filters are defined by dragging and dropping columns in the "Source" panel
- Mappings are defined by selecting every target column and by dragging and dropping columns or by using the advanced expression editor.
- Constraints are defined in the "Control" tab of the interface. They define how flow data is going to be checked and rejected into the Errors table.

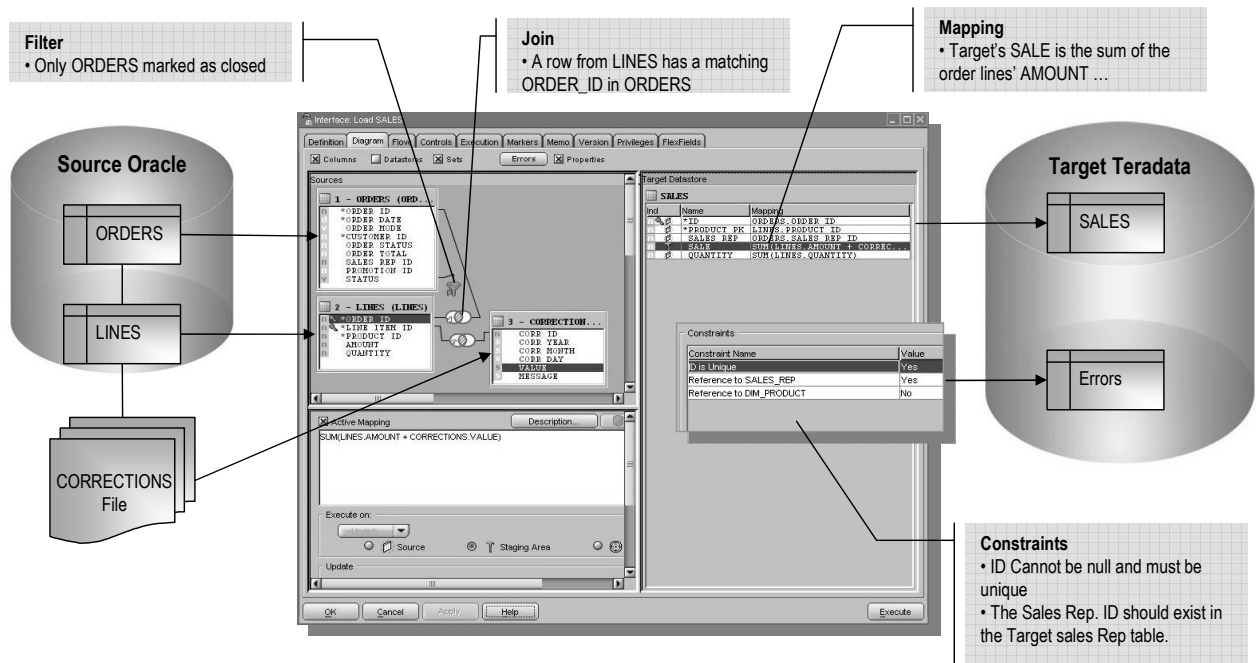


Figure 6: Implementation using Oracle Data Integrator

Business Rules are Converted into a Process

Business rules defined in the interface need to be split into a process that will carry out the joins filters, mappings, and constraints from source data to target tables. Figure 7 defines the problem to be solved.

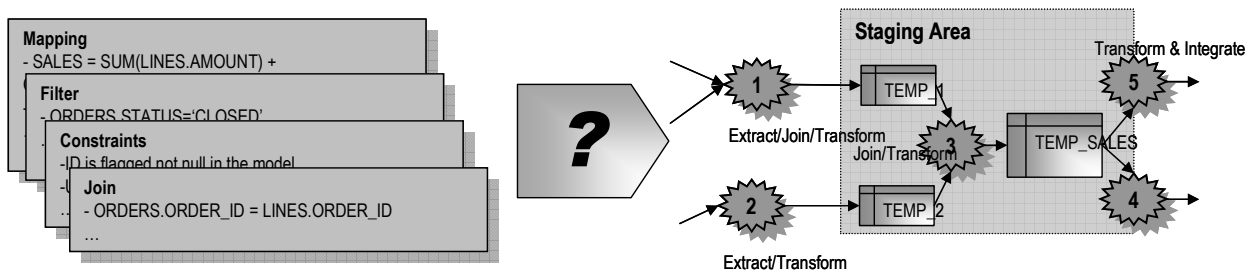


Figure 7: How to convert business rules into a process?

By default, Oracle Data Integrator (ODI) will use the RDBMS as a **staging area** for loading source data into temporary tables and applying all the required mappings, staging filters, joins and constraints.

The **staging area** is a separate area in the RDBMS (a user/database) where Oracle Data Integrator (ODI) creates its temporary objects and executes some of the rules (mapping, joins, final filters, aggregations etc). When performing the operations this way, Oracle Data Integrator (ODI) behaves like an E-LT as it first extracts and loads the temporary tables and then finishes the transformations in the target RDBMS.

In some particular cases, when source volumes are small (less than 500,000 records), this staging area can be located in memory in ODI's in-memory relational database – ODI Memory Engine. ODI would then behave like a traditional ETL tool.

Figure 8 shows the process **automatically generated** by Oracle Data Integrator (ODI) to load the final SALES table. The business rules, as defined in Figure 6 will be transformed into code by the **Knowledge Modules (KM)**. The code produced will generate several steps. Some of these steps will *extract* and *load* the data from the sources to the staging area (Loading Knowledge Modules - LKM). Others will *transform* and *integrate* the data from the staging area to the target table (Integration Knowledge Module - IKM). To ensure data quality, the Check Knowledge Module (CKM) will apply the user defined constraints to the staging data to isolate erroneous records in the Errors table.

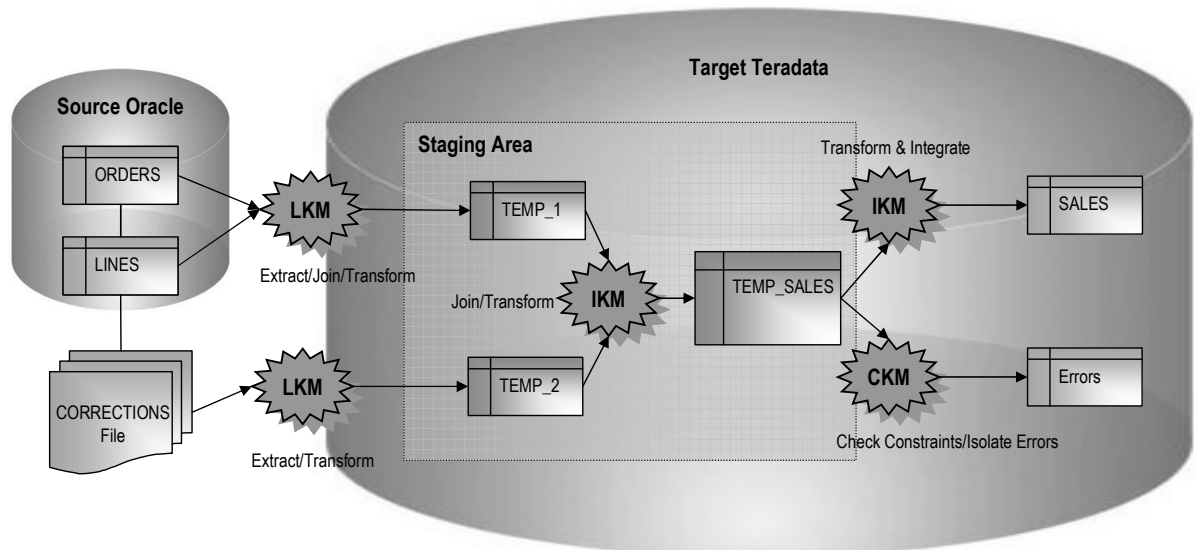


Figure 8: ODI Knowledge Modules in action

ODI Knowledge Modules contain the actual code that will be executed by the various servers of the infrastructure. Some of the code contained in the Knowledge Modules is generic. It makes calls to the ODI Substitution API that will be bound at run-time to the business-rules and generates the final code that will be executed. Figure 9 illustrates this mechanism.

During design time, rules are defined in the interfaces and Knowledge Modules are selected.

During run-time, code is generated and every API call in the Knowledge Modules (enclosed by <% and %>) is replaced with its corresponding object name or expression, with respect to the metadata provided in the Repository.

For example, a call to `<%=snpRef.getTable("TARG_NAME") %>` will return the name of the target table of the interface with the appropriate qualifier according to context information, topology setup etc. A typical SQL INSERT statement would, for example, be coded in a Knowledge Module as follows:

```
INSERT INTO <%=snpRef.getTable("TARG_NAME") %> ...
```

This template of code will of course generate different SQL statements depending on the target table ("INSERT INTO MyDB1.SALES..." when the target is the SALES table, "INSERT INTO DWH_DB.PRODUCT" when the target is the PRODUCT table etc.)

Once the code is generated, it is submitted to an ODI Agent, which will either redirect it to the appropriate database engines and operating systems, or will execute it when needed (memory engine transformation, java or jython code etc.)

In most cases, the agent is simply a conductor that does not touch the data.

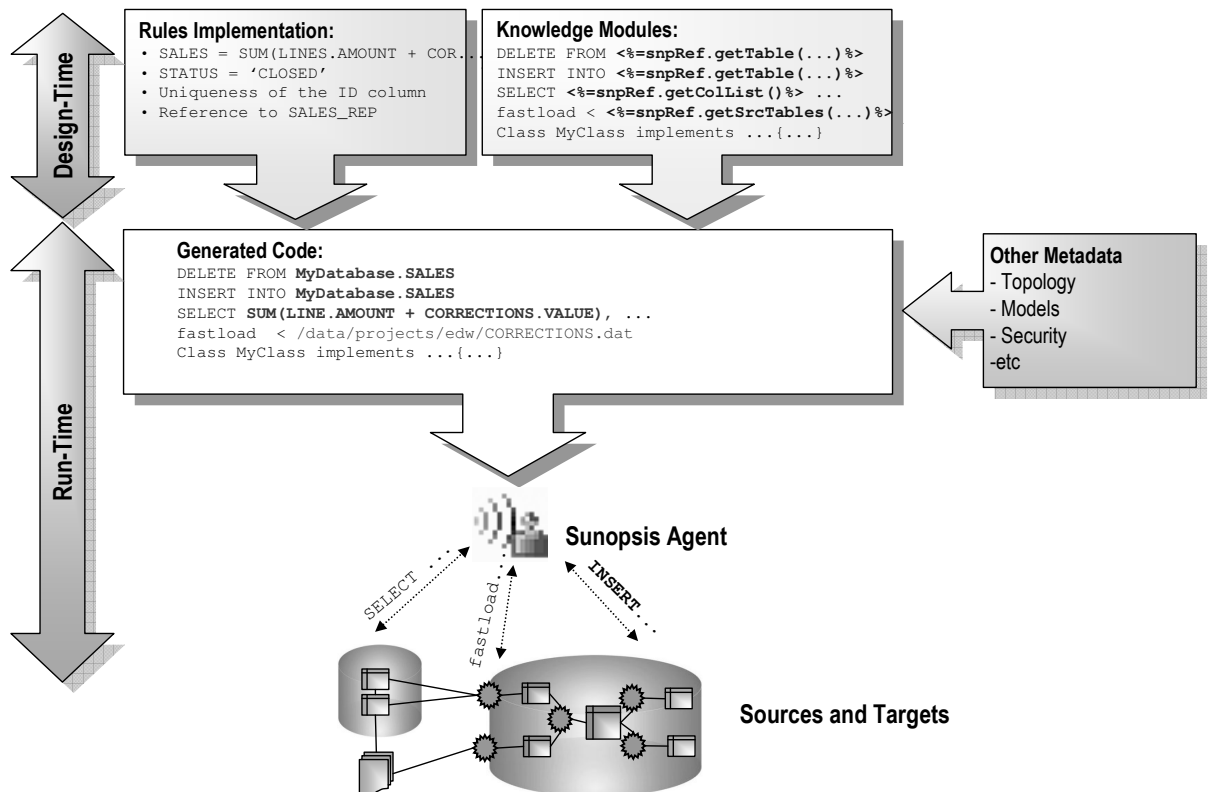


Figure 9: How Knowledge Modules generate native code

Benefits of E-LT Combined with a Business-rule Driven Approach

Compared to other architectures (manual coding and traditional ETL), ODI mixes the best of both worlds:

- Productivity / Maintenance

- The business-rules driven approach delivers greater productivity as developers simply need to concentrate on the “What” without caring about the “How”. They define SQL expressions for the business rules, and ODI Knowledge Modules generate the entire set of SQL operations needed to achieve these rules.
- When a change needs to be made in operational logic (such as “creating a backup copy of every target table before loading the new records”), it is simply applied in the appropriate Knowledge Module and it automatically impacts the hundreds of interfaces already developed. With a traditional ETL approach, such a change would have necessitated opening every job and manually adding the new steps, increasing the risk of mistakes and inconsistency.
- Flexibility and a shallow learning curve are ensured by leveraging the RDBMS’ latest features.
- With a centralized repository that describes all the metadata of the sources and targets and a single unified and comprehensive graphical interface, maintenance is greatly optimized as cross-references between objects can be queried at any time. This gives the developers and the business users a single entry point for impact analysis and data lineage (“What is used where?”, “Which sources populate which targets?” etc.)
- In the ODI repository, the topology of the infrastructure is defined in detail, and moving objects between different execution contexts (Development, Testing, QA, Production, etc.) is straightforward. With a powerful version control repository, several teams can work on the same project within different release stages, with guaranteed consistency of deliverables.
- With a centralized framework for Data Quality, developers spend less time on defining technical steps, and more time on the specification of data quality rules. This helps to build a consistent and standardized Data Warehouse.
- High Performance:
 - The E-LT architecture leverages the power of all the features of in-place databases engines. ODI generates pure set-oriented SQL optimized for each RDBMS. Which can take advantage of advanced features such as parallel processing or other advanced features.
 - Native database utilities can be invoked by the ODI Knowledge Modules provided.
 - When data from the target database is referenced - table lookups for example, it doesn’t need to be extracted from the database, into an engine. It remains where it is, and it is processed by database engine.

- Low Cost:
 - Oracle Data Integrator doesn't require a dedicated server. The loads and transformations are carried out by the RDBMS.

In conclusion, with its business-rule driven E-LT architecture, Oracle Data Integrator is the best solution for taking advantage of both manual coding and traditional ETL worlds.

ARCHITECTURE OF ORACLE DATA INTEGRATOR (ODI)

Architecture Overview

The architecture of ODI relies on different components that collaborate together, as described in Figure 10.

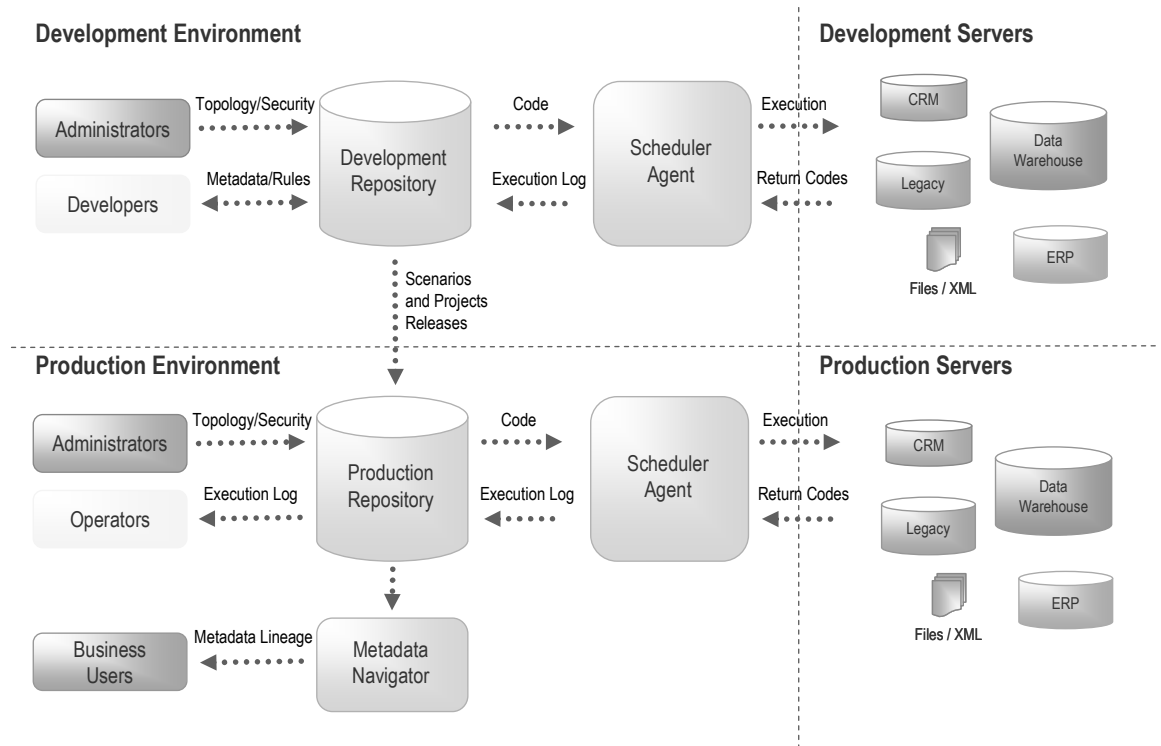


Figure 10: Architecture Overview

The central component of the architecture is the ODI Repository. It stores configuration information about the IT infrastructure, metadata of all applications, projects, scenarios, and the execution logs. Many instances of the repository can coexist in the IT infrastructure. The architecture of the repository was designed to allow several separated environments that exchange metadata and scenarios (for example: Development, Test, Maintenance and Production environments). In the figure above, two repositories are represented: one for the development environment, and another one for the production environment. The repository also acts as a version control system where objects are archived and assigned a version number. The ODI Repository can be installed on an OLTP relational database.

Administrators, Developers and Operators use ODI Graphical User Interfaces (GUI) to access the repositories. These GUIs are for administering the infrastructure (Security and Topology), reverse-engineering the metadata and developing projects (Designer), scheduling and operating scenarios and execution logs (Operator).

Business users (as well as developers, administrators and operators), can have read access to the repository through any web based application (such as Firefox or Internet Explorer). Metadata Navigator application server makes the link between these web browsers and the ODI Repository.

At design time, developers generate scenarios from the business rules that they have designed. The code of these scenarios is then retrieved from the repository by the Scheduler Agent. This agent then connects to the servers of the infrastructure and asks them to execute the code. It then stores the return codes and messages, as well as additional logging information – such as the number of records processed, the elapsed time etc. - in the ODI Repository.

Graphical User Interfaces

ODI Graphical User Interfaces are Java based interfaces. They can be installed on any platform that supports a Java Virtual Machine 1.4 (Windows, Linux, HP-UX, Solaris, pSeries, etc.).

The Oracle Data Integrator GUI is composed of 4 different modules:

- **Topology Manager** is the GUI that manages the data that describe the physical and logical architecture of the infrastructure. It is used to register servers, schemas and agents in the ODI Master Repository. This module is usually used by the administrators of the infrastructure.
- **Security Manager** is the GUI for managing users and their privileges in ODI. Profiles and users can be given access rights to objects and ODI features. This module is usually used by security administrators.
- **Designer** is the GUI for defining metadata and the transformation/data quality rules that will generate the scenarios for production. Project development is managed from that GUI. It is the core module for developers and metadata administrators.
- **Operator** is the GUI for managing and monitoring Oracle Data Integrator in production. It is dedicated to the operators and shows the execution logs with the number of potential errors, the number of rows processed, execution statistics etc. At design time, developers will use Operator for debugging purposes.

Repository

The **ODI Repository** is composed of a Master Repository and several Work Repositories. Objects developed or configured through the ODI GUIs are stored in one or other of these repository types.

There is usually only one Master Repository (except in some particular cases as described in section “Setting up Repositories”) It stores the following information:

- Security information including users, profiles and rights for the ODI platform
- Topology information including technologies, server definitions, schemas, contexts, languages etc.
- Versioned and archived objects.

The information contained in the Master Repository is maintained with the Topology Manager and Security Manager modules.

The Work Repository is the one that contains actual developed objects. Several work repositories may coexist in the same ODI installation (for example, to have separate environments or to match a particular versioning life cycle). A Work Repository stores information for:

- Models, including schema definition, datastores structures and metadata, fields and columns definitions, data quality constraints, cross references, data lineage etc.
- Projects, including business rules, packages, procedures, folders, Knowledge Modules, variables etc.
- Scenario execution, including scenarios, scheduling information and logs.

When the Work Repository contains only the execution information (typically for production purposes), it is then called an Execution Repository.

Figure 11 gives an overview of a detailed (but still simplified) repository architecture where developments, tests and production are in separate Work Repositories. When the Development team finishes developing certain projects, it exports them into versions in the unique Master Repository. A Test team imports these released versions for testing them in a separate Work Repository, thus allowing the development team to continue working on the next versions. When the Test team successfully validates the developments, the Production team then imports the executable versions of the developed projects (called scenarios) into the final production work repository.

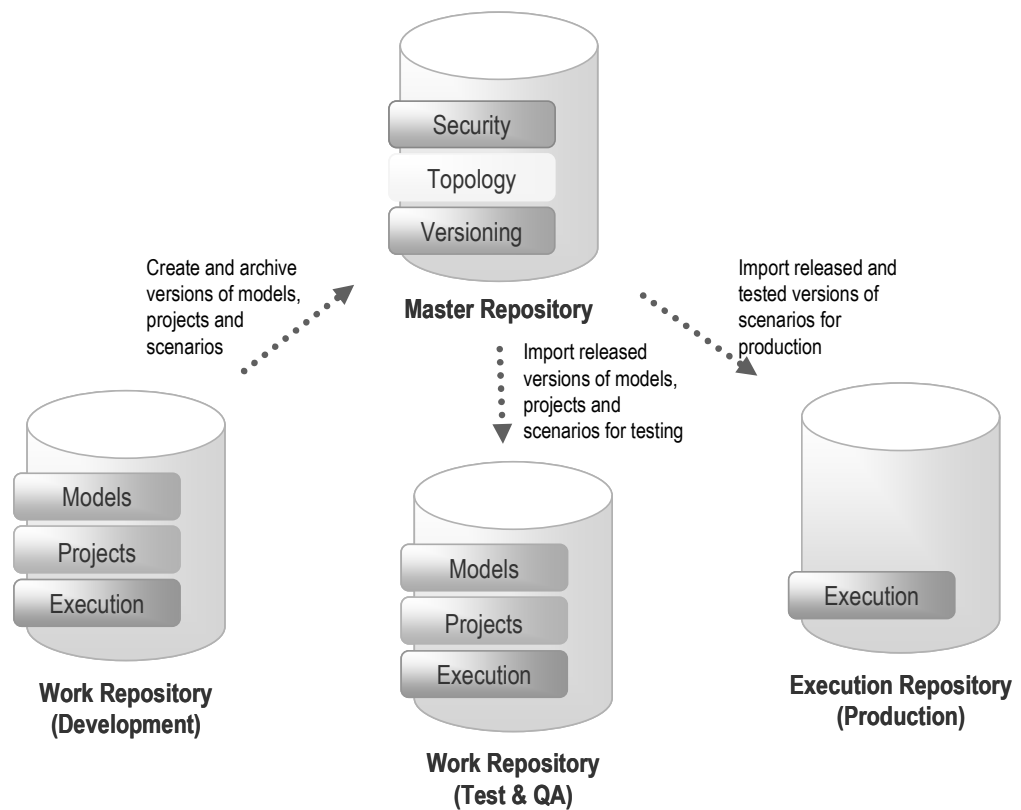


Figure 11: Architecture of ODI Repositories

It is recommended that you install these repositories in an OLTP database distinct from the source or target applications.

Section “Setting up Repositories” further discusses the impact of the architecture of ODI Repositories in large data warehouse projects.

Scheduler Agent

The Scheduler Agent orchestrates the execution of developed scenarios. It can be installed on any platform that supports a Java Virtual Machine 1.4 (Windows, Linux, HP-UX, Solaris, pSeries, iSeries, zSeries, etc.).

External scheduling systems can also invoke the agent to have it execute developed scenarios.

Thanks to Oracle Data Integrator’s E-LT architecture, the Scheduler Agent rarely performs any transformation. It usually simply retrieves code from the Execution Repository, and it requests that database servers, operating systems or scripting engines execute it. After execution, the scheduler agent updates the logs in the repository, reporting error messages and execution statistics.

Typical data warehouse projects will require a single Scheduler Agent in production; however, section “Setting up Agents” discusses particular architectures where several load-balanced agents may be appropriate.

Metadata Navigator

Metadata Navigator is a Java J2EE Application that enables web based applications accessing ODI Repositories to perform metadata navigation within projects, models, logs, etc. By default, it is installed on Jakarta Tomcat Application Server.

Business users, developers, operators and administrators will use their internet browser to access Metadata Navigator. From its web interface, they can display flow maps, audit data lineage and even drill down to the field level to understand the transformations that affect the data.

USING ODI IN YOUR DATA WAREHOUSE PROJECT

ODI and the Data Warehouse Project

The main goal of a Data Warehouse is to consolidate and deliver accurate indicators to business users to help them make decisions regarding their every day business. A typical project is composed of several steps and milestones. Some of these are:

- Defining business needs (Key Indicators)
- Identifying source data that concerns key indicators; specifying business rules to transform source information into key indicators
- Modeling the data structure of the target warehouse to store the key indicators
- Populating the indicators by implementing business rules
- Measuring the overall accuracy of the data by setting up data quality rules
- Developing reports on key indicators
- Making key indicators and metadata available to business users through ad-hoc query tools or predefined reports
- Measuring business users' satisfaction and adding/modifying key indicators

Oracle Data Integrator will help you cover most of these steps, from source data investigation to metadata lineage, and through loading and data quality audit. With its repository, ODI will centralize the specification and development efforts and provide a unique architecture on which the project can rely to succeed.

Organizing the Teams

As Oracle Data Integrator relies on a centralized repository, different types of users may need to access it. The list below describes how ODI may be used by your teams.

Profile	Description	ODI Modules Used
Business User	Business users have access to the final calculated key indicators through reports or ad-hoc queries. In some cases, they need to understand what the definition of the indicators is, how they are calculated and when they were updated. Alternatively, they need to be aware of any data quality issue regarding the accuracy of their indicators.	<ul style="list-style-type: none">• Metadata Navigator
Business Analyst	Business Analysts define key indicators. They know the source applications and specify business rules to transform source data into meaningful target indicators. They are in charge of maintaining translation data from operational semantics to the	<ul style="list-style-type: none">• Designer (limited access)• Metadata Navigator

Profile	Description	ODI Modules Used
	unified data warehouse semantic.	
Developer	Developers are in charge of implementing the business rules in respect of the specifications described by the Business Analysts. They release their work by providing executable scenarios to the production team. Developers must have both technical skills regarding the infrastructure and business knowledge of the source applications.	<ul style="list-style-type: none"> • Topology (read only access) • Designer: <ul style="list-style-type: none"> • Limited access to Models • Full access to Projects • Operator • Metadata Navigator
Metadata Administrator	Metadata Administrators are in charge of reverse-engineering source and target applications. They guarantee the overall consistency of Metadata in ODI Repository. They have an excellent knowledge of the structure of the sources and targets and they have participated in the data modeling of key indicators. In conjunction with Business Analysts, they enrich the metadata by adding comments, descriptions and even integrity rules (such as constraints). Metadata Administrators are responsible for version management.	<ul style="list-style-type: none"> • Topology (limited access) • Designer: <ul style="list-style-type: none"> • Full access to Models • Restore access to Projects • Metadata Navigator
Database Administrator	Database Administrators are in charge of defining the technical database infrastructure that supports ODI. They create the database profiles to let ODI access the data. They create separate schemas and databases to store the Staging Areas. They make the environments accessible by describing them in the Topology	<ul style="list-style-type: none"> • Topology (full access) • Designer (full access) • Operator (full access) • Metadata Navigator
System Administrator	System Administrators are in charge of maintaining technical resources and infrastructure for the project. For example, they may <ul style="list-style-type: none"> • install and monitor Scheduler Agents • backup / restore Repositories • install and monitor Metadata Navigator • Setup environments (development, test, maintenance etc.) 	<ul style="list-style-type: none"> • Scheduler Agents • Topology (limited access) • Metadata Navigator
Security	The Security Administrator is in charge of defining the	<ul style="list-style-type: none"> • Security (full access)

Profile	Description	ODI Modules Used
Administrator	security policy for the ODI Repository. He or she creates ODI users and grants them rights on models, projects and contexts.	<ul style="list-style-type: none"> • Designer (read access) • Topology (read access) • Metadata Navigator
Operator	Operators are in charge of importing released and tested scenarios into the production environment. They schedule the execution of these scenarios. They monitor execution logs and restart failed sessions when needed.	<ul style="list-style-type: none"> • Operator • Metadata Navigator

ODI Master Repository contains built-in default profiles that can be assigned to users. The following table suggests how to use these built-in profiles:

Profile	Built-in Profile in Oracle Data Integrator
Business User	CONNECT, NG REPOSITORY EXPLORER
Business Analyst	CONNECT, NG REPOSITORY EXPLORER, NG DESIGNER
Developer	CONNECT, DESIGNER, METADATA BASIC
Metadata Administrator	CONNECT, METADATA ADMIN, VERSION ADMIN
Database Administrator	CONNECT, DESIGNER, METADATA ADMIN, TOPOLOGY ADMIN
System Administrator	CONNECT, OPERATOR
Security Administrator	CONNECT, SECURITY ADMIN
Operator	CONNECT, OPERATOR

Reverse-engineering, Auditing and Profiling Source Applications

A good starting point for projects is to understand the contents and the structure of source applications. Rather than having paper-based documentation on source applications, a good practice is to connect to these source applications using ODI and to capture their metadata. Once this is achieved, it is usually helpful to define some data quality business rules in ODI Repository. This helps check the data consistency in the source and validate your understanding of the data models. At this phase of the project it is important to start having an answer to the following questions:

- How many different sources do we have to take in account to calculate our indicators?
- Is the data needed for our indicators present in the source systems?

- What data quality challenges will we have to address to ensure the accuracy of the target warehouse?
- What source system(s) will provide the master data (dimensions) that will be referenced by our indicators?
- What data volumes are we going to manipulate?
- And so forth.

In some cases, source applications are not accessible directly. Only ASCII or binary files extracts from these applications are provided. Starting to work with these source files before implementing the data warehouse model is recommended, as they represent the “vision of the truth” that the production source systems want to give you. The metadata of these files should typically be described in the repository. Samples of the files can also be initially loaded into temporary target tables to validate their structure and content.

All this can be implemented in Oracle Data Integrator as follows:

- Connect up source applications or files in Topology
- Define a logical architecture in Topology
- Create one model per logical schema in Designer
- Reverse Engineer models when possible or manually describe datastores
 - Use standard JDBC Reverse-engineering to get database metadata
 - Use Customized Reverse-engineering strategies (Reverse Knowledge Modules) when standard JDBC reverse is not applicable (or not accurate)
 - Use Cobol Copy Book import for ASCII or binary files if available
 - Use Delimited File Reverse for ASCII delimited files
- Enrich the metadata by adding information when it is not available:
 - Datastores and Columns descriptions and titles
 - Unique keys information (Primary Keys, Alternate Keys)
 - Referential Integrity assumptions between datastores (Foreign Keys)
 - Check constraints (for example to check the domain of values for a column)
- When the source data is stored in files, develop simple interfaces to load these files into a temporary area in the target database to better evaluate their level of data quality.
- Locate the important entities in the source data and use the following features to profile the contents:

- View Data on a datastore
- View distribution of values for a column of a datastore
- Count the number of records
- Run ad-hoc SQL queries from Designer
- Validate the constraints that you have defined in the models by performing data quality control:
 - Schedule or interactively run data quality “static” control on the datastores and models
 - Access and understand the contents of the error tables
 - Propose alternatives for handling data discrepancies in the future
 - Define the “acceptable” level of data quality for the target warehouse

Of course, these action items should be done in conjunction with business analysts. Normally, in parallel to this phase, you will be designing the data model of the target Data Warehouse. Data quality audit errors and data profiling will help you refining your understanding of the truth, and consequently, will lead you to a better modeling of the target warehouse.

Designing and Implementing the Data Warehouse’s Schema

This section could be a book on its own. The goal here is not to provide you with guidelines in Data Warehouse modeling, but rather to give some general hints and tips that may affect your ODI development.

It is always a good practice to have an Operational Data Store (ODS) to store raw operational data. Usually, the data model of the ODS is very close to the OLTP source applications data model. Any source data should be accepted in the ODS and almost no data quality rule should be implemented. This ensures to have a store representing all the data-of-the-day from the operational systems.

When designing the Data Warehouse schema, the following general tips may be helpful:

- Use a modeling tool. (This sounds obvious, but how many of you still maintain SQL DDL scripts manually?)
- Where possible, describe columns inside the data dictionary of the database (for example use “COMMENT ON TABLE” and “COMMENT ON COLUMN” SQL statements). By doing so, you allow ODI to retrieve these comments and descriptions into its metadata repository.
- Design a storage space for the Staging Area where ODI will create its necessary temporary tables and data quality error tables.

- Do not use primary keys of the source systems as primary key for your target tables. Use counters or identity columns whenever it is possible. This makes a flexible data model that will easily evolve over time.
- Design referential integrity (RI) and reverse engineer foreign keys in ODI models. Do not implement these foreign keys in the target database as they may lead to performance issues. With the automatic data quality checks, ODI will guarantee the consistency of data according to these RI rules.
- Standardize your object naming conventions. For example, use three letters to prefix table names by subject area. Avoid using very long names as ODI may add a prefix to your table names when creating temporary objects (for example E\$_Customer will be the error table for table Customer).
- Use either 3rd Normal Form modeling (3NF) or dimensional modeling (“Snow Flakes” or “Star Schemas”) doesn’t have any impact on ODI, but rather on the way you will design your future business rules development and business users reports.

Once the ODS and DW are designed, you will have to create the appropriate models in Designer and perform reverse-engineering.

Version XX of ODI comes with a modeling tool called Common Format Designer. It can help you design the structure of the ODS and the DW from the existing structures of the source systems by simply drag and dropping tables columns and constraints. The generated Data Description Language statements (DDL) automatically include specific features of the target database. It also keeps track of the origin of source columns to automatically generate the interfaces for loading the target, thus resulting in significant time savings. Best practices on the usage of Common Format Designer are out of the scope of this book.

Specifying and Designing Business Rules

Usually this step starts almost at the same time as the design of the target schema. Understanding the business needs is clearly a key factor in the success of the overall project. This influences the key indicators selected for the warehouse as well as the rules used to transform source to target data. The more accurate the specification of a business rule, the easier the design of this rule in ODI. Several successful projects have used ODI Designer to specify the business rules, thus avoiding having discrepancies between a Word document and the final developed rules.

The following table summarizes what needs to be specified for every target table of the ODS or the DW before designing business rules.

Target Datastore:	Give the name of the target datastore, possibly preceded by a qualification prefix (such as the database or schema name)
Description of the transformation:	Give a short description of the purpose of the transformation
Integration strategy:	<p>Define how data should be written to the target. Examples:</p> <ul style="list-style-type: none"> • Replace content with new data from sources • Append incoming data onto the target • Update existing records and insert new ones according to an update key (specify the key here) • Use a Slowly Changing Dimension strategy. Specify the surrogate key, the slowly changing attributes, the updatable ones, the columns for start and end date etc. • A home-made strategy (for example: BASEL II compliance, SOX historical audit trail etc.) <p>Every strategy specified here will correspond to an ODI Integration Knowledge Module. Refer to section “SunopsisODI Knowledge Modules” for details.</p>
Refresh frequency:	Specify when this datastore should be loaded (every night, every month, every 2 hours etc)
Dependencies:	Give all dependencies in the load process (e.g. what datastores have to be loaded prior to the current one; what specific jobs need to be executed prior to this one etc.)
Source Datastores:	Give the list of source datastores involved for loading the target table. This list should include all the lookup tables as well. For each source datastore, include the following information.
• Source ODI Model	Give the name of the source model as it appears in Designer
• Datastore Name	Give the name of the datastore
• Purpose/Description	Indicate whether it is the main source set or a lookup one. If Change Data Capture is to be used, it should be indicated here.
Field mappings and transformations:	For every target field (or column), indicate what transformations need to be applied to the source fields. These transformation should be written as expressions or formulas as often as possible.
• Target Column	Give the target column name
• Mapping Description	Describe the purpose of the mapping

• Mapping Expression	Give the expression using the source column names. Try to express it in pseudo code.
Links or join criteria	For every pair of source datastores, specify the criteria to lookup matching records. This is often known as an SQL join.
• Datastore 1	Name of the first datastore
• Datastore 2	Name of the second datastore
• Link Expression	Give the expression used to link the 2 datastores in pseudo code
• Description	Describe this link by specifying if it is a left, right or full outer join.
Filters:	Give the list of filters that apply to source data. Filters should be expressed in natural language and in pseudo code whenever possible.
• Filter Description	Describe this Filter.
• Filter Description	Give the expression used to implement the filter in pseudo code.
Data Quality requirements:	List here all the data quality requirements including error recycling if appropriate. The requirements should be expressed as constraints whenever possible.
• Constraint Name	Name or short description of the constraint to validate.
• Description	Purpose of the data quality check
• Constraint Expression	Expression in pseudo code required to control the data

To illustrate the use of such a form, the following table gives an example for the business problem case study defined in the chapter “

Introduction to Oracle Data Integrator (ODI) Sunopsis E-LT and Business-Rule Driven Approach”:

Target Datastore:	Teradata Warehouse.SALES		
Description of the transformation:	Aggregate orders and order lines from the Oracle source production system.		
Integration strategy:	Append new records of the day		
Refresh frequency:	Every night		
Dependencies:	<p>The following tables need to be loaded prior to this one for referential integrity:</p> <ul style="list-style-type: none"> • PRODUCT • SALES_REP 		
Source Datastores:			
	Source ODI Model	Datastore Name	Purpose/Description
	Oracle Source	ORDERS	Orders from the production system
	Oracle Source	LINES	Order lines. Main source for aggregation
	User Data Files	CORRECTIONS	File containing the value to add to the amount sold if any manual correction has occurred for a particular order line.
Field mappings and transformations:			
	Target Column	Mapping Description	Mapping Expression
	ID	Order Id	ORDERS.ORDER_ID
	PRODUCT_PK	Product Id as it appears in the order lines	LINES.PRODUCT_ID
	SALES_REP	Sales Rep Id as it appears in the order	ORDERS.SALES_REP_ID
	SALE	Amount sold. If a correction for this amount exists in the “corrections” file, the value should be added. The total	SUM (LINES.AMOUNT + (CORRECTIONS.VALUE when it exists))

	value needs to be summed.		
QUANTITY	Total quantity of product sold.	SUM (LINES.QUANTITY)	
Links or join criteria			
Datastore 1	Datastore 2	Link expression	Description
ORDERS	LINES	Link orders and order lines on the order ID. Every order line must match an existing order	ORDERS.ORDER_ID = LINES.ORDER_ID
LINES	CORRECTIONS	Lookup a correction value from the correction file if it exists given the line item ID (left join)	LINES.LINE_ITEM_ID = CORRECTIONS.CORR_ID
Filters:			
Filter Description		Filter Expression	
Orders of the day		ORDER.ORDER_DATE between yesterday and now	
Orders validated		ORDER.STATUS = Closed	
Data Quality requirements:			
Constraint	Description	Constraint Expression	
Reference to SALES_REP	Ensure that every SALES_REP ID exists in the SALES_REP reference table	SALES_REP references SALES_REP(ID)	
Reference to PRODUCT	Ensure that every PRODUCT_PK exists in the PRODUCT reference table	PRODUCT_PK references PRODUCT(ID)	
ID Not null	ID Column is mandatory	Check ID not null	
Quantity greater than 0	The quantity sold must always be positive	QUANTITY > 0	
Uniqueness of sale	A sales rep can't sell the same product twice inside the same order	(PRODUCT_PK, SALES_REP) is unique	

The next step is to design these business rules in Designer. The translation from the specification to the design is straightforward. Every specification of the loading of a target table will be converted into an interface. This process is even faster if the specification is made directly using Designer.

The steps to implement the interface are typically as follows.

- Drag and drop the target datastore
- Drag and drop source datastores
- For every target field, translate the pseudo code of the transformation into SQL expressions. When possible, decide where to execute the expression (source, staging area, target)
- For every join, define the join in the source panel of the interface. Depending on the specification, define the join as inner, left, right, or full outer join. When possible, try to execute joins on the sources to minimize network traffic.
- For every filter, translate the pseudo code into an SQL expression. When the source database accepts SQL filters, set it as “execute on source” to minimize network traffic.
- In the flow diagram of the interface, define the loading strategies (LKM) and specify the integration strategy (IKM) to match the specification. Choose the appropriate options for the Knowledge Module and activate the data quality control when appropriate.
- In the control tab of the interface, select the constraints to control. If the constraints are not defined on the target datastore, define them in the Models view of Designer.

After the design of every interface, you should test it by executing it. The ODI Operator allows you to easily follow the execution of your interface to get the number of records processed and many other useful indicators (such as the generated code, the elapsed time, the number of inserts, updates, deletes etc.) After the interface completes, the resulting data and errors can be directly selected from Designer to validate the accuracy of the business rules.

Thanks to ODI Contexts, the execution of the interface will happen on a “Development” environment and will not affect production data.

Building a Data Quality Framework

With an approach based on business rules, ODI is certainly the most appropriate tool to help you build a data quality framework to track data inconsistencies. Thanks to the Check Knowledge Modules, you simply have to define the control business rules, and inconsistent data is automatically isolated for you in error tables. However, isolating erroneous data is not the only issue in Data Quality. Even if

ODI automatically detects duplicated keys, mandatory fields, missing references, and more complex constraints, you will need to involve business users in the process of qualifying data discrepancies and make the appropriate decisions. You should have an answer to each of the following questions before defining your data quality strategy:

- What level of data quality is required for the Data Warehouse?
- Who are the business owners of source data?
- What should we do with rejected records?
- Do we need to define an error recycling strategy?
- Do we need to involve business owners of the data and report rejected records?
- How would business users modify erroneous source data?
- Do we provide a GUI to have them modify erroneous records in the error tables?

Document “Technical Analysis - Data Quality in ODI” further discusses Data Quality issues and the recommended best practices.

Developing Additional Components

Not all the typical tasks of the loading of the Data Warehouse can be achieved using ODI Interfaces. It is usual to develop additional components or jobs that would carry on tasks such as, for example:

- Receiving and sending e-mails
- Copying, moving, concatenating, renaming files in the file system
- Compressing, decompressing files
- Executing web services
- Writing and executing shell scripts for a specific operating system
- Writing and executing java programs
- and so on.

These components can be developed and tested within Designer as Procedures, Variables, User Functions, or Steps in a package. ODI Procedures offer a broad range of possibilities for developing these components. They can include, for example:

- Any ad-hoc SQL statement to any database
- Any Operating System call
- ODI built-in tools and APIs (Send Mail, Read Mail, Copy File etc.)

- Any scripting language supported by IBM's Bean Scripting Framework. This includes Java, Java Script, Python, Perl, NetRexx, etc.

Of course, the risk here is to start developing transformation processes as Procedures by hand coding shell and SQL scripts rather than using the powerful mechanism of ODI Interfaces combined with Knowledge Modules. To avoid that, try as much as possible to specify your transformations as business rules and not as a technical processes. ODI Procedures should always be considered as technical steps that have to be achieved in the overall process, but they shouldn't have in-depth business logic that applies to the data. Typical Data Warehouse projects would have less than 10% of development in the form of Procedures.

Packaging and Releasing Development

Now that the Interfaces, Procedures and Variables are developed and tested, they need to be ordered as steps within Packages. Start thinking about what should happen in case of error in any of these steps. By default, the ODI Agent will stop the execution of a package at the step that has failed and it will rollback any open transaction on all the connected databases. Even though errors are reported in ODI Logs, it is always a good practice to have an "on-error" procedure that is triggered if some of the steps fail. This procedure can for example send an email to an operator to warn him that the package has failed, reporting the session ID.

Particular attention has to be paid to the following step attributes:

- Next step on failure
- Number of attempts in case of failure
- Interval between attempts

Try to avoid heavy loops inside packages (more than 50 iterations). In most of the cases, a loop can be avoided by simply adding a table as source to an interface!

When the package is successfully tested, you will have to release it as a Scenario. This Scenario will then be imported in the Test environment before validation for Production.

Versioning Development

Before going to production, you should record a stable version of all the objects involved in the release of the Scenarios. Creating versions of objects will allow you restore previous released items for further maintenance in case of problem.

From version XX of ODI, versions of objects are stored in the Master Repository and the dependencies between objects are maintained in objects called Solutions. To make sure all the objects involved in the release are consistently versioned, it is thus recommended that you create a Solution object and add the Project to this

Solution. Designer calculates all the dependencies between objects and automatically creates a new version for Projects, Models, Scenarios and global objects used. Restoring these objects from a previous version can be done when needed from Designer by simply selecting the Restore menu.

Section “Using SunopsisODI Version Management” further discusses the best practices for version control.

Scheduling and Operating Scenarios

Scheduling and operating scenarios is usually done in the Test and Production environments in separate Work Repositories. ODI Operator provides a graphical interface to perform these tasks. Any scenario can be scheduled by an ODI Scheduler Agent or by any external scheduler, as scenarios can be invoked by an operating system command.

When scenarios are running in production, agents generate execution logs in an ODI Work Repository. These logs can be monitored either through the Operator GUI or through any web browser when Metadata Navigator is setup.

Operating teams need to be trained on the use of ODI Operator or Metadata Navigator. They will have to monitor logs, restart failing jobs and submit ad-hoc tasks for execution.

Monitoring the Data Quality of the Data Warehouse

Business users rely on the accuracy of key indicators of the Data Warehouse to make decisions. If these indicators are wrong, the decisions are worthless.

Depending on the data quality strategy you have defined, business users will actively participate in the monitoring of data discrepancies. They will have to help the IT team to better refine the calculations of the indicators as well as the qualification of the erroneous data. This generally leads to modifications of the business rules. These updates are done in the development environment and follow the normal release cycle defined for the project.

Typical projects report data quality errors to business users through reports generated from ODI'S error tables. These reports are pushed to the data owner in their email system or to any workflow application used internally.

Publishing Metadata to Business Users

Business users usually need to have access to other data quality indicators based on metadata such as:

- When was the last time my table was updated?
- How many records were added, removed or updated in my table?
- What are the rules that calculate a particular indicator?
- Where does the data come from, and how is it transformed?

- Where does the data go to, and how is it transformed?
- Etc.

All these questions can be answered if you give access to Metadata Navigator to business users. This is done simply by assigning them a user ID in Security Manager. The web-based interface, allows them to see all the data models, and the interactions between them. This includes:

- Flow Maps
- Data Lineage, which is usefully for understanding the path taken by data and the transformations between applications
- Execution logs with accurate statistics on the number of records processed (number of inserts, updates, delete and errors).

Planning for Next Releases

A Data Warehouse evolves as the business need of the enterprise change. This leads to frequent updates of the data models. This constant changing has an impact on the development of new components and the maintenance of existing ones. However it is important to understand that it should always be a cycle driven by business requirements. The general steps listed below give you a brief overview of how this could be planned:

1. Define or refine Business Rules specifications
2. Make sure previous releases of the objects are properly versioned and can be restored safely in a new empty work repository before modifying anything in your current work repository.
3. Define new sources or targets in Topology Manager.
4. Reverse-engineer new and/or existing models. Use ODI Cross References to evaluate the impact of the changes to the structure of source and target fields. Pay special attention to fields that were removed from tables as they may still be used in some transformations.
5. Develop new interfaces and update existing ones. Create or update additional components. Test every item separately in the development environment.
6. Update existing packages or create new ones. Test every package, even those that haven't changed since the last release, as the underlying schema may have changed.
7. Regenerate the new scenarios for your next release.
8. Release the scenarios into each Test environment. Correct any bug that would be reported from the Test process. Involve business users in the test process.

9. Create new Solutions and version every project and model. Use the change indicators that appear on every object to detect which objects should be versioned.
10. When acceptance tests are successful, release your new scenarios into production.

DEFINING THE TOPOLOGY IN ORACLE DATA INTEGRATOR

Introduction to Topology

The Topology describes the physical and logical architecture of your Information System. It gives you a very flexible way of managing different servers, environments and agents. All the information of the Topology is stored in ODI Master Repository and is therefore centralized for an optimized administration. All the objects manipulated within Work Repositories refer to the Topology. That's why it is the most important starting point when defining and planning your architecture.

Typical projects will have separate environments for Development, Test and Production. Some project will even have several duplicated Test or Production environments. For example, you may have several production contexts for subsidiaries running their own production systems (Production New York, Production Boston etc). There is obviously a difference between the logical view of the information system and its physical implementation as described in the figure below.

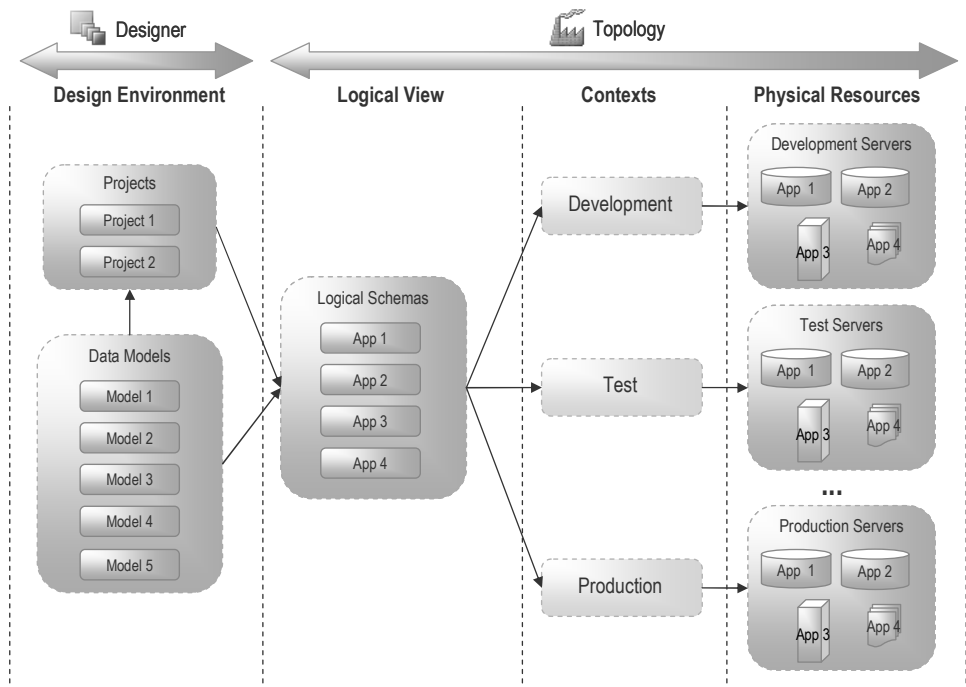


Figure 12: Logical and Physical View of the Infrastructure

The logical view describes logical schemas that represent the physical schemas of the existing applications independently of their physical implementation. These logical schemas are then linked to the physical resources through contexts.

Designers always refer to the logical view defined in the Topology. All development done therefore becomes independent of the physical location of the resources they address. At runtime, the logical information is mapped to the physical resources,

given the appropriate contexts. The same scenario can be executed on different physical servers and applications simply by specifying different contexts. This brings a very flexible architecture where developers don't have to worry about the underlying physical implementation of the servers they rely on.

This chapter gives best practices for building a robust and flexible Topology.

Data Servers

Understanding Data Servers and Connectivity

Data servers describe connections to your actual physical application servers and databases. They can represent for example:

- A Teradata System
- An Oracle Instance
- An IBM DB2 Database
- An IBM DB2 iSeries Instance
- A Microsoft SQL Server Instance
- A Sybase ASE Server
- A Websphere MQ router
- A File System
- An XML File
- A Microsoft Excel Workbook
- etc.

At runtime, ODI Agent uses the connection information you have described to connect to the servers. ODI Agent and ODI Designer can use two ways for addressing a particular physical data server:

- JDBC – Java Database Connectivity. This is a Java API that provides access to most popular databases or to other tabular data sources. For more information about JDBC, you can visit <http://java.sun.com/products/jdbc>. JDBC is a flexible API that provide 4 different ways to connect to a data server:
 - Type 4 drivers: These convert JDBC calls to direct network calls to the database servers without the need of any middle-tier component. Oracle recommends the use of these types of drivers whenever possible as these don't require any particular operating-system-dependent libraries or middleware. They provide great flexibility for your architecture. Most popular databases on the market, such as Teradata, Oracle, Microsoft SQL Server, IBM DB2, Informix, Sybase, etc. propose Type 4 drivers.
 - Type 3 drivers: These connect to a middle-tier server (middleware) using a

proprietary protocol. The middleware translates the calls to the final files or database servers and returns the results to the client program. Type 3 drivers are usually used to access database systems that were not designed in a Client/Server model, such as IMS, VSAM, ADATABASE, etc.

- Type 2 drivers: These rely on operating system specific libraries provided by the database vendors. They convert JDBC calls to native library calls, and are therefore operating system dependant. When using such drivers with ODI, you need to pay attention to the operating system on which ODI Agent is installed. You should also make sure that all the client side libraries provided by the database vendor are properly installed on the system.
- Type 1 drivers: These provide a bridge via ODBC (Open Database Connectivity). ODBC binary libraries, and in many cases, database client libraries, must be loaded on the Agent machine. These drivers should be considered only when no other solution for accessing the data server is available.
- JNDI – Java Naming Directory Interface. This API provides unified access to a number of naming and directory services. ODI uses this API to build connections to JMS (Java Message Services) middleware such as IBM Websphere MQ, Tibco JMS, Sonic MQ, etc. For more information about JNDI and JMS, you can visit <http://java.sun.com/products/jndi> and <http://java.sun.com/products/jms>

ODI uses JDBC drivers to connect to the following data sources:

- Relational Databases Servers such as Teradata, Oracle, IBM DB2, Microsoft SQL Server, Sybase, Informix, MySQL, PostgreSQL, etc
- Desktop Databases including Microsoft Access, Hypersonic SQL, Dbase, FoxPro, Microsoft Excel, etc.
- ASCII or BINARY files when using ODI'S JDBC Driver for Files. This requires the file to be loaded using an ODI Agent. This driver is not used when the Agent calls native database loaders such as FastLoad, MultiLoad, SQL*LOADER, BCP, etc.
- XML files, by using ODI JDBC Driver for XML. Any XML file structure is converted by this driver into a relational database structure (either in memory or persistently in any other database)
- LDAP directories, when using ODI Open Connector for LDAP.
- Non-relational Databases such as IMS, ADATABASE, DATACOM, VSAM, etc, by using third-party adapters (iWay, Attunity, Hit Software are examples of solution providers)

Oracle Data Integrator uses JNDI essentially to connect to JMS providers.

Defining User Accounts or Logins for ODI to Access your Data Servers

At runtime, ODI Agent will need a user name and password to connect to data servers. A good practice is to create a dedicated login on every data server that needs to be accessed by ODI. Using a single login for every server brings administration benefits as privileges can be centralized by the database administrator. Usually every data server has several schemas (or databases or catalogues or directories or libraries) under which the datastores (tables, views, files, queues etc.) are stored. It is recommended that ODI Agent uses a single connection for every data server, as discussed later. Therefore, you should follow these rules when creating these user accounts:

- For every source data server that should be accessed in “read-only” mode, create a user login and make sure this login has “read” access rights on all the schemas and datastores that are needed in the project. If you plan to use operating system calls to unload data (for example for performance issues) the ODI login should also be granted the rights to use any unload utility on the source server.
- For every data server that should be accessed in “write” mode, create a user login with “write” privileges on all the target tables/files/queues. If you plan to use operating system calls to load data (for example for performance issues) this ODI login should also be granted rights to use any load utilities on the target server.
- If you need to use ODI to create a target schema or rebuild indexes, or create any particular objects at runtime (such as views, or database links etc.), the ODI login should be granted the appropriate rights.
- For every source or target, set the staging area’s dedicated storage area (see next section) as the default storage area for this login. You should grant this login full privileges on the staging area.

ODI stores encrypted versions of the passwords for all data servers in ODI’S Master Repository. It is very important to secure access to the database hosting the ODI Master Repository.

Defining Work Schemas for the Staging Area

ODI Knowledge Modules often use temporary objects to stage temporary data for jobs optimization. These temporary objects are always stored in a particular schema or directory called the Work Schema (or staging area).

On the target data servers, you should almost certainly create a dedicated work schema. This will store:

- Temporary tables and views created during the loading phase if the source data is not in the same server as the target (C\$ tables)
- Temporary tables created during the integration phase when using certain Integration Knowledge Modules (I\$ tables)
- Permanent Error tables created during Data Quality Control phase when using Check Knowledge Modules (E\$ tables)
- Permanent Journal tables, views and triggers created when the target is to be used as a source with Change Data Capture enabled by using Journalizing Knowledge Modules (J\$ objects)

On “read-only” source data servers the following objects may be created depending on the way you want to address these servers:

- Views during Load phases, when using certain Loading Knowledge Modules
- Error tables when performing a “static” Data Quality Control (e.g. when controlling the data of a particular source schema independently of any load job)
- Journal tables, views and triggers when using Change Data Capture on the sources.

Defining the Data Server

When converting an interface to code, ODI uses data servers’ definitions to determine the loading phases. Having 2 data servers defined pointing to a single physical data server may cause extra loading phases that are not needed. We therefore recommend defining a single data server per physical server, regardless of the number of physical schemas that need to be accessed on this data server. For example, if you have a single Teradata system, only one data server should be defined in Topology Manager.

To define your data server, you can follow these steps:

1. Get the database or application administrator to create a login (user name and password) for this data server.
2. Create a work schema to store temporary objects and assign this schema as the default for the login previously created. Set the size of this schema in relation to the size of the temporary tables that will be created. For example, for a target server, the size of this schema should be at least twice the size of the largest concurrent transactions running in parallel.
3. Make sure you can successfully connect to the data server from any ODI client or Agent machine, using the client tools.

4. Copy the JDBC or JMS or any Java libraries required to connect to the data server to the “**drivers**” directory of the ODI installation. (see ODI online help for details)
5. Test your connection using a pure JDBC client such as Squirrel-sql (shipped and installed by default with ODI)
6. In Topology Manager, create a new data server as follows:
 - a. User Name and password: Use the login information for the specific login that you have created for ODI.
 - b. Define the JDBC or JNDI driver (refer to the documentation of your driver)
 - c. Define the URL (use hostnames rather than IP addresses whenever possible)
 - d. Test your connection locally and for every agent you have defined
 - e. Refer to ODI's online documentation library for advanced options
7. The name you give to the data server is very important. Please refer to the Object Naming Conventions paragraph for details.

Examples of Data Servers Definitions

The tables below give information on how to define some of the most common data servers. Every JDBC or JMS driver may propose additional URL parameters. Please refer to their specific documentation for further details.

Teradata

Name Example:	TERADATA_MYSERVER
Physical Usage:	A single data server definition is equivalent to a single Teradata system
Driver Files:	<ul style="list-style-type: none"> • log4j.jar; terajdbc4.jar (TTU 7.x) • tdgssconfig.jar, tdgssjava.jar, terajdbc4.jar (TTU 8.x)
Driver Type:	JDBC Type 4
Driver Vendor	NCR
Driver Name:	com.ncr.teradata.TeraDriver
Driver URL:	jdbc:teradata://<Teradata_system_ip_name>
Server Name Usage:	The server name may be used in some Knowledge Modules that generate BTEq, FastLoad, Multiload or Fastexport scripts. It should therefore indicate the host name as it appears in the hosts file (without the COPn postfix).

Oracle

Name Example:	ORACLE_MYSERVER
Physical Usage:	A single data server definition is equivalent to a single Oracle Instance
Driver Files:	ojdbc4.jar
Driver Type:	JDBC Type 4
Driver Vendor	Oracle Corp.
Driver Name:	oracle.jdbc.driver.OracleDriver
Driver URL:	jdbc:oracle:thin:@<hostname>:<port>:<sid_or_servicename>
Server Name Usage:	The server name may be used in some Knowledge Modules that generate database links or sql*loader scripts. It should therefore indicate the instance name as it appears in the tnsnames.ora file.

Microsoft SQL Server

Name Example:	MSSQL_MYSERVER
Physical Usage:	A single data server definition is equivalent to a single Microsoft SQL server Instance
Driver Files:	Gate.jar
Driver Type:	JDBC Type 4
Driver Vendor	Inet Software
Driver Name:	com.inet.tds.TdsDriver
Driver URL:	jjdbc:inetdae7:<hostname>:<port>
Server Name Usage:	The server name may be used in some Knowledge Modules that generate linked servers aliases or bcp scripts. It should therefore indicate the machine name as it appears in the Microsoft network.

IBM DB2 UDB (v8 and higher)

Name Example:	DB2UDB_MYSERVER
Physical Usage:	A single data server definition is equivalent to an IBM DB2 UDB database.
Driver Files:	db2jcc.jar

Driver Type:	JDBC Type 4
Driver Vendor	IBM Corp.
Driver Name:	com.ibm.db2.jcc.DB2Driver
Driver URL:	jdbc:db2://<hostname>:<port>/<database_name>
Server Name Usage:	The server name may be used in some Knowledge Modules that generate UNLOAD or LOAD scripts. It should therefore indicate the alias of the database as defined for the client machine (DB2 CLI).

IBM DB2 UDB (v6, v7) and IBM DB2 MVS

Name Example:	DB2UDB_MYSERVER
Physical Usage:	A single data server definition is equivalent to an IBM DB2 UDB database.
Driver Files:	db2java.zip; <ul style="list-style-type: none"> - All IBM Connect operating system libraries should be registered in the PATH of the host machine - The java library path refers to the path of the IBM Connect libraries (use option – Djava.library.path= <IBM_InstallPath/SQLLIB/BIN)
Driver Type:	JDBC Type 2
Driver Vendor	IBM Corp.
Driver Name:	COM.ibm.db2.jdbc.app.DB2Driver
Driver URL:	jdbc:db2:<database_alias>
Server Name Usage:	The server name may be used in some Knowledge Modules that generate UNLOAD or LOAD scripts. It should therefore indicate the alias of the database as defined for the client machine (DB2 CLI).

IBM DB2 400 (iSeries)

Name Example:	DB2400_MYSERVER
Physical Usage:	A single data server definition is equivalent to a single iSeries Server.
Driver Files:	jt400.zip

Driver Type:	JDBC Type 4
Driver Vendor	IBM Corp.
Driver Name:	com.ibm.as400.access.AS400JDBCdriver
Driver URL:	jdbc:as400://<iSeries_hostname>
Server Name Usage:	The server name may be used in some Knowledge Modules that generate specific iSeries commands. It should therefore indicate the hostname of the iSeries machine.

Flat Files (Using ODI Agent)

Name Example:	FILE_GENERIC
Physical Usage:	A single data server definition should be created for ALL files regardless of their location. The files will be accessed by ODI Agent.
Driver Files:	Built-in (Included in sunopsis.zip)
Driver Type:	JDBC Type 4
Driver Vendor	Oracle
Driver Name:	com.sunopsis.jdbc.driver.FileDriver
Driver URL:	jdbc:snps:file
Server Name Usage:	None

XML

Name Example:	XML_FILEALIAS
Physical Usage:	A single data server definition should be created for every XML schema or file. XML files will be accessed by ODI Agent. Refer to the ODI XML Driver for JDBC online documentation.
Driver Files:	snpsxmlo.jar; jdbcxkey.xml
Driver Type:	JDBC Type 4
Driver Vendor	Oracle
Driver Name:	com.sunopsis.jdbc.driver.xml.SnpsXmlDriver
Driver URL:	jdbc:snps:xml?f=<filename>&s=<schema>[&<property>=<value>...]

Server Name Usage:	None
-----------------------	------

Microsoft Excel

Name Example:	EXCEL_FILEALIAS
Physical Usage:	A single data server definition should be created for every Microsoft Excel work book. Microsoft Excel files are accessed using ODBC drivers. Make sure you create an ODBC DSN for the Microsoft Excel workbook.
Driver Files:	Built-in JDBC/ODBC bridge provided by Sun Microsystems
Driver Type:	JDBC Type 1
Driver Vendor:	Sun Microsystems
Driver Name:	sun.jdbc.odbc.JdbcOdbcDriver
Driver URL:	jdbc:odbc:<ODBC_DSN>
Server Name Usage:	None

Physical Schemas

Physical schemas indicate the physical location of the datastores (tables, files, topics, queues) inside a data server. When accessing a JDBC data server, these physical schemas are often proposed to you in the schema creation window of the Topology module. All the physical schemas that need to be accessed have to be registered under their corresponding data server. Physical schemas are used to prefix object names when generating code (qualified names).

When creating a physical schema, you need to specify a temporary, or work schema that will store temporary or permanent object needed at runtime. You should select the temporary schema to dedicate to ODI when defining the data server (see Defining Work Schemas for the Staging Area). The temporary or permanent objects that may be created by ODI are summarized below:

- Temporary Loading tables created by the Loading Knowledge Modules – LKM (usually prefixed by C\$)
- Temporary Integration tables created by the Integration Knowledge Modules – IKM (usually prefixed by I\$)
- Permanent Error table when using Data Quality Knowledge Modules – CKM (usually prefixed by E\$)
- Permanent Journalizing tables, views and triggers when using Change Data Capture (or Journalizing) Knowledge Modules– JKM (usually prefixed J\$,

JV\$ and T\$)

Note: the prefix of the temporary objects can be changed. It is recommended that you use different prefixes when 2 datastores located in 2 different schemas have the same name.

For every data server, you need to define a **default** physical schema. The work schema in this default schema will be used by ODI to store permanent objects for the data server (such as the table summarizing the errors – SNP_CHECK_TAB or the subscribers' table when using CDC – SNP_SUBSCRIBERS).

The table below gives a description of the meaning of the physical schema concept for the most commonly used technologies:

Technology	Schema meaning	Example of a qualified object name
Teradata	Database or user	db_prod.Client
Oracle	Schema or user	SCOTT.Client
DB2 UDB or MVS	Schema	DB2ADMIN.Client
DB2 for iSeries	Library or Collection	LIBPROD.Client
Microsoft SQL Server or Sybase ASE	Database AND Owner	Northwind.clark.Client
IBM Informix	Database	Db_ifx:Client
Sybase IQ	Owner	DBA.Client
Flat Files	Directory location of the file	/prod/data/Client.dat
XML, MS Excel, MS Access, Desktop databases	Not Appropriate, use a <Default> schema	Client
JMS Topics	Sub topics	SubTopic1.SubTopic2.Client

Contexts

Contexts are used to group physical resources belonging to the same environment. Once created in Topology, you should avoid deleting or modifying the code of a context as it may be referenced in one or more work repositories. A typical project will have very few contexts (less than 10). These can be for example:

Context Name	Context Code	Description
--------------	--------------	-------------

Development	DEV	Servers used for development.
Unit Tests	UTEST	Servers used for testing interfaces on larger volumes for optimization.
Qualification	QTEST	Servers shared with business users for better qualifying business rules
Acceptance Tests	ATEST	Servers used for final acceptance tests
Production 1	PROD1	Production servers for location 1
Production 2	PROD2	Production servers for location 2
Etc.		

It is important to have the same number of data servers and schemas defined within every context. For example, if you have 2 target servers in production and only one target server for development, you should probably split the definition of the target server for development in 2 data servers in the physical architecture definition as shown in figure below:

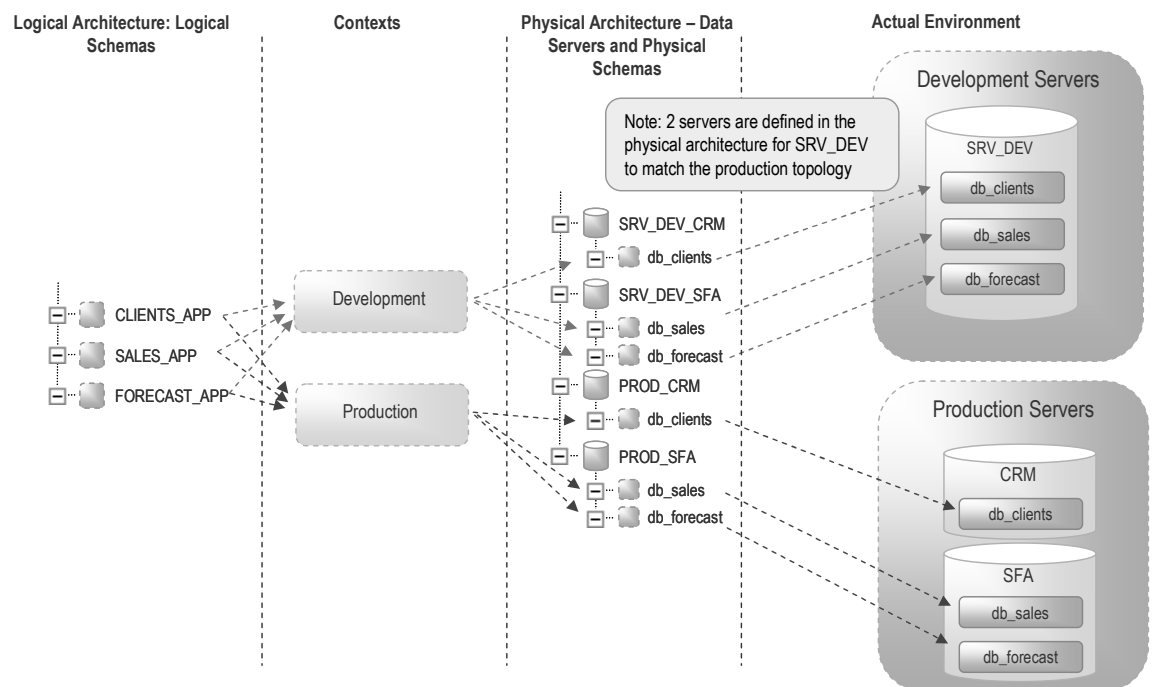


Figure 13: Example of a complex Topology definition

From this example, we can see that server SRV_DEV needs to be defined twice in the physical architecture to match the actual physical environment for production. Therefore data servers' definitions SRV_DEV_CRM and SRV_DEV_SFA both refer to the same server SRV_DEV. By doing so, you force ODI to consider

SRV_DEV as 2 separate servers when generating code in the Development context. This code will be valid when running on the production servers.

Logical Schemas

A logical schema is an alias that allows a unique name to be given to all the physical schemas containing the same datastore structures. The aim of the logical schema is to ensure the portability of procedures and models on different physical schemas at runtime. A logical schema can have one or more physical implementations on separate physical schemas, but they must be based on data servers of the same technology. A logical schema is always directly linked to a technology. To be usable, a logical schema must be declared in a context. Declaring a logical schema in a context consists of indicating which physical schema corresponds to the alias - logical schema - for this context.

At runtime, the agent will use the following formula to find the physical resources:

Logical Schema Name + Context = 1 Physical Schema (and 1 Physical Data Server as well)

Physical and Logical Agents

A physical agent is an ODI Java service that runs on a host as a listener on a TCP/IP port. It can execute your scenarios when it is invoked from one of the ODI GUI modules or schedule these executions when it is started as a scheduler. You can find more information about how to setup agents and schedulers in the

Architecture Case Studies section. A physical agent should have a unique name in Topology Manager. It is referenced by its host's IP address or name and the TCP port it is listening on.

A physical agent can run multiple sessions in parallel (multi-threaded). For optimization purposes, it is recommended that you adjust the maximum number of concurrent sessions it is allowed to execute. When this maximum number is reached, any new incoming session will be queued by the agent and executed later when other sessions have terminated. If you plan to run multiple parallel scenarios, you can consider load balancing executions as described in section "Setting up Agents".

Physical agents are part of the physical architecture. Therefore, you can choose to have different agents according to your environment and contexts. To do so, you define one logical agent and link it to several physical agents according to the context. When starting a session, you simply indicate the logical agent and the context, and ODI will translate it to the physical address of the agent.

The Topology Matrix

As Topology defines the final location of data servers, schemas and agents, it is important to have a detailed idea of where the resources are in all environments. It is recommended that you have a document summarizing all required resources and

their logical mapping in Topology. An example of such a document - called a Topology Matrix – is given in the table below:

Context	<Name of the Context>
Physical Data Server Information:	
Technology:	Technology of the data server
Name in ODI:	Name of the data server as it should be declared in Topology
URL:	URL for this server (usually composed of an IP address, a port and eventually a database)
Login Name Dedicated to ODI:	Account used to connect ODI to this data server
Default schema name for the Staging Area:	Optional ODI storage space for the staging area
<i>Physical Schemas / Logical Schemas mapping:</i>	
Physical Schema	Logical Schema
Physical schema 1	Corresponding Logical Schema
Etc.	
Physical Data Server Information:	
Etc.	
Context	<Name of the Context>
Etc.	

The example given in Figure 13 can be defined as follow:

Context	Development
Physical Data Server Information:	
Technology:	Teradata
Name in ODI:	TERADATA_SRV_DEV_CRM
URL:	devhost_srvdev
Login Name Dedicated to ODI:	sunopsis

Default schema name for the Staging Area:		Sunopsis_db
<i>Physical Schemas / Logical Schemas mapping:</i>		
Physical Schema	Logical Schema	
db_clients	TERADATA_CLIENTS_APP	
Physical Data Server Information:		
Technology:	Teradata	
Name in ODI:	TERADATA_SRV_DEV_SFA	
URL:	devhost_srvdev (defined twice to match the production environment)	
Login Name Dedicated to ODI:	sunopsis	
Default schema name for the Staging Area:	Sunopsis_stg	
<i>Physical Schemas / Logical Schemas mapping:</i>		
Physical Schema	Logical Schema	
db_forecast	TERADATA_FORECAST_APP	
db_sales	TERADATA_SALES_APP	
Context	Production	
Physical Data Server Information:		
Technology:	Teradata	
Name in ODI:	TERADATA_CRM_PROD	
URL:	prod_crm	
Login Name Dedicated to ODI:	Sunopsis_prod	
Default schema name for the Staging Area:	Sunopsis_stg_db	
<i>Physical Schemas / Logical Schemas mapping:</i>		
Physical Schema	Logical Schema	
db_clients	TERADATA_CLIENTS_APP	
Physical Data Server Information:		

Technology:	Teradata
Name in ODI:	TERADATA_SFA_PROD
URL:	Prod_sfa
Login Name Dedicated to ODI:	Sunopsis_prod
Default schema name for the Staging Area:	Sunopsis_stg_db
<i>Physical Schemas / Logical Schemas mapping:</i>	
Physical Schema	Logical Schema
db_forecast	TERADATA_FORECAST_APP
db_sales	TERADATA_SALES_APP

Object Naming Conventions

Topology defines the core information about your information system. This information is stored in the ODI Master Repository and is referenced in several Work Repositories. When naming objects in Topology, you should follow a strict convention to facilitate maintenance. The following are some suggestions for naming objects:

Object Type	Naming Rule	Examples
Physical Data Server	We recommended that you prefix any data server name with the technology it belongs to. The name should be a meaningful name such as the one used usually to reference this server. Put the name in uppercase. <TECH_NAME>_<SRV_NAME>	TERADATA_EDW ORACLE_CRM_DEV DB2UDB_SFA_TST1 XML_CLIENT001
Logical Schema	We recommended that you prefix any logical schema name with the technology it belongs to. The name should be a meaningful name such as the one used usually to reference applications, schemas or business areas. Put the name in uppercase. <TECH_NAME>_<SCHEMA_NAME>	TERADATA_STG_LEVEL1 TERADATA_MSTR_DATA ORACLE_CRM_MAIN ORACLE_CRM_SALES ORACLE_CRM_FRCST
Context	You should use a meaningful name for the context. However, the context code should be in uppercase. Avoid spaces and special characters in the context code.	Name: Development Ctx Code: DEV

	Name: <Context Name> Code: <CONTEXT_CODE>	Name: Production NY Code: PROD_NY
Physical Agent	<p>A physical agent is identified by the host name and the TCP port it is listening on. Try to define your agent names in uppercase. Avoid spaces and special characters in the agent name.</p> <p><HOST>_<PORT></p>	AS400DEV_29000 WXFRAUN2345_PROD_29000 WXFRAUN_DEV2345_29010 LXWWWMAIN_29020
Logical Agent	<p>Logical agent names should be meaningful according to the physical agents they refer to. Define the name in uppercase and avoid any spaces or special characters.</p> <p><LOGICAL_NAME></p>	WXFRA2345 LXWWWMAIN

DEFINING ODI MODELS

Introduction to Models

One of the key issues of your data warehouse project is to understand and locate the datastores (data structures) you need for building key indicators for business users. These datastores, located on a logical schema, are usually composed of fields (or columns), linked to each other through referential integrity rules and constrained by uniqueness rules and other field relationships and constraints.

ODI Models represent the data models of your Information System. They refer to a set of datastores on a logical schema. All the mappings and transformations will refer to data models' datastores fields and constraints.

ODI does not distinguish between source and target models. Models are stored in the Work Repository and are the basis for building business rules. They centralize the core **metadata** of your systems. Models can be versioned as described further in this document.

Datastores in a model can be organized in sub-models. As of version 4.1, models can be grouped in folders.

As explained in the introduction to this document, a **datastore** is usually one of :

- a table stored in a relational database
- an ASCII or EBCDIC file (delimited, or fixed width)
- a node from an XML file
- a JMS topic or queue from a Message Oriented Middleware IBM Websphere MQ
- a node from an LDAP directory
- an API that returns data in the form of an array of records

This abstraction allows ODI to connect theoretically to any source of data.

Models referring to a Relational Database can be automatically populated from the tables and views definition of the RDBMS through the JDBC API. For other models, you will have to define the metadata manually or write a specific reverse-engineering Knowledge Module to populate the ODI Work Repository from a metadata provider.

Models are linked to the logical architecture you have defined in the topology through the logical schema that you have selected. You can therefore develop rules safely using models' metadata without worrying about their physical implementation.

Notes:

Models should be administered by Metadata Administrators as described in section Organizing the Teams.

Although you can change the technology or the logical schema of a model, you should exercise extreme caution when doing this.

Model Contents

Models can contain the following objects:

- **Sub-models:** Sub-models are folders inside a model that you can use to organize your datastores.
- **Datastores:** A datastore represents a set of formatted records from a logical schema of a data server
- **Columns:** Columns represent attributes or fields belonging to a datastore. They are typically defined with a unique column name, a short description, a position, a data type, a length and scale, a mandatory (not null) indicator, and other data checking attributes.
- **Constraints:** Constraints define additional data quality business rules that apply at the data level for the datastore. When getting metadata from an RDBMS, some of the constraints defined in the physical schema are imported. You can add your own constraints on the datastore for data quality control purposes without altering the database schemas. The following types of constraints can be reverse-engineered (reversed) into ODI.
 - **Primary Key:** A primary key defines the list of columns used to uniquely identify a record in a datastore. it is recommended that you define a primary key for every datastore of your model. Columns participating in the primary key should be defined as mandatory.
 - **Unique Keys (Alternate Keys):** A unique key defines an alternative list of columns that can uniquely identify a record. Columns belonging to a unique key can contain null values. Defining unique keys can help enforcing the uniqueness of records according to different criteria than the primary key (for example, controlling that the (customer_name + phone_number) are unique in the customer table identified by the customer_id)
 - **Indexes:** An index defines a set of columns indexed in the underlying database. Indexes are defined for information purpose only. They are used in v4.1 with Common Format Designer when creating the physical definition of the table in the database.
 - **References (Foreign Keys):** A reference define a relationship between 2 datastores to enforce referential integrity. When

foreign keys are not implemented in the physical schema, it is recommended that you define them manually for the model and have ODI perform data quality control during integration. ODI has also introduced a complex reference type allowing you to define a link between 2 datastores that should match a complex SQL expression rather than the equality between foreign key fields and primary key referenced fields. This type of reference is very useful when dealing with models not in 3rd Normal Form where column values can have several meanings. An example of using a complex reference could be: “check that the 3 first characters in the address field of the customer table match the country code key of the country table”.

- **Conditions:** A condition defines an expression that a record must match to be considered valid. A condition can be implemented in the database as a check constraint. By defining conditions you can enforce the integrity of fields within the same record during the integration phase. You can use conditions, for example, to check the allowed domain values for a column or to compare the value of a column to other columns.
- **Filters:** A filter defines a default filtering expression used when the datastore is a source of an ODI interface. When the datastore is dragged and dropped in the interface, the filter condition becomes a filter in the interface.

Note: For readability, try to avoid having more than 200 datastores per model.

Importing Metadata and Reverse-engineering

Introduction to Reverse-engineering

Capturing metadata from a metadata provider into a model is called “Reverse-engineering” in ODI. When creating a model, you need to specify its technology, its logical schema and decide which reverse-engineering strategy you want to use.

ODI has 3 different ways to define metadata:

- **Standard Reverse-engineering using the JDBC API:** This mode is fast and useful when the underlying schema resides on a relational database, and when the JDBC driver you use to connect to the database has implemented the `getMetadata()` JDBC API (For more information about this API, please refer to <http://java.sun.com/products/jdbc>). When using this mode ODI is limited to the metadata published by the JDBC driver. Therefore, the metadata defined in the database may not all be retrieved, depending on the implementation of the driver. To get an idea of which metadata is supported by your driver, either refer to the product documentation or use a generic JDBC database browser (such as Squirrel-SQL add-on shipped by default

with ODI) to view the database schema as it is returned by your driver.

- Customized Reverse-engineering using Reverse Knowledge Modules (RKM): When using an RKM, you can define specific code and rules to capture metadata from any external metadata provider into ODI models. RKMs usually provide advanced metadata support by querying data dictionaries at the database or application level. Examples of RKMs are provided by default with ODI for most common databases and applications (such as Teradata, Oracle, SAP/R3 etc.) Refer to the SunopsisODI Knowledge Modules section for more information about how to design your own RKM.
- Defining metadata manually or semi-manually: This method can be used on non-relational databases. Semi-manual import is provided for ASCII delimited files and COBOL Copy Books. When no metadata provider is available, the only solution remains to define the metadata manually for the model by creating datastores, columns and constraints.

Note: When performing a reverse-engineering with ODI, objects that don't exist anymore in the underlying metadata provider are not deleted from the ODI Work Repository as they may still be referenced within interfaces or packages. For example, a column that was removed from a database table will not be removed from the corresponding ODI datastore after a Reverse-engineering. You will have to delete it manually, after deleting all dependant objects.

Reverse-engineering Relational Databases

Most common databases on the market such as Oracle, IBM DB2, Sybase, MySQL etc. provide a JDBC driver that supports the Metadata API. It is therefore recommended that you use the standard JDBC reverse-engineering strategy whenever possible. To ensure the quality of the metadata returned by the driver, you can connect to the database using Squirrel-SQL and browse the schema's metadata.

If the JDBC standard reverse-engineering mode does not provide enough metadata, you can choose to develop your own Reverse Knowledge Module or update an existing one.

Non Relational Models

Flat Files and JMS Queues and Topics

To create metadata for a flat ASCII file or a JMS queue or Topic, it is recommended that you do the following:

- If the structure is delimited, you can manually insert a datastore under your model and click on the Reverse button located in the Columns tab of the datastore. This will parse some records from the file and assign column names and data types to the datastore. You will have to modify the column attributes if they don't match your needs.

- If the structure is fixed length and you don't have a Cobol Copybook describing your file, you will have to create a new datastore and enter you columns manually.
- If you have a document describing your structures, you can also decide to use the RKM File from Excel that retrieves metadata from a formatted Microsoft Excel workbook. You should refer to the documentation of this Knowledge Module for additional information.
- If your structures are already defined in another metadata repository, you can develop your own RKM that connects to the metadata provider and converts it into usable ODI metadata.

Note: If you have an existing datastore structure similar to your file or message structure, you can copy it from any model to your file model. To do that, simply drag and drop the datastore into the model and press the Ctrl key. ODI converts the data types from the source technology to the target one.

Fixed Files and Binary Files with COBOL Copy Books

If you have a COBOL Copybook describing your file structure, you can choose to reverse engineer it from the Columns tab of your datastore by clicking on the Reverse CopyBook button. The following restrictions apply when using this feature:

- The COBOL CopyBook description file must be accessible from Designer
- The CopyBook must start with segment 01
- If a record has several redefine statements, ODI will pick the first definition. To reverse any another definition, you will have to manually edit the CopyBook and keep only the appropriate definition.
- COBOL "Occurs" are converted into as many columns as defined for the occurrence. For example, if your segment MONTH OCCURS 12 has 2 fields MONTH-NAME and MONTH-ID, it will be converted by ODI into 12 x 2 = 24 columns: MONTH_NAME, MONTH_ID, MONTH_NAME_1, MONTH_ID_1, ..., MONTH_NAME_11, MONTH_ID_11.
- Line numbers are not supported. Every segment definition must start with a level number (01, 05, etc)
- FILLER fields should be assigned a unique column name before importing the CopyBook (FILLER_1, FILLER2, etc.).

XML

Oracle provides a JDBC driver for XML (ODI XML Driver (JDBC Edition)). Any XML file will be recognized by ODI as a relational database with tables and references (corresponding to XML nodes and hierarchies). Therefore, JDBC

reverse-engineering is supported by the ODI XML driver as a standard feature. You simply need to define the appropriate topology and click the reverse button of your XML model.

You will find more information about the ODI XML driver in the on-line documentation library.

LDAP Directories

Oracle provides a JDBC driver for LDAP (ODI LDAP Driver (JDBC Edition)) included in the Open Connector for LDAP. Any LDAP Directory will be recognized by ODI as a relational database with tables and references (corresponding to LDAP Classes and hierarchies). Therefore, JDBC reverse-engineering is supported by the ODI LDAP driver as a standard feature. You simply need to define the appropriate topology and click on the reverse button of your LDAP model.

You will find more information about ODI LDAP driver in the on-line documentation library of ODI.

Other Non Relation Models

ODI provides Reverse-engineering Knowledge Modules for packaged applications. These RKM rely either on APIs or on the underlying data dictionaries of these applications. Open connectors shipped by Oracle include RKMs for the following packaged applications:

- SAP R/3
- Oracle Applications
- J.D. Edwards One World
- Siebel
- People Soft
- Oracle Hyperion Essbase
- Oracle Hyperion Planning
- Oracle Hyperion Financial Management

Refer to the documentation of your Open Connectors for more information.

You may also find it useful to define new technologies in Topology Manager which simulate access to an API or to any other application. You will then have to enter the metadata depending on your definitions as datastores, columns and constraints.

Troubleshooting Reverse-engineering

JDBC Reverse-engineering Failure

It may happen that the JDBC driver you use to connect to your database does not support some of the JDBC API calls triggered by ODI. The standard reverse will then fail.

You can try to understand what is happening by connecting to your server using a SQL query tool and browse the metadata as it is returned by the driver.

To fix the problem, you can either develop a specific RKM based on the underlying data dictionary of the database server starting from an existing RKM as an example, or adapt the RKM SQL (JYTHON) that illustrates how the JDBC API is invoked. You will find more information in the SunopsisODI Knowledge Modules section of this document.

Missing Data Types

After a reverse-engineering, you may find that some of the columns were imported with a blank data type resulting in a “?” (question mark) in their icon in ODI Designer. This means that ODI was not able to match the data type of these columns to a known data type in Topology Manager.

When using the standard JDBC reverse-engineering feature, ODI attempts to match the data type name as returned by the API (TYPE_NAME) to a data type name as defined for the technology in Topology Manager.

If you face this problem, you can solve it easily by following these steps:

1. Run Squirrel-SQL and connect to your database server with the same URL, user and password as the ones used for the data server in Topology Manager
2. In the tree view, click on the table for which ODI returned empty data types
3. Select the Columns tab and locate the columns for which ODI returned empty data types. Write down the TYPE_NAME property of these columns for future reference.
4. Run Topology Manager and connect to your master repository.
5. In the physical architecture, insert each missing data type for your technology:
 - a. Use the same name as the TYPE_NAME you have noted in step 3 for setting properties Code, Name and Reversed Code
 - b. Enter the SQL syntax when using this data type for creating a table. You can use the %L and %P wildcards for length and precision (example: NUMERIC(%L,%P))
 - c. Enter the letter you want to appear as the icon for this data type in ODI Designer and indicate whether this data type is writable
 - d. In the Converted To tab, map this data type to other existing data types for other technologies. This mapping is used when ODI

converts data types from one technology to the other during loading phases.

- e. Apply your changes
 - f. Repeat these steps for every missing data type
6. If your technology is intended to be used as a target, locate the technologies potentially used as sources and edit their data types. In the Converted To tab, map the appropriate source data types to the ones you have created in step 5.
 7. Run ODI Designer and re-execute the reverse-engineering process. Your missing data types should now be recognized and mapped to the newly created data types.

This procedure can also be used when reverse-engineering in customized mode using an RKM. When using this method, ODI will attempt to match the data type you have used for populating column the SNP_REV_COL.DT_DRIVER with the data type defined in the Topology. You will find more information in the SunopsisODI Knowledge Modules section of this document.

Missing Constraints

Due to some known limitations of JDBC drivers, ODI may not retrieve some database constraints such as unique keys, foreign key, check constraints and default values. If you know that these constraints are returned by the JDBC API, you can customize the RKM SQL (JYTHON) to have them imported in ODI ; otherwise you can customize an existing RKM or write a new one to overcome this problem. Refer to SunopsisODI Knowledge Modules for further information.

Creating User-defined Data Quality Rules

Many database implementations don't enforce data integrity using constraints in the database engine. This often leads to data quality issues when populating the data warehouse. It is therefore good practice to define these constraints in your data models inside the work repository to match the data quality requirements of your Information System. You can define constraints either in the source models for data quality audit or in the target models for flow integrity.

Most common data quality rules can be implemented as constraints. To do this, you can add the following objects to your datastores:

- Primary or Alternate Keys to ensure the uniqueness of records for a set of columns. For example: ensure that the email address and company name are unique in the contacts table
- References (Simple) to check referential integrity of records against other records from a referenced table. For example: check that every record of my order lines table references an existing product ID in my products table
- References (Complex) to check complex referential integrity based on a

“semi-relationship” defined by a complex expression. For example: check that the 3 first characters of the address field concatenated with the 4 last characters of the postal code of the customer table reference an existing country code in my country table.

- Conditions to check a rule based on several columns of the same record. For example: check that the order update date is between the creation date and the purchase date.
- Mandatory columns to check that a column contains a non-null value.

To enforce the data quality rules of your source applications, you will have to run static data checks on the existing source data. This often reveals discrepancies in the data and can help you better understand the quality issues you may face when loading your Datawarehouse. You can do that easily from the models by assigning a Check Knowledge Module (CKM) to your model and right clicking on the datastore you want to control and selecting the “Control -> Check” menu item. This will launch a data quality session which you can follow in ODI Operator. This session will simulate every constraint you have defined on this datastore and create an error table in the work schema of the source application. This error table will contain all records that fail the constraints you have defined for this datastore. To view these records, you can simply right-click on your datastore and select the “Control -> Errors...” menu item. You can repeat this operation for every datastore or choose to check the entire model or a sub-model. If you want to automate this operation, you can simply drag and drop your model, sub-model or datastore inside a package and set your step type to a Model, Sub-Model or Datastore Check. When executing the package, ODI will trigger the data quality process to populate the error tables with any records that don’t meet your data quality requirements.

The same mechanism is used to check data in the flow before applying it to a target table within an interface. ODI will attempt to simulate the constraints you have defined for your target table on the incoming flow data. To setup this feature, you have to select the FLOW_CONTROL option of the Integration Knowledge Module (IKM) for your interface and select a Check Knowledge Module (CKM) in the Control tab. When executing your interface, any flow record that violates the constraints defined for your target table will be rejected, and stored in the errors table created in the staging area (work schema). Erroneous records are not loaded to the target. However, if you plan to recycle these records during the next execution of the interface, you can choose to do so by simply setting the RECYCLE_ERRORS option of your IKM.

In some cases, you may want to load erroneous records in the target table and simply be notified of the existence of these errors. In that case, you can choose to run a static check after loading your target by setting the STATIC_CONTROL option in the IKM of your interface to true. To access the error tables, you can

right-click on your target datastore and select the “Control -> Errors” menu or choose to access them from the reporting tool of your choice.

You will find more relevant information in section Synopsis ODI Knowledge Modules of this book. The “Technical Analysis - Data Quality in ODI” document also brings further analysis for setting up Data Quality best practices.

Adding User-Defined Metadata with Flex Fields

ODI provides a feature called Flex Fields designed to help you add your own metadata properties in ODI repository to enrich the existing default one. The benefits of this approach are:

- Providing more documented metadata to business users.
- Using your own specific metadata at runtime to perform particular tasks

Flex Fields are attributes that you add for certain types of ODI object in the Security module. These attributes will then appear as properties on every object of this type in Designer.

Suppose, for example, that you want to add a metadata field for every datastore of any Teradata model to enter the path for a MS Word document that describes this datastore. You would do that by following these steps:

1. Open the Security Manager module and switch to the “Objects” view
2. Edit the object named “Datastore”
3. In the FlexFields tab, add a new field with these properties
 - a. Name: Document Location
 - b. Code : DOCUMENT_LOC
 - c. Technology: Select Teradata (if left empty, the FlexField will be available for all datastores of any technology)
 - d. Type: String
 - e. Default: Set a default value for this field
4. Apply your changes and close Security Manager
5. Open the Designer Module
6. Edit one of your Teradata datastores and select the FlexFields tab. The “Document Location” flex field should be available as additional metadata for your datastore.
7. Uncheck the “Default” check box and enter the location of the MS Word document that describes this datastore.
8. Apply your changes

Figure 14 illustrates this example.

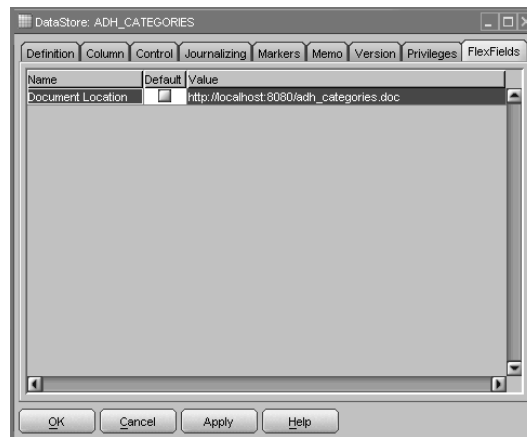


Figure 14: Setting a Flex Field Value for a Datastore

Flex Fields can also be used as additional metadata to perform specific tasks in Knowledge Modules. As soon as you define a Flex Field, it becomes available in the ODI substitution methods. You can therefore use the Flex Field value and customize your KMs accordingly.

For example, suppose you don't want to use ODI's default names for creating error tables during a data quality control, and you prefer to specify this name as metadata for every datastore. You would then do the following:

- Add the "Error Table Name" Flex Field to the datastore object in Security Manager
- Define the values for this Flex Field for every datastore that needs to be checked (for example, set it to "ERR_CUST1" for table "Customer")
- Modify the Check Knowledge Module and replace the name of the standard error table: (`<%=snpRef.getTable("L", "ERR_NAME", "W")%>`) with the value of your Flex Field for the target table (`<%=snpRef.getTargetTable("Error Table Name")%>`)
- From now on, the CKM will generate ERR_CUST1 instead of E\$_Customer

Section SunopsisODI Knowledge Modules further discusses the substitution methods available and best practices for customizing your KMs.

Documenting Models for Business Users and Developers

A good practice for better understanding the metadata stored in your applications is to document every datastore and column by adding descriptions, memos and headings. These descriptions are not only useful to developers but also to business users and analysts when browsing ODI's metadata using Metadata Navigator. The figure below shows how ODI displays column heading information and datastores' memos for a better comprehension of metadata.

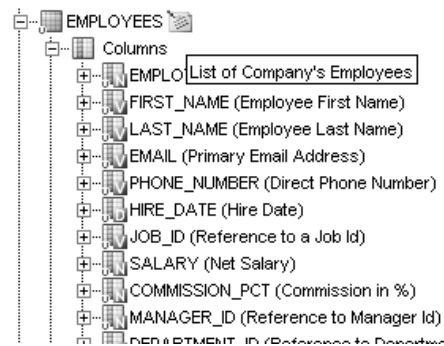


Figure 15: Example of Datastore and Columns Descriptions

Note: Some JDBC drivers implement the REMARKS API for tables and columns. If your tables and columns are already documented in the underlying databases with SQL comments, it should automatically be reverse-engineered.

Exporting Metadata

Although the ODI Repository, accessed by Metadata Navigator, provides a single interface for browsing the entire metadata of your Information System, you may find it useful to export a part of this information in order to share it with other metadata repositories. The easiest way to do this is to access the work repository tables in **read-only** mode using SQL queries to export the metadata in the format of your choice. When doing so, make sure you **do not** insert, update or delete manually any record from the ODI Repository tables, as this operation is not supported. For any modification of the contents of the repository, use the client modules Designer, Topology, Security and Operator.

Figure 16 shows ODI work repository tables used to manage models, sub-models, datastores, columns, key constraints, reference constraints and check conditions.

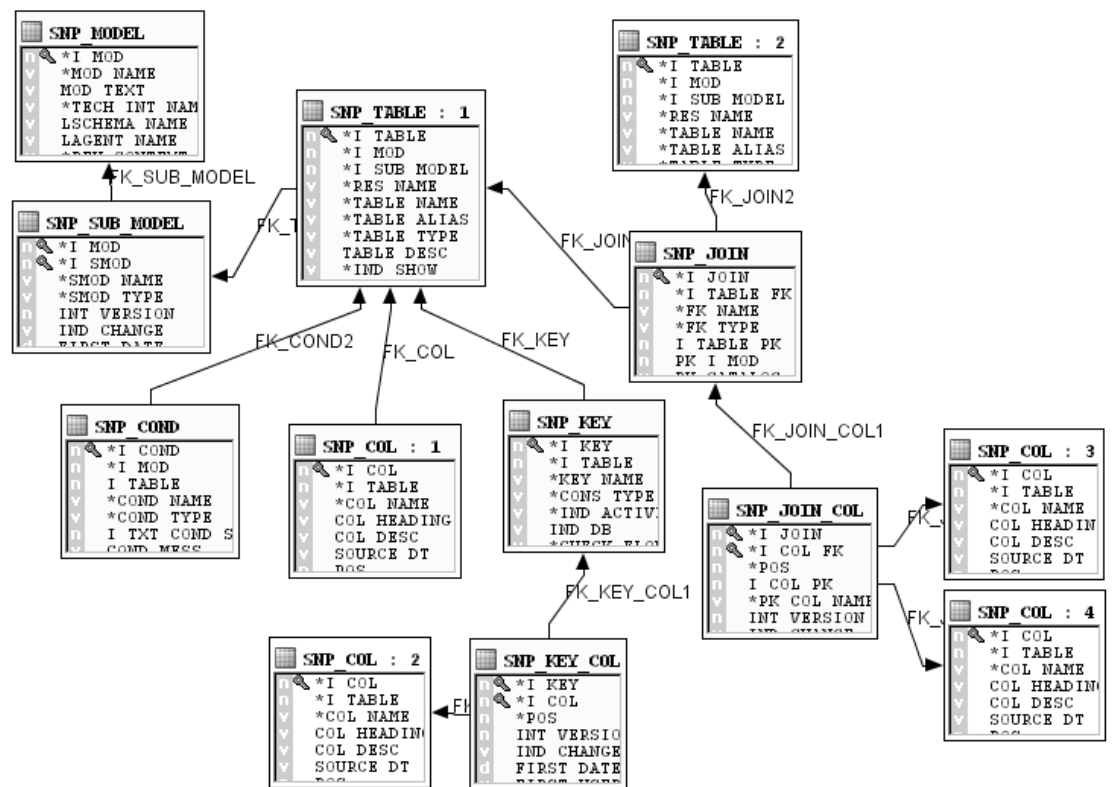


Figure 16: Database Model for ODI Work Repository Metadata

Table below gives a description of the tables used to store model metadata:

Table Name	Description
SNP_MODEL	Models and their attributes. Contains one row for each model.
SNP_SUB_MODEL	Sub-models and their attributes. Contains one row for each sub-model. When datastores appear under a model, they are in fact linked to a particular sub-model named Global and tagged 'D' in its SMOD_TYPE. Sub-models belong to a model and may reference other sub-models.
SNP_TABLE	Datastores and their attributes. Contains one row for every datastore. Datastores reference a Sub-model
SNP_COL	Columns and their attributes.

Table Name	Description
	Contains one row for every column. Columns reference datastores
SNP_KEY	Primary key (PK), Alternate keys (AK) and Indexes. Contains one row for every PK, AK or index. Keys reference datastores
SNP_KEY_COL	PK, AK and Indexes Columns. Contains one row for every column defined for the PK, AK or index. Key Columns reference Keys and Columns
SNP_JOIN	References or Foreign Keys. Contains one row for every reference. References reference datastores twice (reference datastore and referenced datastore)
SNP_JOIN_COL	Simple Reference Columns mapping. Contains one row for every pair of reference column/primary column definition. When the parent reference is complex, this table doesn't contain any row. JOIN_COL references JOIN and COL (twice)
SNP_COND	Conditions and Filters defined for a datastore. Contains one row for every condition or filter. Conditions reference datastores.

For a complete list of tables and columns with their description, ask for the ODI Repository Model documentation from oracledi-pm_us@oracle.com.

Note:

A standard metadata export feature for Teradata Metadata Services (MDS) is in available in beta test.

Impact Analysis, Data Lineage and Cross-References

ODI Repository maintains an up-to date list of cross-references between objects. This applies to models, datastores and columns. The list of these cross-references is described in detail below:

Object	Reference Type	Description
Model	Used In	What are the packages that reference this model in a Reverse-Engineering, Journalizing or control step?
Datastore	Used In -> As a Source	What are the interfaces that reference this datastore and use it to populate other datastores?
Datastore	Used In -> As a Target	What are the interfaces that are used to populate this datastore?
Datastore	Used in -> In a Package	What are the packages that reference this datastore in a Reverse Engineer, Journalizing or static control step?
Datastore	Used to Populate	What are the datastores that are populated by this datastore? (Where does the data from this datastore go to?) This reference is recursive as the populated target datastore can, itself, be used as a source for other datastores, etc. This powerful feature helps you follow the data lineage as it shows you the path of the data from this datastore.
Datastore	Populated By	What are the datastores that populate this datastore? (Where does the data come from?).
Column	Used to Populate	What are the columns that are populated by this column? (Where does the data of this column go to?) This reference is recursive as the populated target column can, itself, be used as a source for other columns, etc. This powerful feature helps you follow the data lineage as it shows you the path of the data from this particular column.
Column	Populated By	What are the columns that populate this column? (Where does the data of this column come from?).
Column	Used In -> As a Source	What mapping expressions, filters and joins reference this column in source diagrams of interfaces?
Column	Used In -> As a Target	What mappings reference this column as a target column within interfaces?

It is recommended that you use these cross-references to:

- Perform impact analysis before changing the structure of a datastore. For example, it will help you understand what impact updating a column data type may have on the existing interfaces.
- Understand data lineage (where it comes from and goes to) either for regulatory compliance or simply for maintenance purposes.

Through the Flow Map and Data Lineage links, Metadata Navigator provides the same information in a more friendly and graphical format. It can be used by business analysts to better understand the path of data within their applications.

Figure 17 below show an example of data lineage for the FACT_SALES table in Metadata Navigator. Figure 18 focuses on the F_SAL_VAT column lineage in ODI Designer.

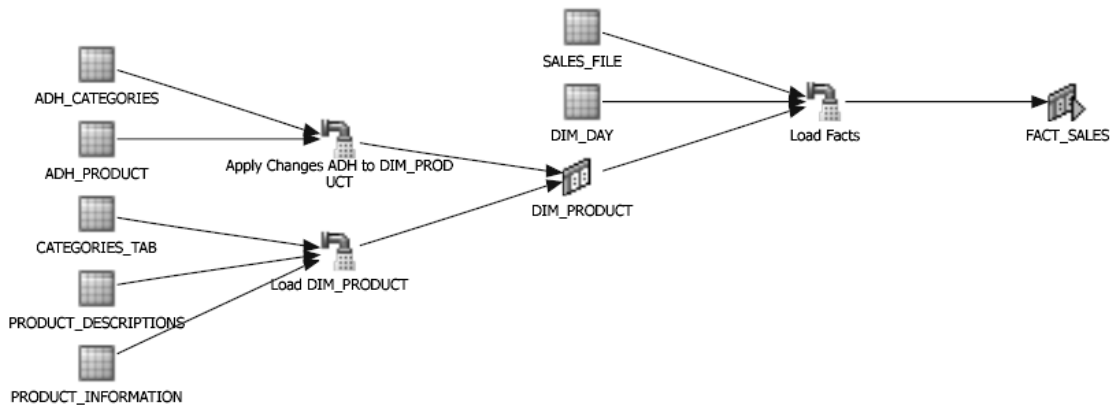


Figure 17: Data Lineage for the FACT_SALES table in Metadata Navigator

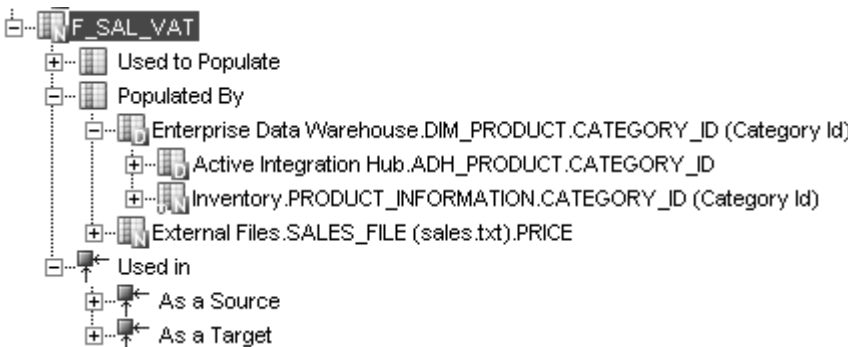


Figure 18: Column Lineage in ODI Designer

Object Naming Conventions

Models define the core metadata provided by your applications. They will be accessed by business users and by developers. It is therefore important to have every object match a naming convention so that it can be referenced appropriately. You can apply the following rules when assigning a name to your objects:

Object Type	Naming Rule	Examples
Model	Free. Business Area Name or Application Name	SAP

Object Type	Naming Rule	Examples
Folder		Financial Application
Model	Model names can be assigned freely and should match your standards. However, Model codes should be set with caution as they may be referenced by other objects within the repository. You could use the following rule for your model codes: <TECHNOLOGY>_<Model Acronym>	Model Name: Inventory XARP Model code: ORACLE_INV_XARP
Sub Model	Sub-model names can be assigned freely, but consider using the same constraints for the sub-model codes as for the model codes.	Sub Model Name: Parts & Products Sub Model Code: ORACLE_INV_PART_PRD
Datastore, Columns	Depends on the reverse-engineering strategy selected. When creating your own datastores and columns within ODI, it is recommended that you enter object names and codes in upper case and avoid special characters such as spaces, commas, semi-colons etc.	CUSTOMER_FILE
Primary key	PK_< Name of the datastore > May depend on the way primary keys are defined in the database or application.	PK_CUSTOMER
Alternate Key	AK_<Name of the datastore>_<extra> May depend on the way alternate or unique keys are defined in the database or application.	AK_CUSTOMER_NAME
Reference	FK_<FROM Datastore>_<TO Datastore>_number May depend on the way foreign keys are defined in the database or application.	FK_CUSTOMER_SALES_REP_001
Check Constraint	CK_<Datastore>_<extra> May depend on the way check constraints are defined in the database or application	CK_CUSTOMER_VALID_AGE

IMPLEMENTING ODI PROJECTS AND MAPPINGS

Introduction to Projects

Projects store objects that you develop to populate your Data Warehouse. Projects can be organized in folders and contain the following types of objects:

- **Knowledge Modules:** The cornerstone for building interfaces to populate target datastores from heterogeneous sources. The section *SunopsisODI Knowledge Modules* further discusses best practices for importing existing Knowledge Modules or developing your own.
- **Variables:** Variables defined in a project can be used anywhere within the scope of that project. They can be refreshed or accessed at runtime. Packages can evaluate or change their current value to implement control structures. Variables can be set as persistent to be retained across multiple runtime sessions.
- **Sequences:** Sequences are objects that can be used by the ODI Agent as a counter. They can be used for example to assign an automatically calculated serial number to a field of a datastore. However, as the values of sequences are managed by the agent, using them forces a loading job to be processed row-by-row rather than in bulk, which may lead to performance issues. Thus, use caution when considering using sequences. When databases have facilities for assigning an automatic counter for a column (such as sequences or identity columns), ODI recommends using these mechanisms, as they will always be more efficient than ODI's Sequences.
- **User Functions:** A comprehensive way of sharing transformations across different interfaces and translating these transformations to make them available on different technologies. Using user functions is recommended when the same transformation pattern needs to be assigned to different datastores within different interfaces.
- **Folders:** Folders are used to organize the developed objects within a project. Use caution when referencing objects across folders as it may lead to import/export issues. For example, try to avoid referencing interfaces of another folder within a package of your current folder as much as possible.
- **Interfaces:** The main objects where you design your transformations as business rules. Interfaces map several heterogeneous source datastores to a target datastore.
- **Procedures:** Composed of a set of steps that allow you to execute your own specific code. This code can be either database-specific SQL (such as PL/SQL or Transact SQL), operating system commands, Java, Jython, or ODI built-in commands (ODI API). Steps within a procedure can mix calls to any of these programmatic languages. A transaction control mechanism is also available for database specific commands.

- **Packages:** Packages are used to implement a technical workflow composed of several steps linked together. They define the sequence of execution of jobs that you plan to release in production.

Note:

Projects in the ODI terminology don't necessarily match your enterprise's definition of a project. You can have several projects defined in ODI, all of them belonging to the "Data warehouse" project. For example, you may have one project by business area rather than a large project containing everything. It is even recommended that you split your development into several small projects containing less than 300 objects. This will improve the efficiency of your import/export and version control operations. See "Using SunopsisODI Version Management" for details about versioning.

Importing Knowledge Modules

As Knowledge Modules are the cornerstone of your integration processes that transform your business rules into comprehensive code executed by your servers and databases, they must be imported prior to starting development. Before importing Knowledge Modules into your project, you should be able to answer the following questions:

- What technologies will I be using as sources and targets?
- What kind of loading strategies do I want to adopt between my sources and targets?
- What type of integration strategy do I want to adopt to populate my targets? (Do I need Incremental Updates, Insert/Replace, Slowly Changing Dimensions, etc?)
- Do I need to set up a data quality strategy on my sources and/or for my data flows?
- Do I need to use a customized reverse-engineering strategy for my models or can I rely on the standard JDBC reverse engineering method provided?
- Am I using another framework with its own set of Knowledge Modules rather than the standard framework provided by ODI?

The answer to these questions will help you narrow down the number of Knowledge Modules that you need to import into your project. Section "SunopsisODI Knowledge Modules" discusses all these issues and gives advanced guidelines for building your own framework or using the standard one.

Implementing Business Rules in Interfaces

Definition of an Interface

An Interface defines a heterogeneous query on several source datastores to populate a target datastore. It contains the business rules that will be converted into appropriate code depending on the Knowledge Modules selected. When defining an interface, you should focus on **what** you want to do (in terms of rules) rather

than on **how** you want to do it (in terms of technical process). Consider implementing an interface as if you were writing a SELECT statement using several datastores in the FROM clause. The main differences are:

- The query to retrieve source data is designed graphically
- The datastores of the query can be on different databases, servers or applications.

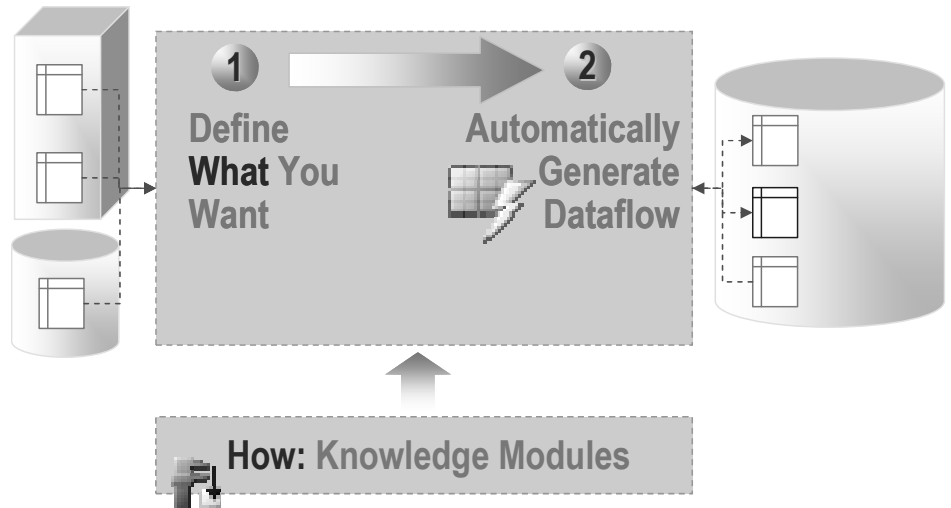


Figure 19: ODI's business-rule driven architecture

As described in Figure 19, once you define **what** you want to achieve in terms of rules, ODI generates the appropriate data flow to perform the transformations depending on **how** you want to do it using Knowledge Modules.

Designing the Business Rules

Business rules within ODI are defined either as expressions or as constraints. It is recommended that you document every business rule that contains complex logic so that business analysts can quickly understand what is done within your interface. Expressions for the business rules are defined using the supported language of the source, staging area or target technology. In most of the cases, expressions contain specific SQL operations and transformation functions that will be executed by your technologies. Therefore, you can mix languages for your expressions in the same interface to leverage the power and flexibility of your servers. Business rules within an interface can be mappings, joins, filters or constraints as defined below.

Mappings

A mapping defines an expression that mixes several source columns to populate a target column. To design your transformation into a mapping, you can use complex logic. This includes ODI user functions, ODI variables, database built-in scalar functions, database aggregation functions, database windowing functions, user-defined database functions (when supported by the technology) etc.

It is important to keep the following rules in mind when designing your mapping expressions:

- A mapping can be **executed on the source** when all the source columns used in the expression come from the same source data server AND the source data server technology supports select expressions. In this case, you will use the SQL capabilities of your source data server to carry out the transformation.
- A mapping can always be **executed on the staging area**. However, you should consider executing your mappings on the source whenever possible to distribute some of the workload onto the sources for better efficiency.
- When using **aggregate functions** such as SUM(), AVG(), COUNT(), MIN(), MAX(), etc. in your mapping, it is recommended that you execute the expression on the staging area. In some particular cases, mixing heterogeneous sources with aggregations executed on the source may produce incorrect results. This is not the case when you execute those aggregate functions on the staging area.
- A mapping can be **executed on the target** when it does not contain any reference to a source column. These kinds of mappings are executed post staging area, right before the insert or update statement on the target datastore. They are useful for setting a target column to a constant, variable or function without any parameter (such as current_date(), user() etc.). Executing these types of mappings on the target improves efficiency as the constant, variable or result of the function is not processed or stored in the staging area.

In the following table:

- A and B represent 2 source columns from 2 distinct datastores in the source panel of your interface.
- $f(A, B)$ represents a complex expression mixing columns A and B to produce a result (for example: `case when A > 10 and B < 20 then A+B else A-B end`). $f()$ doesn't contain any aggregate functions.
- $g()$ represents a database or user-defined function without any parameters.
- `#MY_VARIABLE` is an ODI variable.
- `'MY_CONSTANT'` is a string constant.

The table gives you a quick overview on the recommendations for executing your mapping:

Mapping expression	Executable on source?	Executable on Staging Area?	Executable on the Target?
--------------------	-----------------------	-----------------------------	---------------------------

Mapping expression	Executable on source?	Executable on Staging Area?	Executable on the Target?
$f(A, B)$	Yes, if A and B come from the same source server and the source server supports $f()$. In that case it is recommended that you execute the mapping on the source	Yes, if $f()$ is supported by the technology of the staging area.	No
$SUM(A+B)$	Possible, when A and B come from the same source server. NOT recommended	Always. Recommended	No
$g()$	Yes when $g()$ Is supported by the source. NOT recommended	Yes when $g()$ Is supported by the staging area. NOT recommended	Yes when $g()$ Is supported by the target. Recommended
#MY_VARIABLE	Possible but NOT recommended	Possible but NOT recommended	Yes. Recommended
'MY_CONSTANT'	Possible but NOT recommended	Possible but NOT recommended	Yes. Recommended

Notes:

- Describe your mappings in natural language, as much as possible. Business users will drill down into the details of your rules when using the data lineage to understand where the data came from and how it was transformed.
- Use user functions for reusable mappings

Joins

A join is an expression that links 2 source datastores together. It references columns of the 2 datastores. As an example, consider 2 datastores (A and B). The following expressions are valid joins:

- $A.C1 = B.C2$ and $(A.C3 = B.C4$ or $A.C3$ is null and $B.C4 > 20)$
- $A.C1$ between $B.C2$ and $B.C3$
- $substring(A.C1, 1, 3) || cast(A.C2 + A.C3 as varchar) = case$
when $B.C4$ is null then $coalesce(B.C5)$ else $B.C6$ end

An ODI join respects standard relational algebra. It can be defined as inner join, left or right outer join, full outer join or cross join (Cartesian product). Although it is possible to execute a join in the staging area, it is recommended that you execute it whenever possible on the sources to minimize the network traffic.

The following rules apply when designing your join expressions:

- A join can be **executed on the source** when all the source columns used in

the expression come from the same source data server AND the source data server technology supports SQL joins. Left, right, and outer joins as well as ordered ISO joins options depend on the capabilities of your source server. Executing joins on the source is recommended in order to reduce network traffic between the source and the staging (or target) server.

- A join can always be **executed on the staging area**. However, you should consider executing your join on the source whenever possible to distribute some of the workload onto the sources for better efficiency. Left, right, and outer joins as well as ordered ISO joins options depend on your staging server's capabilities.
- When the technology on which the join is executed supports ISO syntax, ODI will generate the appropriate join expression in the FROM clause of the SELECT statement in the form (A left outer join B on (A.C1 = B.C2)).
- When the technology on which the join is executed supports ordered joins, ODI will use the order defined for the join to build the FROM clause of the SELECT statement in the form: ((A left outer join B on (A.C1 = B.C2)) left outer join D on (D.C3 = B.C4))

Notes:

- Describe your joins in natural language, as much as possible. Business users will drill down into the details of your rules when using data lineage to understand where the data came from and how it was transformed.
- When a join is executed on source, you can right-click on it in the source panel to see either the number or rows returned by the join or the data of the 2 sets used in the join. This is a very helpful feature for debugging your rules.
- Cartesian products between 2 datastores must be declared as a join with an empty expression and the "Cross Join" check box checked.

Filters

A filter is an expression that filters data coming from a source datastore. Therefore, it references one or several columns of the datastore. The natural recommendation is obviously to try to execute filters on the sources to reduce the number of rows processed.

The following rules apply when designing your filters:

- A filter can be **executed on the source** when all the source columns used in the filter come from the same source data server AND the source data server technology supports filters in the "where" clause. Executing filters on the source is recommended.
- A filter can always be **executed on the staging area**. However, you should consider executing your filter on the source whenever possible.

Notes:

- Filters contribute to your business rules and may include important business logic. Therefore, try to describe them in natural language to allow business users to understand your interfaces.
- When a filter is executed on source, you can right-click on it in the source panel to see either the number or rows returned by the filter or the filtered data. This is a very helpful feature for debugging your rules.

Constraints or Data Quality Rules

Constraints are business rules that participate in the processes to ensure data quality of your target datastore. They are set in models for the target datastore, and you can choose from the “Control” tab of your interface which ones to activate.

The section *Creating User-defined Data Quality Rules* explains how the constraints that you define on your target datastore can be used to enforce the integrity of your flow data.

Oracle recommends using the data quality control features as much as possible to provide consistent and accurate data in the Enterprise Data Warehouse. See section Data Quality Strategies (CKM) for details.

Insert/Update Checkboxes on Mappings

When designing your mappings, and if using an “Incremental Update” strategy to populate a target datastore, you can decide which of your mappings will be included in the INSERT statement, and which ones will be included in the UPDATE statement. This can be done simply by checking the “Insert” and/or “Update” checkbox on your mappings.

To better understand this feature, consider a target datastore containing a CREATION_DATE and UPDATE_DATE columns. You want to set the CREATION_DATE to the system date (current_date()) when the record is created. This value should never be updated so that it will always reflect the actual creation date of the record. On the other hand, you want to set the UPDATE_DATE column to the last date when the record was updated. This column should not be set for new records.

In this kind of situation, you would set the CREATION_DATE mapping to “current_date()” and check only the “Insert” marker. And of course you would do the same for the UPDATE_DATE mapping by checking only the “Update” marker as described in the table below.

Target Column	Mapping	Insert?	Update
CREATION_DATE	Current_date()	YES	NO

Target Column	Mapping	Insert?	Update
UPDATE_DATE	Current_date()	NO	YES

Using the Insert and Update markers for your mappings brings extra flexibility when designing your business rules. These markers interact closely with the type of Knowledge Module you choose. See the section *SunopsisODI Knowledge Modules* for details.

UD1..UD5 Markers on the Mappings

The “Insert” and “Update” markers on your mappings bring a certain amount of flexibility when designing your business rules. However, you may want even more flexibility by defining your own behavior depending on markers that you can then set on your mappings. To do this, you can use the user markers UD1 to UD5 on your mappings. You should then use your own appropriate Knowledge Module that uses the value of these markers to turn this information into code.

To give an example of how markers can bring flexibility and productivity while designing your rules, suppose that you want to generate a message for every record inserted in a table separate from the target datastore. This message would simply be a varchar column containing the concatenation of some (but not all) of the target columns. Of course, you would like to generalize this behavior to a set of target tables of your Data Warehouse. You could therefore adapt an Integration Knowledge Module to your needs by adding the appropriate logic to build the message string to generate the messages by using all target columns marked as UD1. Then, when designing your interfaces to populate your target datastores, you would simply need to indicate which columns on your target you want to use to build the message string by setting their UD1 marker.

For more information about how to get the best results using the UD1 to UD5 markers, please refer to the section *SunopsisODI Knowledge Modules*.

Note:

Some of the built-in ODI Knowledge Modules use the UDx markers. In this case the meaning of each marker within the scope of this KM should be documented in the KM itself.

Update Key and Incremental Update Strategy

If you plan to use an Incremental Update strategy to populate your target datastore, you will need to consider setting an update key for your interface. This will indicate the subset of columns of your target datastore that you want to use to compare your flow data with your target existing data to determine what should be inserted and what should be updated. By default, ODI sets this update key to the primary key of your target datastore. In some cases, using the primary key as the update key is not appropriate, for example:

- When the values of the primary key are automatically calculated (by an identity column, using a sequence or using specific code etc.). In this case, as your source flow doesn't contain this information, it is not possible to compare it to the existing target values. You should therefore choose another set of unique columns to compare the flow data with the target data (e.g. another update key than the primary key)
- When it is desirable for the comparison to be made on a set of unique columns other than the primary key, either for efficiency reasons or for specific business rules (such as comparing customers using the name+city+country columns rather than on their customer_id primary key)

Notes:

- The update key must always uniquely identify each row.
- To change the default update key in your interface, you can:
 - o Add an alternate key with the appropriate unique columns to your target datastore in your models.
 - o Open your interface.
 - o Click on the target datastore in the diagram tab.
 - o Select the new key from the update key list box in the property panel. The target columns belonging to this key should have a blue key icon.
 - o OR
 - o Open your interface.
 - o Click on the target datastore in the diagram tab.
 - o Select <Undefined> in the update key list box in the property panel.
 - o Click on a target column that belongs to the new update key
 - o Check the "Key" check box in the property panel of the mapping. The target column should have a blue key icon.
 - o Repeat these steps for other columns in the update key.

Flow Diagram, Execution Plan and Staging Area

As we have seen in the previous chapter, when you design your business rules, you do it in a logical way by drag and dropping objects onto the Diagram tab of your interface to define what you want to do rather than how you want to do it. For optimization purposes, it is now important to understand the "execution plan" (or flow diagram) that ODI will suggest to carry on the business rules from your sources to the target. Some very simple options can affect this execution plan in several ways. Using these options intelligently can help you optimize your interfaces for efficient execution.

To understand the following examples, let's consider that we have an interface with 4 source datastores (A, B, C and D) that are used to map a target (TARGET) datastore. A, B, C and D are joined together. A and D are filtered. Figure 20 illustrates this example.

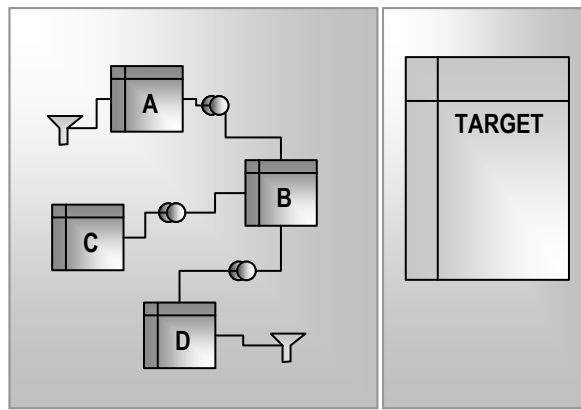


Figure 20: Interface Example

The next sections explain the impact of the location of the datastores of your interface as well as the choice of staging area.

Sources, Target and Staging Area on the Same Server

In our example, suppose that:

- The source datastores A, B, C and D are in your Teradata server
- The target datastore is in the Teradata server as well
- The staging area of your interface is the same as the target (the “Staging Area Different from Target” check box is not checked in your “Definition” tab for this interface).

In other words, sources, target and staging area are in the same Teradata server. In this simple architecture, all the mappings, joins, filters and controls will be executed by the Teradata server as described in Figure 21

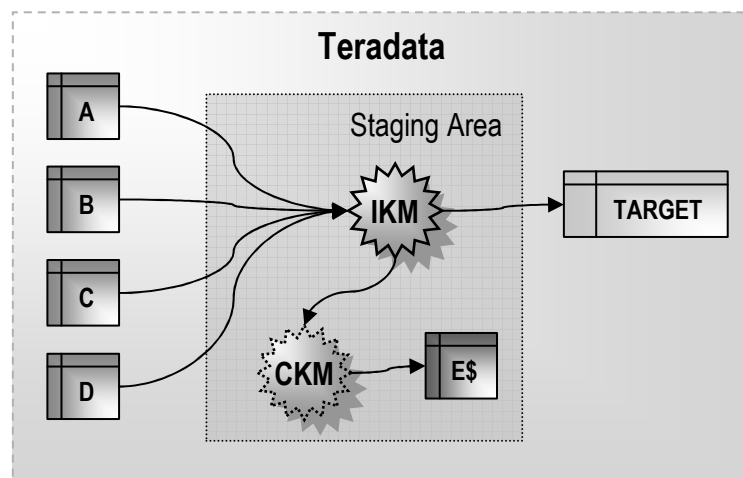


Figure 21: Sources, Target and Staging Area on the Same Server

The Integration Knowledge Module (IKM) will generate the appropriate SQL SELECT statement to join, filter and transform the data from A, B, C and D and build the resulting flow data. If the FLOW_CONTROL option is set to yes, it will then trigger the Check Knowledge Module (CKM) to validate the data quality rules of the target (constraints) and isolate any erroneous records into the error table (E\$). After that, the IKM will populate the target table with valid data according to the strategy it was designed for (Incremental Update, Insert Replace, Append, Slowly Changing Dimension, User defined strategy etc.)

This behavior is certainly the most efficient for performing such a job. Source records are NOT extracted to a different server. Data remains in the server and is transformed using bulk operations with the power of database engine.

Target and Staging Area on the Same Server

In our example, suppose that:

- The source datastores A and B are 2 IBM DB2 tables
- The join between A and B is set to be executed on the source
- The filter on A is set to be executed on the source
- Datastore C is a flat file
- Source datastore D and the target datastore are in the same Teradata server
- The staging area of your interface is not different from the target (the “Staging Area Different from Target” check box is not checked in your “Definition” tab for this interface).

In other words, the interface combines heterogeneous sources as described in Figure 22. By default, the flow diagram of the figure will be the recommended execution plan that ODI will suggest for your interface.

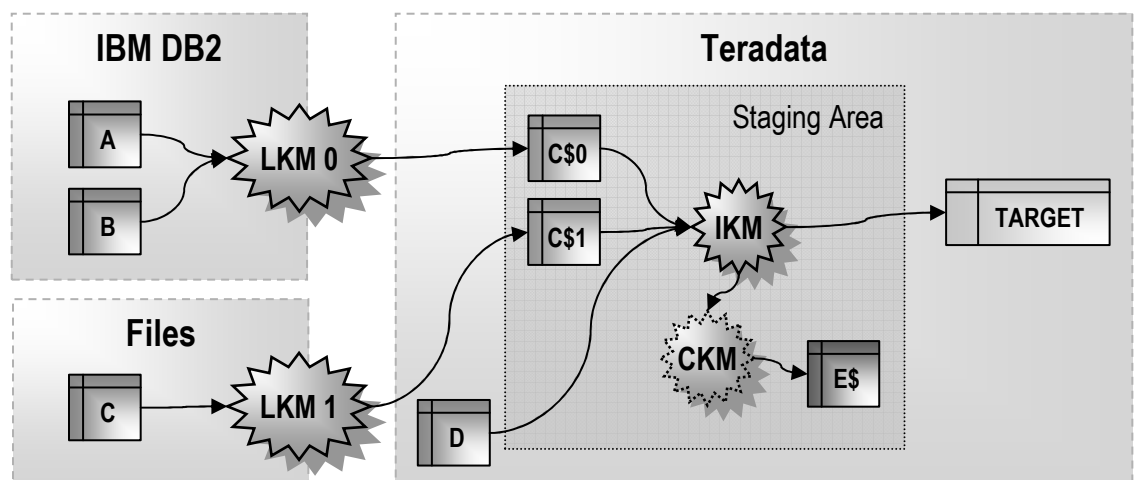


Figure 22: Target and Staging Area on the Same Server

As A and B are on a remote IBM DB2 server that can perform joins, filters and some of the mappings, the first Loading Knowledge Module (LKM 0) will generate a SQL SELECT statement on IBM DB2. This statement will build a result set by executing a subset of your transformations on the source, using the power of IBM DB2. Then this same LKM will transfer this result set to a temporary table (C\$0) in the Teradata staging area. How this is performed depends on the type of LKM you chose for this source (JDBC to JDBC, Unload and FastLoad, etc.).

The second Loading Knowledge Module (LKM 1) will basically do the same to load the C flat file to the temporary C\$1 table in the staging area. Again, the efficiency of this loading phase will depend on the LKM you choose for this source (FastLoad, MultiLoad, Read then write using JDBC etc.)

As the source datastore D is in the same server as the staging area, no loading is required. Therefore, the Integration Knowledge Module (IKM) will finish the work in the Teradata server by generating the appropriate SQL to join and transform C\$0, C\$1 and D to produce the flow data. If the FLOW_CONTROL option is set to yes, it will then trigger the Check Knowledge Module (CKM) to validate the target's data quality rules (constraints) and isolate any erroneous records into the error table (E\$). After that, the IKM will populate the target table with the valid data according to the strategy it was designed for (Incremental Update, Insert Replace, Append, Slowly Changing Dimension, User defined strategy etc.)

Although a part of the workload was dedicated to the source IBM DB2 server, most of the heavy work is still done by the Teradata server. This demonstrates the powerful E-LT architecture of ODI.

In this same example, if you simply change the execution location of the join between A and B from "Source" to "Staging Area", you will notice that ODI will suggest another execution plan as described in Figure 23.

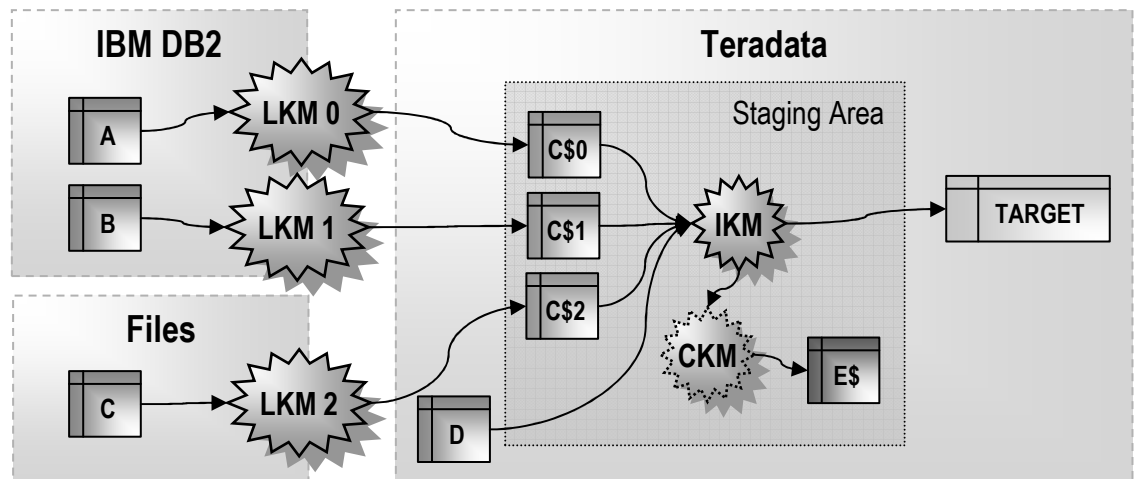


Figure 23: Target and Staging Area on the Same Server (A & B Joined on the Staging Area)

Rather than extracting the data from IBM DB2 using a single query within a single LKM, it suggests doing this for every table with 2 LKMs (LKM 0 and LKM 1). This change can, of course, dramatically reduce the efficiency of the interface, depending on the number of rows contained in A and B. It should be considered only if you are sure that the execution time for loading A from DB2 to Teradata + loading B from DB2 to Teradata + joining A and B in Teradata is shorter than the execution time for joining A and B in IBM DB2 and loading the result in Teradata.

Staging Area in ODI Memory Engine

In our example, suppose now that:

- The staging area of your interface is different from the target and set to `SUNOPSIS_MEMORY_ENGINE` (the “Staging Area Different from Target” check box is checked in your “Definition” tab and you have selected the appropriate logical schema from the list box).
- A and B are still in DB2, C is still a flat file and D and the target are in the same Teradata server

The execution plan in the Flow Diagram tab of your interface will be as described in Figure 24.

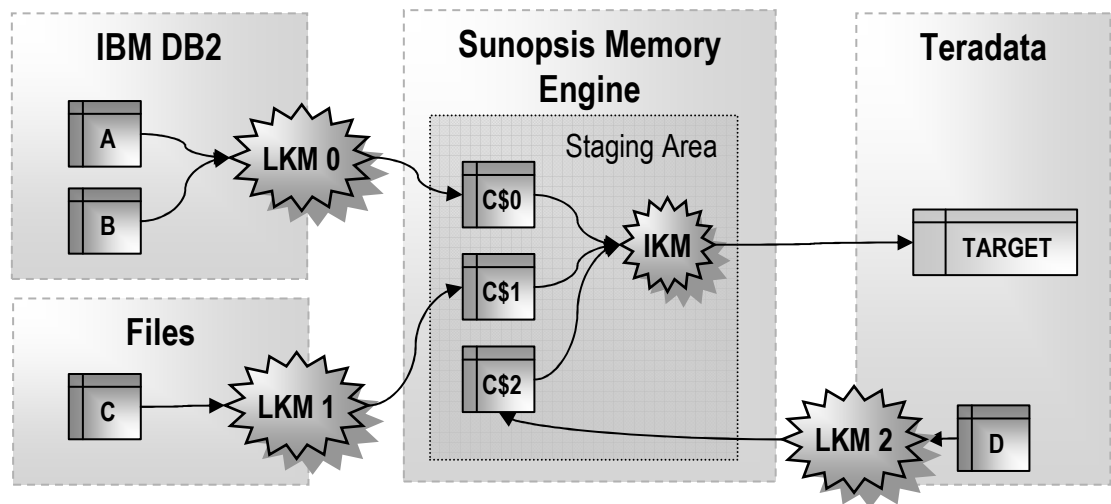


Figure 24: Staging Area in ODI Memory Engine

The Loading Knowledge Modules (LKM 0, LKM1 and LKM2) will load the source data from all the sources (A+B, C and D) to temporary tables in memory in the ODI Memory Engine.

The IKM will finish the job in memory by generating the appropriate SQL SELECT statement on the C\$ tables to build the flow data, before writing it to the target Teradata table.

This example is certainly **NOT RECOMMENDED** for handling larger volumes of data, but it can be useful in certain rare cases where:

- Volumes from each of the sources are **VERY** small (less than 10 000 records)
- Processing needs to happen very quickly in memory for real-time or near real time integration.
- The IKM uses a Loading utility designed for continuous loading (such as Tump)

Note:

When using a Staging Area different from the target it is not possible to use flow control.

Staging Area on one of the Sources

In our example, suppose now that:

- A and B are still in DB2, C is still a flat file and D and the target are on the same Teradata server
- The staging area of your interface is different from the target and set to the IBM DB2 database that contains A and B (the “Staging Area Different from

Target” check box is checked in your “Definition” tab and you have selected the appropriate logical schema from the list box).

The execution plan suggested by ODI is described in Figure 25.

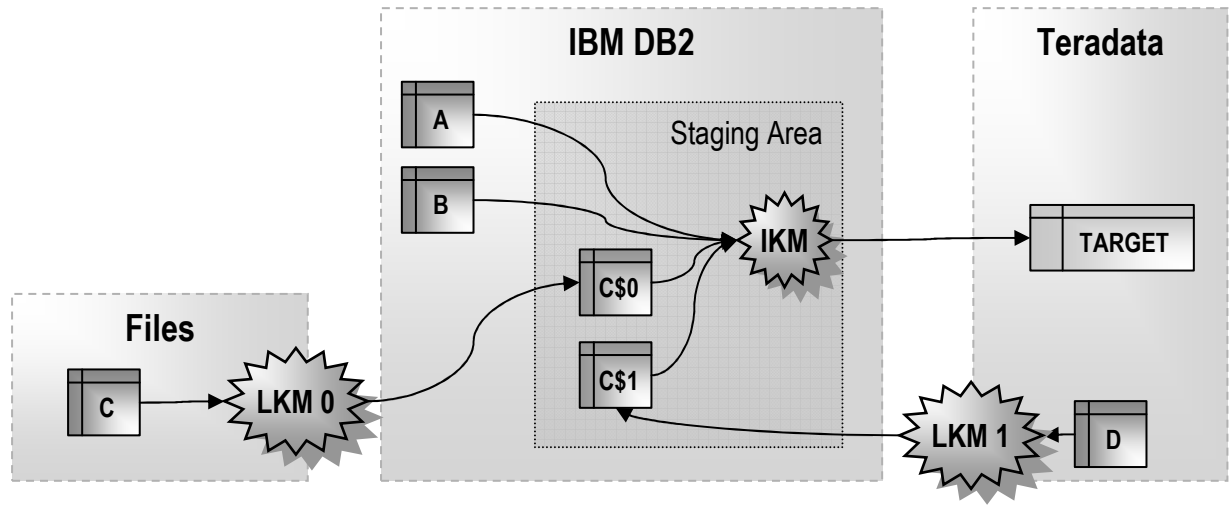


Figure 25: Staging Area on one of the Sources

LKM 0 and LKM 1 load source data into the tables C\$0 and C\$1 respectively on the IBM DB2 staging area.

The IKM uses DB2 to perform the final joins between A, B, C\$0 and C\$1 for building the flow data. It also exports and loads the resulting flow into the target Teradata table.

Although this architecture doesn't make any use of the processing power of the Teradata server, it could be useful if the following conditions are met:

- The IBM DB2 database administrators allow you to set up a staging area on DB2
- The volume of the join between A and B is large (> 5,000,000 records)
- C and D are small (less than 5,000 records)
- The result produced by joining A, B, C\$0 and C\$1 is small (about 10,000 records)

If the staging area was set to Teradata (see *Target and Staging Area on the Same Server*), the volume of data to load in the staging area would have been: 5,000,000 + 5,000 records and Teradata would have joined 5,000,000 + 5,000 + 5,000 records.

In the case of our example, rather than transferring these 5,005,000 records from the sources to Teradata, only 20,000 records are transferred across technologies (5,000 + 5,000 to IBM DB2, then 10,000 from IBM DB2 to Teradata). Provided that IBM DB2 is able to join the 5,000,000 + 5,000 + 5,000 records in an

acceptable time, it may be worth considering this configuration to optimize your job.

Target Datastore is a Flat File

In our example, suppose now that:

- The target is a flat file
- A and B are still in DB2, C is still a flat file and D is on the Teradata server
- The staging area of your interface is different from the target and set to one of the Teradata databases on the same server that contains D (the “Staging Area Different from Target” check box is checked in your “Definition” tab and you have selected the appropriate logical schema from the list box).

The generated flow will be as described in Figure 26.

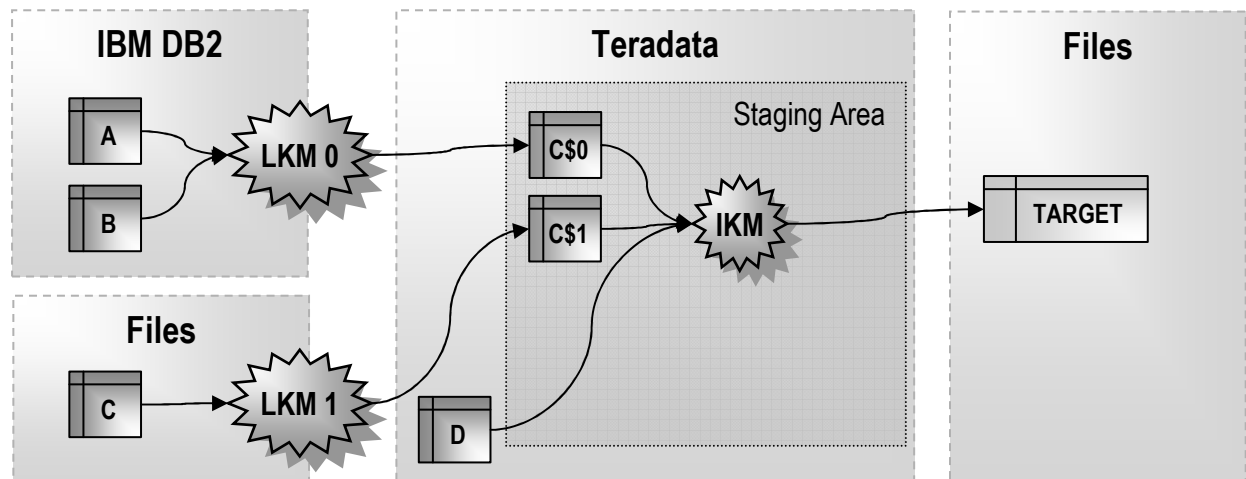


Figure 26 : Target Datastore is a Flat File

Source data will be loaded by the LKMs to the Teradata staging area as described in the previous chapters. After that, the IKM will perform the appropriate SQL transformations on the C\$0, C\$1 and D tables to generate the flow. It will then export this flow to the target flat file according to the strategy selected (FastExport, writing using JDBC or API etc.)

This architecture is certainly the one recommended for loading a flat file from heterogeneous sources.

Note:

When populating a target flat file in your interface (or any datastore from a technology that doesn't support joins and filters), you **MUST** select a staging area different from the target and set it to a logical schema for a technology that has advanced SQL

Impact of Changing the Optimization Context

To calculate the best execution plan, ODI requires physical information about the actual location of the logical schemas used. To resolve this physical path, it uses the optimization context specified in the Definition tab of your interface. By using this context, ODI can determine which physical servers will be involved in the transformation process. This information is then used to split the process into appropriate loading and integration phases. By default, ODI assumes that the number of physical servers involved in your Optimization Context will be the same as the number of physical servers in your production context. If this is not the case, you should refer to section *Contexts* in chapter *Defining the Topology in Sunopsis* for best practices for defining the topology. This problem is often referred to as the “fragmentation” problem and can be summarized with this rule:

The development environment’s topology should always reflect the most fragmented production environment.

Notes:

Use caution when changing the optimization context if the number of your production servers is not the same in all your contexts. Sometimes, choosing the production context as the optimization context can solve this issue

Changing the optimization context on an appropriate topology – as defined in chapter *Defining the Topology in Sunopsis* – should not have any impact on the execution plan generated by ODI.

Enforcing a Context for Sources or Target Datastores

In an interface, you can choose a different context for each datastore involved in the diagram tab. You can do that for your source datastores as well as for your target. In most cases these contexts should remain set to “undefined” or “execution context”. The physical path to your servers will normally be bound to the selected context at runtime, enabling the execution of the same interface across different environments.

Now, suppose you want to create an interface to move data from your PARTS table in your production server A to the same PARTS table in your development server B. A and B are defined in the physical architecture of the topology but both are accessed using the same logical schema INVENTORY. In Designer, the PARTS datastore appears only once in the models as it belongs to the INVENTORY logical schema. Therefore in this rather unusual case, you will need to set up your interface as follows:

1. Drag and drop the PARTS datastore from your model to the target of your interface
2. Set the context of your target datastore to your development context

- rather than <Undefined>. ODI will use the datastore on development server B regardless of the context specified at runtime.
3. Drag and drop the same PARTS datastore from your model to the source panel of your diagram
 4. Set the context of your source datastore to your production context rather than <Execution Context>. ODI will use the datastore on production server A regardless of the execution context.
 5. Executing your interface in the development or production context will have exactly the same result, e.g. moving data from server A to B.

Notes:

Consider enforcing a context on your sources or target datastores in your interfaces only if your interface is designed to move data across certain contexts regardless of the chosen context at runtime, for example, when moving data between your development and your production environment.

Avoid enforcing a context on your sources or target datastores when designing your normal interfaces for loading your Datawarehouse.

Using Procedures

Introduction to Procedures

A Procedure in ODI is a powerful object that allows you to develop additional code that will be executed in your packages. Procedures require you to develop all your code manually, as opposed to interfaces. Procedures should be considered only when what you need to do can't be achieved in an interface. In this case, rather than writing an external program or script, you would include the code in ODI and execute it from your packages.

A procedure is composed of steps, possibly mixing different languages. Every step may contain two commands that can be executed on a source and on a target. The steps are executed sequentially from the start to the finish. Some steps may be skipped if they are controlled by an option.

The code within a procedure can be made generic by using string options and the ODI Substitution API.

Writing Code in Procedures

Commands within a procedure can be written in several languages. These include:

- SQL: or any language supported by the targeted RDBMS such as PL/SQL, Transact SQL etc. Usually these commands can contain Data Manipulation Language (DML) or Data Description Language (DDL) statements. Using SELECT statements or stored procedures that return a result set is subject to some restrictions as described further in section *Using Select on Source / Action on*. To write a SQL command, you need to select:
- A valid RDBMS technology that supports your SQL statement, such as Teradata or Oracle etc.

- A logical schema that indicates where it should be executed. At runtime, this logical schema will be converted to the physical data server location selected to execute this statement.
- Additional information for transaction handling as described further in section *Handling RDBMS Transactions*.
- Operating System Command: Useful when you want to run an external program. In this case, your command should be the same as if you wanted to execute it from the command interpreter of the operating system of the Agent in charge of the execution. When doing so, your objects become dependant on the platform on which the agent is running. To write an operating system command, select “Operating System” from the list of technologies of you current step.
- ODI Tools or ODI API Commands: ODI offers a broad range of built-in tools that you can use in procedures to perform some specific tasks. These tools include functions for file manipulation, email alerts, event handling, etc. They are described in detail in the online documentation. To use an ODI Tool, select “ODI API” from the list of technologies of your current step.
- Scripting Language: You can write a command in any scripting language supported by the Bean Scripting Framework (BSF) as defined in <http://jakarta.apache.org/bsf/>. The BSF scripting engine is located under the `s/lib/scripting` directory of your ODI installation. You can refer to the BSF documentation for tips on how to customize BSF to support additional scripting languages. By default, ODI includes support for the following scripting languages that you can access from the technology list box of the current step:
 - Python (through Jython). See <http://www.jython.org>
 - JavaScript: web scripting language.
 - Java (through BeanShell). See <http://java.sun.com>

The following figures give examples of commands using the different languages:

Command: Example of SQL Command

Definition Options Version Privileges

Name
Example of SQL Command

Log Counter: <Undefined> Log Level: 3 ☐ Ignore Errors

Command on Target Command on Source

Technology: Teradata Transaction Isolation: <Undefined>
 Context: <Execution Context> Schema: TERADATA_EDW
 Transaction: Autocommit Commit: <Undefined>

Command
 insert into db_test.my_spectral_log
 (log_no, start_date, message)
 values
 (123, current_date, 'Session 123 is starting')

Figure 27: Example of a SQL command

Command: Example of OS Command

Definition Options Version Privileges

Name
Example of OS Command

Log Counter: <Undefined> Log Level: 3 ☐ Ignore Errors

Command on Target Command on Source

Technology: Operating System Transaction Isolation: <Undefined>
 Context: <Execution Context> Schema: <Undefined>
 Transaction: Autocommit Commit: <Undefined>

Command
 sh -c ls -ltr /var/opt > /tmp/filelist.txt

Figure 28: Example of an Operating System Command

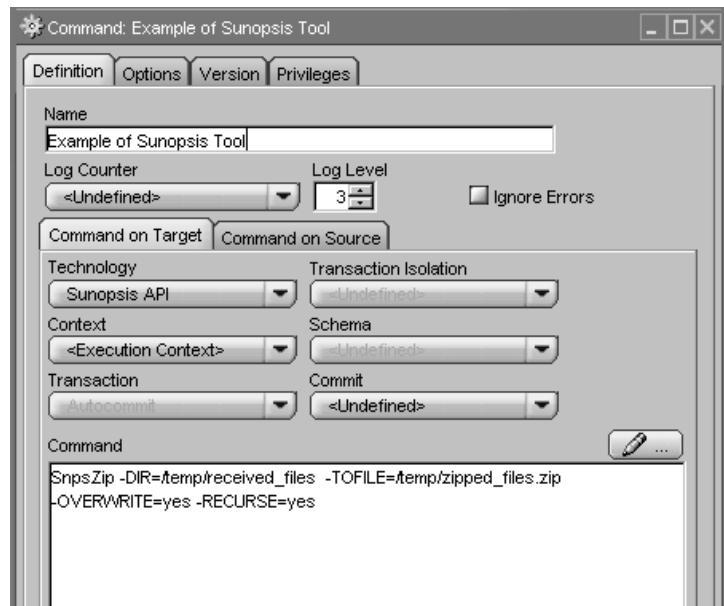


Figure 29: Example of an ODI Tool Command

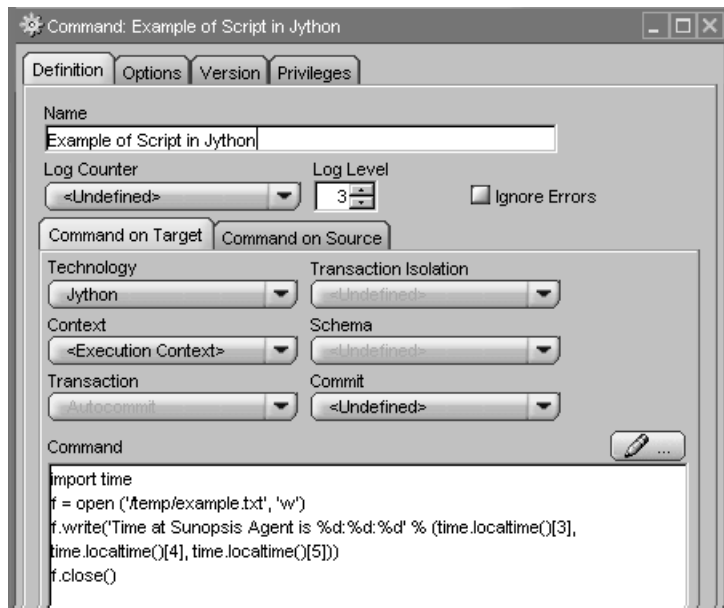


Figure 30: Example of a Scripting Command

Using the Substitution API

It is recommended that you use the ODI substitution API when writing commands in a procedure to keep it independent of the context of execution. You can refer to the online documentation for information about this API. Common uses of the substitution API are given below:

- Use `getObjectName()` to obtain the qualified name of an object in the current logical schema regardless of the execution context, rather than hard coding it.

- Use `getInfo()` to obtain general information such as driver, URL, user etc. about the current step
- Use `getSession()` to obtain information about the current session
- Use `getOption()` to retrieve the value of a particular option of your procedure
- Use `getUser()` to obtain information about the ODI user executing your procedure.

The example below shows the proper way to create a log entry in a log table to trace the current operation:

Rather than writing:

```
insert into dev_log_db.generic_log
(
    session_no,
    operation_type,
    driver_used,
    user_name,
    date_and_time)
values
(
    12345001,
    'Insert Facts',
    'com.ncr.TeradataDriver',
    'JOHN',
    current_timestamp)
```

You should write:

```
insert into <%=snpRef.getObjectName("L","generic_log", "D")%>
(
    session_no,
    operation_type,
    driver_used,
    user_name,
    date_and_time)
values
(
    <%=snpRef.getSession("SESS_NO")%>,
    '<%=snpRef.getOption("OPERATION_TYPE")%>',
    '<%=snpRef.getInfo("DEST_JAVA_DRIVER")%>',
    '<%=snpRef.getUser("USER_NAME")%>',
    current_timestamp
)
```

Using User Options

User options act like parameters that you can use within your steps. It is good practice to define options to improve code reusability. There are 2 types of options:

- Boolean options called Check Boxes. Their value can be used to determine whether individual steps are executed or not. They act like an “if” statement.
- Value and Text options used to pass in short or long textual information respectively.

Within your code, the value of an option can be retrieved using the substitution API `getOption()`. When using your procedure in a package, its values can be set on the step.

Use a Boolean option to determine whether a particular step of your procedure is executed as follows:

1. Define the option for your procedure and set it as type Check Box.
2. Open the procedure.
3. Open the step that you want to execute if and if this option is true.
4. On the Options tab, uncheck the “Always Execute” box and check only this option.
5. When using your procedure in a package, select yes or no as appropriate for that option on the relevant step.

Handling RDBMS Transactions

ODI procedures include an advanced mechanism for transaction handling across multiple steps or even multiple procedures. Transaction handling applies only for RDBMS steps and often depends on the transaction capabilities of the underlying database. Within procedures, you can define for example a set of steps that would be committed or rolled back in case of an error. You can also define up to 9 independent sets of transactions for your steps on the same server. Using transaction handling is of course recommended when your underlying database supports transactions. However, use caution when using this mechanism as it can lead to deadlocks across sessions in a parallel environment.

To understand how you can use the transaction handling mechanism of ODI with your procedures, consider the following example:

Suppose that you want to load some data into table A from tables C and D. In the meanwhile you realize that you need a staging table called B where you want to prepare some of the work. Although it would be recommended to use an interface rather than a procedure to do that, let’s use a procedure for the purpose of the example. Write your procedure as described below:

Step	Action
Step 1	Insert some values from C into table A
Step 2	Create table B
Step 3	Insert some values from C and D into B
Step 4	Update A with some values from B
Step 5	Insert some values from D into A

Step	Action
Step 6	Drop table B

Now, to control your transaction, you want to be able to rollback any action on A if an error occurs. In the meantime, you want to keep track of the data that you have produced in B for debugging purposes. Therefore, steps 1, 4 and 5 should belong to the same transaction unit and steps 2, 3, and 6 should be performed in a separate transaction.

For every RDBMS step in ODI, you can define a transaction number in the “Transaction” list box and a commit behavior in the “Commit” list box. You would then set these options as follow to match the requirements of this example:

Step	Action	Transaction	Commit
Step 1	Insert into table A some values from C	Transaction 0	No Commit
Step 2	Create table B	Transaction 1	No Commit
Step 3	Insert into B some values from C,D	Transaction 1	Commit
Step 4	Update A with some values from B	Transaction 0	No Commit
Step 5	Insert into A some values from D	Transaction 0	Commit
Step 6	Drop table B	Transaction 1	Commit

Steps 1, 4 and 5 belong to the same transaction (0) and are committed on step 5 e.g. when steps 1, 2, 3, 4 and 5 are successful. In case of any error prior to step 5, all opened transactions will be rolled back. If an error occurs in step 4 or 5, steps 1, 4 and 5 will be rolled back, whereas steps 2 and 3 will remain committed as they belong to the same transaction unit (1). This means that the data in B will remain in the table, even if the data of A was cancelled.

At runtime, transactions IDs are translated by the ODI Agent into distinct open connections to the RDBMS during the execution of the session. This means that if step 5 fails to commit, the transaction 0 would have remained opened until committed by another step of another procedure or at the end of the session. This feature allows you to chain transactions across multiple procedures within the same package.

Notes:

If you don't require transactions for your procedure, set the transaction to “Autocommit” to avoid opening additional connections to the database.
If you want to commit a transaction every 1000 rows, you should design your statement

as a Select/Insert statement as defined in section “Using Select on Source / Action on Target”, and set the Commit mode to “Commit 1000 rows”.
Opening several transaction channels requires having as many JDBC connections to the database. Do this with caution.

Using Select on Source / Action on Target

ODI includes a mechanism in procedures that allow you to use implicit cursors to perform an action for every row returned by a SQL SELECT statement. To do so, you simply need to specify the SELECT statement in the source tab of your procedure step and the action code in the target tab. The action code can itself be an INSERT, UPDATE or DELETE SQL statement or any other code such as an ODI API commands or Jython. You can refer to “SunopsisODI Tools or API Commands” for details on how to use the ODI API commands. The source SELECT statement will be fetched using the “array fetch size” defined for the data server in the topology. When the target statement is a SQL DML statement, ODI agent will use the “batch update size” value of the target data server to perform the DML.

The values returned by the source result set can be referred to in the target part using the column names as defined for the SELECT statement. They should be prefixed by colons “:” whenever used in a target DML statement and will act as “bind variables”. If the target statement is not a DML statement, then they should be prefixed by a hash “#” sign and will act as substituted variables.

The following examples give you common uses for this mechanism. There are, of course, many other applications for this powerful mechanism.

Example1: Loading Data from a Remote SQL Database

Suppose you want to insert data into the Teradata PARTS table from an Oracle PRODUCT table. The following table gives details on how to implement this in a procedure step:

Source Technology	Oracle
Source Logical Schema	ORACLE_INVENTORY
Source Command	<pre>select PRD_ID MY_PRODUCT_ID, PRD_NAME PRODUCT_NAME, from PRODUCT</pre>
Target Technology	Teradata
Target Logical Schema	TDAT_EDW
Target Command	<pre>insert into PARTS (PART_ID, PART_ORIGIN, PART_NAME) values (:MY_PRODUCT_ID, 'Oracle Inventory', :PRODUCT_NAME)</pre>

ODI will implicitly loop over every record returned by the SELECT statement and bind its values to “:MY_PRODUCT_ID” and “:PRODUCT_NAME” bind variables. It then triggers the INSERT statement with these values after performing the appropriate data type translations.

When batch update and array fetch are supported by the target and source technologies respectively, ODI prepares arrays in memory for every batch, making the overall transaction more efficient.

Notes:

This mechanism is known to be far less efficient than a fast or multi load in the target table. You should only consider it for very small volumes of data.
The section *Using the Agent* in the Loading Strategies further discusses this mechanism.

Example2: Dropping a List of Tables

In this example, suppose you want to drop all tables that start with ‘TMP_’ in the staging area database. The following table shows how to implement this in a procedure step:

Source Technology	Teradata
Source Logical Schema	TDAT_STAGING
Source Command	Select DatabaseName, TableName From DBC.Tables Where DatabaseName = 'db_staging' And TableName like 'TMP_'
Target Technology	Teradata
Target Logical Schema	TDAT_STAGING
Target Command	Drop table #DatabaseName.#TableName

Variables “#DatabaseName” and “#TableName” will be substituted with the corresponding value for every record returned by the SELECT statement. In this particular case, you can’t use bind variables as they are not supported in SQL DDL statements.

Notes:

This example is provided only to illustrate the select/action mechanism. Use it at your own risk, as dropping tables from a database may be irreversible.

Example3: Sending Multiple Emails

Suppose you have a table that contains information about all the people that need to be warned by email in case of a problem during the loading of your Data Warehouse. You can do it using a single procedure step as follow:

Source Technology	Teradata
Source Logical Schema	TDAT_TECHNICAL
Source Command	Select FirstName, Email From Operators Where RequireWarning = 'Yes'
Target Technology	ODI API
Target Logical Schema	None
Target Command	SnpsSendMail -MAILHOST=my.smtp.com - FROM=admin@acme.com "-TO=#Email" "-SUBJECT=Job Failure" Dear #FirstName, I'm afraid you'll have to take a look at ODI Operator, because session <%=snpsRef.getSession("SESS_NO")%> has just failed! -Admin

The “-TO” parameter will be substituted by the value coming from the “Email” column of your source SELECT statement. The “SnpsSendMail” command will therefore be triggered for every operator registered in the “Operators” table.

Example4: Starting Multiple Scenarios in Parallel

You will often need to execute several ODI scenarios either in parallel or synchronously from another scenario. Again, using this Select/Action mechanism can help achieve that very easily with a minimum of code. Simply define a table that contains the names of the scenarios you want to run and develop your procedure as follows:

Source Technology	Teradata
Source Logical Schema	TDAT_TECHNICAL
Source Command	Select ScenarioName, ScenarioVersion, ExecMode From MyScenarioList Where BusinessArea = 'LOAD_DIMENSIONS'
Target Technology	ODI API
Target Logical Schema	None
Target Command	SnpsStartScen "-SCEN_NAME=#ScenarioName" - SCEN_VERSION=#ScenarioVersion "-SYNC_MODE=#ExecMode"

With this example, to execute your scenario in synchronous mode, set the ExecMode column to 1 in the “MyScenarioList” table. Otherwise, set it to 2 to cause the scenario to be run asynchronously.

Note:

This example demonstrates a powerful way of starting several scenarios in parallel. If you want to wait until all the scenarios are completed before performing the next step of your procedure or your package, you can use the `SnpsWaitForChildSessions` API command as described in the online documentation.

Common Pitfalls

- Although you can perform transformations in procedures, using these for this purpose is not recommended; use interfaces instead.
- If you start writing a complex procedure to automate a particular recurring task for data manipulation, you should consider converting it into a Knowledge Module. See section “SunopsisODI Knowledge Modules” for details.
- Whenever possible, try to avoid operating-system-specific commands. Using them makes your code dependant on the operating system that runs the agent. The same procedure executed by 2 agents on 2 different operating systems (such as Unix and Windows) will not work properly.

Using Variables

Using Variables is highly recommended to create reusable packages, interfaces and procedures. Variables can be used everywhere within ODI. Their value can be stored persistently in the ODI Repository if their action type is set to “Historize” or “Last Value”. Otherwise, with an action type of “Non-Persistent”, their value will only be kept in the memory of the agent during the execution of the current session.

Referring to variable `MY_VAR` in your objects should be done as follows:

- `#MY_VAR`: With this syntax, the variable must be in the same project as the object referring to it. Its value will be substituted. To avoid ambiguity, consider using fully qualified syntax by prefixing the variable name with the project code.
- `#MY_PROJECT_CODE.MY_VAR`: Using this syntax allows you to use variables across projects. It prevents ambiguity when 2 variables in different projects have the same name. The value of the variable will be substituted at runtime.
- `#GLOBAL.MY_VAR`: This syntax allows you to refer to a global variable. Its value will be substituted in your code. Refer to section *Global Objects* for details.
- Using “`:`” instead of “`#`”: You can use the variable as a SQL bind variable by prefixing it with a colon rather than a hash. However this syntax is subject to

restrictions as it only applies to SQL DML statements. Refer to ODI online documentation for more information about bind variables.

Declaring a Variable

If some of the objects used in your package refer to a variable that you plan to set at runtime through another scenario, you should declare it in your package to have ODI allocate the appropriate memory space in the agent. Simply drag and drop your variable into the package and select the “Declare Variable” option in the properties panel.

Other variables that you explicitly use in your packages for setting, refreshing or evaluating their value do not need to be declared.

Assigning a Value to a Variable

ODI lets you assign a value to a variable in several different ways:

- Retrieving the variable value from a SQL SELECT statement: When creating your variable, define a SQL statement to retrieve its value. For example, you can create variable NB_OF_OPEN_ORDERS and set its SQL statement to:

```
select COUNT(*) from ORDERS where STATUS = 'OPEN'.
```

 Then in your package, you will simply drag and drop your variable and select the “Refresh Variable” option in the properties panel. At runtime, ODI agent will execute the SQL statement and assign the first returned value of the result set to the variable.
- Explicitly setting the value in a package: You can also manually assign a value to your variable for the scope of your package. Simply drag and drop your variable into your package and select the “Set Variable” and “Assign” options in the properties panel as well as the value you want to set.
- Incrementing the value: Incrementing only applies to variables defined with a numeric data type. Drag and drop your numeric variable into the package and select the “Set Variable” and “Assign” options in the properties panel as well as the desired increment.
- Assigning the value at runtime: When you start a scenario generated from a package containing variables, you can set the values of its variables. You can do that in the StartScenario command by specifying the VARIABLE=VALUE list. Refer to the SnpsStartScen API command and section *Executing a scenario from an OS command* in the online documentation.

Evaluating the Value of a Variable

The value of a variable can also be used to perform conditional branching in a package. For example, you can choose to execute interfaces A and B of your package only if variable EXEC_A_AND_B is set to “YES”, otherwise you would execute interfaces B and C. To do this, you would simply drag and drop the variable in your package, and select “Evaluate Variable” in the properties panel as illustrated in the figure below:

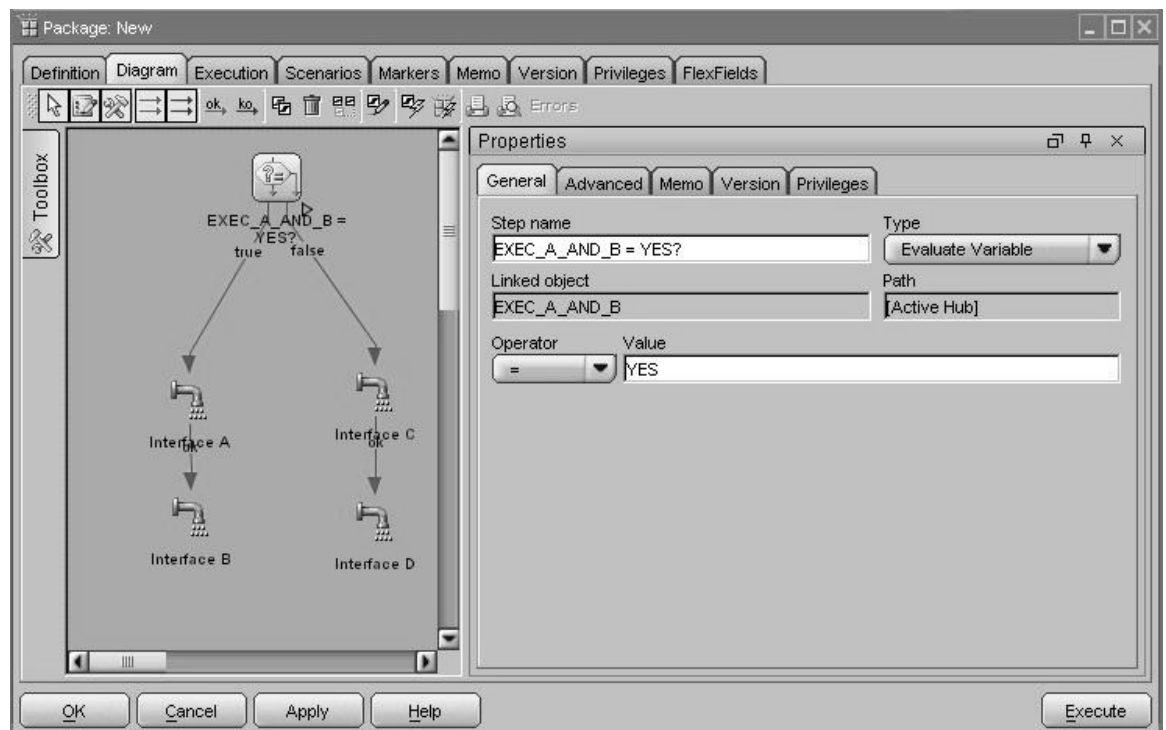


Figure 31: Example of Variable Evaluation

Evaluating variables in a package allows great flexibility in designing reusable, complex workflows.

Use Cases for Variables

Using Variables in Interfaces

You can refer to variables in your interfaces in:

- Mapping expressions
- Filter expressions
- Join expressions
- As a value for a textual option of a Knowledge Module

The following table gives you examples of these:

Type	Expression	Description
Mapping	'#PRODUCT_PREFIX' PR.PRODUCT_ID	Concatenates the current project's product prefix variable with the product ID. As the value of the variable is substituted, you need to

Type	Expression	Description
		enclose the variable with quotes.
Join	CUS.CUST_ID = #DEMO.UID * 1000000 + FF.CUST_NO	Multiply the value of the UID variable of the DEMO project by 1000000 and add the CUST_NO column before joining it with the CUST_ID column.
Filter	ORDERS.QTY between #MIN_QTY and #MAX_QTY	Filter orders according to the MIN_QTY and MAX_QTY thresholds.
Option Value	TEMP_FILE_NAME: #DEMO.FILE_NAME	Use the FILE_NAME variable as the value for the TEMP_FILE_NAME option.

Using Variables in Procedures

You can use variables anywhere within your procedures' code as illustrated in the example below:

Step Id	Step Type	Step Code	Description
1	SQL	Insert into #DWH.LOG_TABLE_NAME Values (1, 'Loading Step Started', current_date)	Add a row to a log table of which the name is only known at runtime
2	Jython	f = open('#DWH.LOG_FILE_NAME', 'w') f.write('Inserted a row in table %s' % ('#DWH.LOG_TABLE_NAME')) f.close()	Open file defined by LOG_FILE_NAME variable and write the name of the log table into which we have inserted a row.

You should consider using options rather than variables whenever possible in procedures. Options act like input parameters. Therefore, when executing your procedure in a package you would set your option values to the appropriate variables.

In the preceding example, you would write "Step 1"'s code as follows:

```
Insert into <%=snpRef.getOption("LogTableName")%>
Values (1, 'Loading Step Started', current_date)
```

Then, when using your procedure as a package step, you would set the value of option LogTableName to #DWH.LOG_TABLE_NAME.

Using Variables in Packages

Variables can be used in packages for different purposes:

- Declaring a variable,
- Refreshing a variable value from the underlying SQL SELECT statement,
- Setting its value manually,
- Incrementing a numeric value,
- Evaluating the value for conditional branching

All these types are available when you drag and drop a variable into a package.

Using Variables within Variables

It is sometimes useful to have variables depend on other variable values, as follows:

Variable Name	Variable Details	Description
STORE_ID	Alphanumeric variable. Passed as a parameter to the scenario	Gives the ID of a store
STORE_NAME	Alphanumeric variable. Select statement: <code>Select name From Stores Where id = '#DWH.STORE_ID'</code>	The name of the current store is derived from the ID using a SELECT statement on the Stores table.

In this example, you would build your package as follow:

1. Drag and drop the STORE_ID variable to declare it. This would allow you to pass it to your scenario at runtime.
2. Drag and drop the STORE_NAME variable to refresh its value. When executing this step, the agent will run the select query with the appropriate STORE_ID value. It will therefore retrieve the corresponding STORE_NAME value.
3. Drag and drop the other interfaces or procedures that use any of these variables.

Using Variables in the Resource Name of a Datastore

You may face some situations where the names of your source or target datastores are dynamic. A typical example of this is when you need to load flat files into your Data Warehouse with a file name composed of a prefix and a dynamic suffix such as the current date. For example the order file for March 26 would be named “ORD2006.03.26.dat”.

To develop your loading interfaces, you would follow these steps:

1. Create the FILE_SUFFIX variable in you DWH project and set its SQL SELECT statement to select current_date (or any appropriate date transformation to match the actual file suffix format)

2. Define your ORDERS file datastore in your model and set its resource name to: “ORD#DWH.FILE_SUFFIX.dat”.
3. Use your file datastore normally in your interfaces.
4. Design a package as follows:
 - a. Drag and drop the FILE_SUFFIX variable to refresh it.
 - b. Drag and drop all interfaces that use the ORDERS datastore.

At runtime, the source file name will be substituted to the appropriate value.

Notes:

The variable in the datastore resource name must be fully qualified with its project code.

When using this mechanism, it is not possible to view the data of your datastore from within Designer.

Using Variables in a Server URL

There are some cases where using contexts for different locations is less appropriate than using variables in the URL definition of your data servers. For example, when the number of sources is high (> 100), or when the topology is defined externally in a separate table. In these cases, you can refer to a variable in the URL of a server’s definition.

Suppose you want to load your warehouse from 250 Oracle source applications used within your stores. Of course, one way to do it would be to define one context for every store. However, doing so would lead to a complex topology that would be difficult to maintain. Alternatively, you could define a table that references all the physical information to connect to your stores and use a variable in the URL of your data server’s definition. The following example illustrates how you would implement this in ODI:

1. Create a StoresLocation table as follow:

StoreId	StoreName	StoreURL	IsActive
1	Denver	10.21.32.198:1521:ORA1	YES
2	San Francisco	10.21.34.119:1525:SANF	NO
3	New York	10.21.34.11:1521:NY	YES
Etc.

2. Create 3 variables in your EDW project:

- STORE_ID: takes the current store ID as an input parameter
- STORE_URL: refreshes the current URL for the current store ID with
SELECT statement: `select StoreUrl from StoresLocation where
StoreId = #EDW.STORE_ID`

- **STORE_ACTIVE**: refreshes the current activity indicator for the current store ID with SELECT statement: `select IsActive from StoresLocation where StoreId = #EDW.STORE_ID`

3. Define one physical data server for all your stores and set its JDBC URL to:

`jdbc:oracle:thin:@#EDW.STORE_URL`

4. Define your package for loading data from your store as described in the figure below:

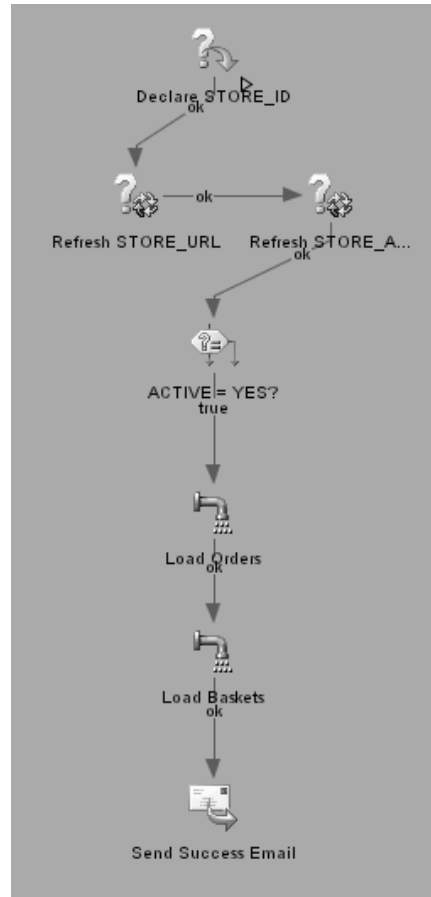


Figure 32: Example of a Package Using a Dynamic URL

The input variable **STORE_ID** will be used to refresh the values for **STORE_URL** and **STORE_ACTIVE** variables from the **StoresLocation** table. If **STORE_ACTIVE** is set to “YES”, then the next 3 steps will be triggered. The interfaces refer to source datastores that the agent will locate according to the value of the **STORE_URL** variable.

To start such a scenario on Unix for the New York store, you would issue the following operating system command:

```
startscen.sh LOAD_STORE 1 PRODUCTION "EDW.STORE_ID=3"
```

If you want to trigger your LOAD_STORE scenario for all your stores in parallel, you would simply need to create a procedure with a single SELECT/action command as follows:

Source Technology	Teradata
Source Logical Schema	TDAT_TECHNICAL
Source Command	Select StoreId From StoresLocation
Target Technology	ODI API
Target Logical Schema	None
Target Command	SnpsStartScen "-SCEN_NAME=LOAD_STORE" "-SCEN_VERSION=1" "- SYNC_MODE=2" "-EDW.STORE_ID=#StoreId"

The LOAD_STORE scenario will then be executed for every store with the appropriate STORE_ID value. The corresponding URL will be set accordingly.

Refer to sections *Using Select on Source / Action on Target* and *Setting up Agents* for further details.

Using Sequences

Sequences can be used to generate an automatic counter. The current value of the counter is stored either in an ODI repository table or in a column of any table that you specify.

For the sequences to be incremented, the data needs to be processed row-by-row by the agent. Therefore, using sequences is not recommended when dealing with large numbers of records. You would, in this case, use database-specific sequences such as identity columns in Teradata, IBM DB2, Microsoft SQL Server or sequences in Oracle.

To use a sequence in your mappings, filters or joins, use one of these notations:

1. #<PROJECT_CODE>.<SEQUENCE_NAME>_NEXTVAL for substitution mode.
2. :<PROJECT_CODE>.<SEQUENCE_NAME>_NEXTVAL to use it as a SQL bind variable.

You also need to make sure that your INSERT or UPDATE statements as defined in your Knowledge Module use the ODI specific select/action syntax rather than the insert/select or update/select syntax.

Refer to “Using Select on Source / Action on Target” and to “SunopsisODI Knowledge Modules” for details.

Using User Functions

User functions are an advanced feature for reusing your transformations across interfaces and technologies. Using them is recommended in your projects for code sharing and reusability. They facilitate the maintenance and the portability of your developments across different target platforms.

User functions can be used anywhere in mappings, joins, filters and conditions.

The following example illustrates how to implement a user function that would be translated into code for different technologies:

Suppose you want to define a function that, given a date, gives you the name of the month. You want this function to be available for your mappings when executed on Oracle, Teradata or Microsoft SQL Server. The table below shows how to implement this as a user function:

Function Name	GET_MONTH_NAME
Function Syntax	GET_MONTH_NAME(\$ (date_input))
Description	Retrieves the month name from a date provided as date_input
Implementation for Oracle	<code>Initcap(to_char(\$ (date_input), 'MONTH'))</code>
Implementation for Teradata	<pre>case when extract(month from \$(date_input)) = 1 then 'January' when extract(month from \$(date_input)) = 2 then 'February' when extract(month from \$(date_input)) = 3 then 'March' when extract(month from \$(date_input)) = 4 then 'April' when extract(month from \$(date_input)) = 5 then 'May' when extract(month from \$(date_input)) = 6 then 'June' when extract(month from \$(date_input)) = 7 then 'July' when extract(month from \$(date_input)) = 8 then 'August' when extract(month from \$(date_input)) = 9 then 'September' when extract(month from \$(date_input)) = 10 then 'October' when extract(month from \$(date_input)) = 11 then 'November' when extract(month from \$(date_input)) = 12 then 'December' end</pre>
Implementation for Microsoft SQL	<code>datename(month, \$(date_input))</code>

You can now use this function safely in your interfaces for building your mappings, filters and joins. ODI will generate the appropriate code depending on the execution location of your expression.

For example, if you define mapping:

`substring(GET_MONTH_NAME(CUSTOMER.LAST_ORDER_DATE), 1, 3)`, ODI will generate code similar to the following, depending on your execution technology:

Oracle	<code>substring(Initcap(to_char(CUSTOMER.LAST_ORDER_DATE 'MONTH')) , 1, 3)</code>
Implementation for Teradata	<code>substring(case when extract(month from CUSTOMER.LAST_ORDER_DATE) = 1 then 'January' when extract(month from CUSTOMER.LAST_ORDER_DATE) = 2 then 'February' ... end, 1, 3)</code>
Implementation for Microsoft SQL	<code>substring(datetime(month, CUSTOMER.LAST_ORDER_DATE) , 1, 3)</code>

Of course, user functions can call other user functions.




Building Workflows with Packages











Introduction to packages

Packages are the main objects used to generate scenarios for production. They represent the workflow that will be processed by the agent. Every step in your package has options that control the execution flow:

- Step name: By default, the name of the object on which the step is based
- Next step on success: step to execute in case of a success
- Next step on failure: step to execute in case of failure. After a variable evaluation, the “step on failure” is executed if the condition is not matched.
- Number of attempts in case of failure: Number of times the agent will try to re-execute the step if it fails.
- Time between attempts: Interval in seconds that an agent waits before re-executing the step if it fails.
- Log Step in the Journal: whether the events within the step should be kept in the log even if it succeeds.



Steps within your package are designed by dragging and dropping objects in the diagram window, and by linking them with green (on success) or red (on failure) links. The following steps can be added to your diagram:



















	Step Type	Description
	Interface	Interfaces where you have designed your business rules
	Procedure	Procedures that you have designed
	ODI Tool	See “SunopsisODI Tools or API Commands”

















		for all available tools
	Declare Variable	Declares a variable in your package
	Refresh Variable	Refreshes the value of a variable from its SQL SELECT statement
	Set or Increment Variable	Sets or increments the value of a variable
	Evaluate Variable	Evaluates the value of a variable
	Datastore Check	Performs a static check for a data quality audit of a datastore
	Sub-model Check	As above, but for all datastores within a sub-model.
	Model Check	As above, but for a whole model
	Journalizing Datastore	Performs journalizing operations on a datastore such as starting/stopping journal, adding/locking/unlocking subscribers, etc.
	Journalizing Model	As above, but for a whole model.
	Model Reverse	Automates a customized reverse engineering.



ODI Tools or API Commands

ODI includes a set of tools, also known as API commands, that you can either use in the commands of your procedures or graphically when designing your packages. These tools tremendously reduce your development efforts for common tasks you would like to design. Refer to ODI online documentation for details on how to use these tools. The table below gives an overview of all the tools available:

	Tool Name	Description
	Operating System	Executes an operating system command
	SnpsAnt	Calls the building tool Ant of the "The Apache Software Foundation". This tool can be used to run several commands such as archiving, compiling, deploying, documenting, EJBs, run OS processes, manipulate files, .NET task, remote tasks,

	Tool Name	Description
		etc.
	SnpsBeep	Produces a beep sound on the machine hosting the agent.
	SnpsDeleteScen	Deletes a scenario given its name and version.
	SnpsExecuteWebService	Executes a web service given its WSDL file.
	SnpsExportAllScen	Exports scenarios from projects or folders into several XML files.
	SnpsExportObject	Exports an ODI object into an XML file, given its ID.
	SnpsExportScen	Exports a scenario into an XML file.
	SnpsFileAppend	Concatenates a set of files into a single file.
	SnpsFileCopy	Copies files and folders.
	SnpsFileDelete	Removes files and folders.
	SnpsFileMove	Moves files and folders.
	SnpsFileWait	Waits for a set of files in a folder.
	SnpsGenerateAllScen	Generates a set of scenarios from a folder or project.
	SnpsImportScen	Imports a scenario from an XML file.
	SnpsKillAgent	Stops an ODI agent
	SnpsMkDir	Creates a folder.
	SnpsOutFile	Creates a text file.
	SnpsPingAgent	Connects to an agent to check its availability.
	SnpsPurgeLog	Purges ODI logs.

	Tool Name	Description
	SnpsReadMail	Waits for incoming emails and extracts them.
	SnpsRefreshJournalCount	Refreshes, for a given subscriber, the number of rows to consume for the given table list or CDC set.
	SnpsReinitializeSeq	Re-initializes an ODI sequence.
	SnpsRetrieveJournalData	Retrieves journal data for asynchronous third tier Change Data Capture provider.
	SnpsReverseGetMetaData	Reverse engineers a model.
	SnpsReverseResetTable	Resets the SNP_REV_XXX tables to prepare a customized reverse engineering strategy.
	SnpsReverseSetMetaData	Updates metadata in the work repository from SNP_REV_XXX tables.
	SnpsSAPALEClient	Executes an SAP ALE Client call to send iDOCs to SAP/R3.
	SnpsSAPALEServer	Runs an SAP ALE Server to receive incoming iDOCs from SAP/R3.
	SnpsSendMail	Sends an email.
	SnpsSleep	Waits for a specified amount of time.
	SnpsSqlUnload	Unloads data from a SQL SELECT statement to an ASCII file. This command is very useful for unloading massive data from a database to a flat file.
	SnpsStartScen	Starts an ODI scenario.
	SnpsUnzip	Unzips a zip file into a folder.
	SnpsWaitForChildSession	Waits for sessions started asynchronously from the current session to complete.
	SnpsWaitForData	Captures data changes in a table according to a filter.

	Tool Name	Description
	SnpsWaitForLogData	Captures events produced from the Change Data Capture framework.
	SnpsZip	Compresses files and folders.

Packages Examples

Performing a Loop in a Package

You may want to perform a loop inside your package, for example to process incoming files from 4 different providers. The example here illustrates how you would do it in ODI:

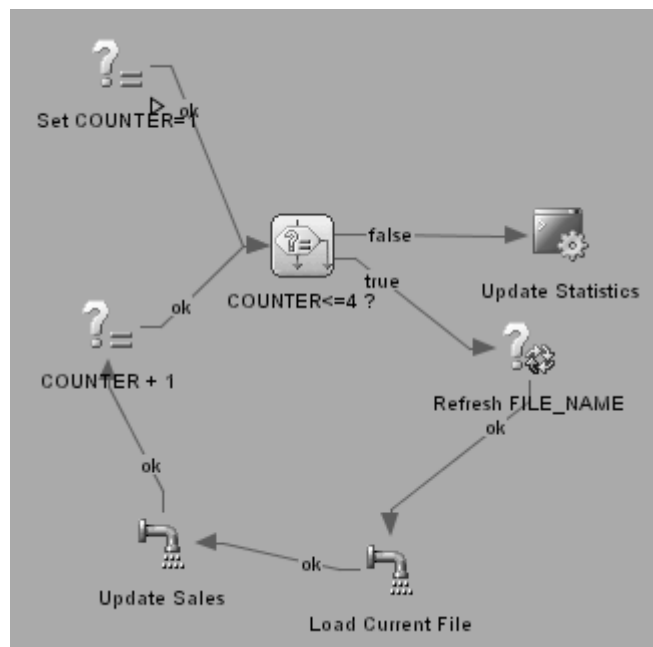


Figure 33: Example of a Package Loop

Step Name	Step Type	Description
Set COUNTER=1	Set Variable	Sets the numeric COUNTER variable to 1 at the starting of the package.
COUNTER<=4?	Evaluate variable	Evaluates the value of the COUNTER variable. If it is greater than 4, it executes the “Update Statistics” step and terminates. Otherwise, it executes the “Refresh FILE_NAME” step.
Refresh FILE_NAME	Refresh Variable	Refreshes the value of the FILE_NAME variable given the value of the COUNTER variable.

Step Name	Step Type	Description
Load Current File	Interface	Loads and transforms the current file. The source datastore name is dynamically resolved. Refer to section “Using Variables in the Resource Name of a Datastore” for details.
Update Sales	Interface	Another transformation process.
COUNTER + 1	Increment Variable	Increments the value of the COUNTER variable.
Update Statistics	Procedure	Executes a procedure when the loop is terminated.

Notes:

You should consider this solution only for small loops (less than 10 occurrences). For large loops, you would rather use the select/action mechanism as described in section “Using Select on Source / Action on Target”.

Starting Scenarios from a Package

You may want to start one or several scenarios in parallel from a single scenario and wait until their execution is terminated before sending a completion email to the administrator. The figure below illustrates this example:

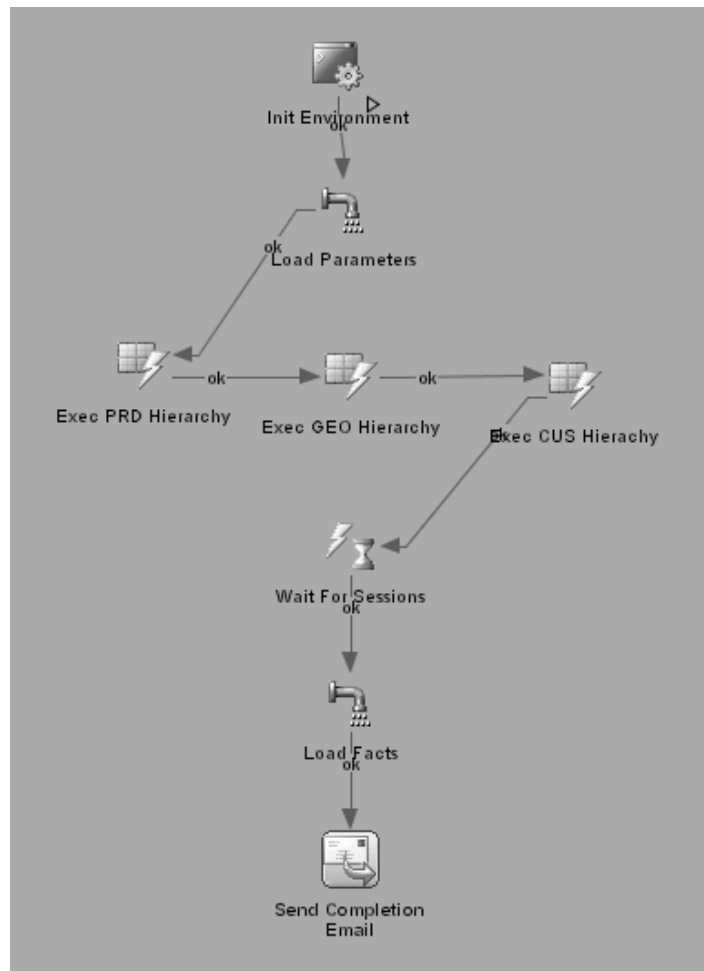


Figure 34: Example of a Package Starting Multiple Scenarios

Step Name	Step Type	Description
Init Environment	Procedure	Executes a procedure that performs some initial settings.
Load Parameters	Interface	Executes an interface that loads a parameter table.
Exec PRD Hierarchy	Tool SnpsStartScen	Starts a scenario that loads the product dimension hierarchy in asynchronous mode (SYNC_MODE = 2)
Exec GEO Hierarchy	Tool SnpsStartScen	Starts a scenario that loads the geography dimension hierarchy in asynchronous mode (SYNC_MODE = 2)
Exec CUS Hierarchy	Tool SnpsStartScen	Starts a scenario that loads the customer dimension hierarchy in asynchronous mode

Step Name	Step Type	Description
		(SYNC_MODE = 2)
Wait For Sessions	Tool SnpsWaitForChildSession	Waits until all child sessions are terminated. The next step will be triggered only when the 3 preceding scenarios have completed successfully.
Load Facts	Interface	Loads the facts table
Send Completion Email	Tool SnpsSendMail	Sends an email to the administrator

In this example, the 3 scenarios are executed in parallel, as the SYNC_MODE parameter of the SnpsStartScen command is set to 2. The SnpsWaitForChildSession step waits until all of them have completed before continuing with execution. The parameter MAX_CHILD_ERROR of this command lets you define the behavior in case of an error of any or all of the child sessions.

If you want to start a high number of scenarios in parallel according to a table containing their names and parameters, you should consider doing it in a procedure that contains a select/action command as described in section “Example4: Starting Multiple Scenarios in Parallel”. Your package will therefore appear as follows:

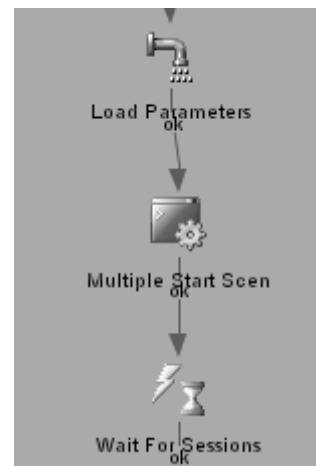


Figure 35: Starting a Batch of Scenarios

Waiting for incoming files

This example shows how you would process files that are sent to you in a predefined directory. The file names are dynamic but all have the same format. After processing the file, you also want to compress it and store it in a separate directory.

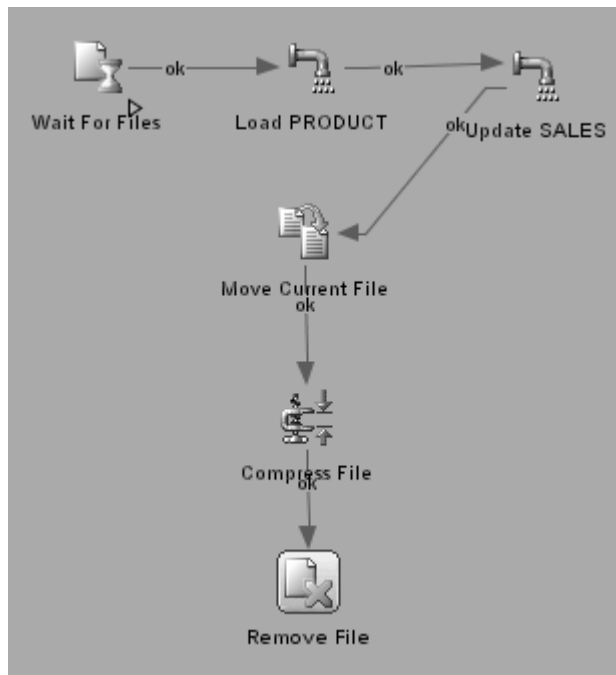


Figure 36: Example of a Package to Wait for Files

Step Name	Step Type	Description
Wait For Files	Tool SnpsFileWait	Waits for an incoming file name PRD*.dat in the /data/files/incoming directory and moves it to the /data/files/current/PRODUCT.dat file.
Load PRODUCT	Interface	Loads and transforms the file /data/files/current/PRODUCT.dat.
Update SALES	Interface	Updates sales information
Move Current File	Tool SnpsFileMove	Moves and renames the /data/files/current/PRODUCT.dat to /data/files/processed/PRD<SessionNumber>.dat file. The target file name can be expressed with an ODI substitution API that retrieves the current session number as follow: PRD<%=snpRef.getSession("SESS_NO")%>.dat
Compress File	Tool SnpsZip	Compresses the file PRD<%=snpRef.getSession("SESS_NO")%>.dat into PRD<%=snpRef.getSession("SESS_NO")%>.zip
Remove File	Tool SnpsFileDelete	Deletes the file PRD<%=snpRef.getSession("SESS_NO")%>.dat as it has been archived.

Once your scenario is generated from this package, you would schedule it to run every 5 minutes either through the ODI built-in scheduler or using an external scheduler.

The `SnpsFileWait` command can also wait for several files and concatenate their content into a single large file. For example, if you know that you will asynchronously receive 25 files from your subsidiaries, you would set the `FILE_COUNT` parameter to 25 and the `ACTION` parameter to `APPEND`. Therefore, rather than loading and processing 25 files, you would only load a single large file in a more efficient and set-oriented way.

Organizing Objects with Markers

Designer offers an elegant way of tagging your objects with markers. These help you organize your workspace by setting graphical or textual indicators on each of your objects. When organizing your development team, you can agree on the meaning of markers to help you maintain a comprehensive metadata repository.

Refer to the online documentation for additional information.

Global Objects

ODI allows you to define global variables, sequences and user functions. These global objects are available for all your projects.

Although you may find it useful to use global objects, you should use caution, due to the following potential side effects:

- Modifying a global object can affect several projects and teams. Therefore, all teams must agree before updating a single object.
- Releasing or versioning your projects and models may require the referenced global objects to be released as well.

Objects Naming Conventions

Your projects are shared by your teams. Every developer should be able to maintain existing objects with seamless efforts. It is therefore important to have every object match a naming convention so that it can be referenced appropriately. The following rules give you a starting point for your naming conventions:

Object Type	Naming Rule	Examples
Project	<p>Project names should match your internal conventions. It is recommended that you split large projects into smaller ones.</p> <p>The project code should be in upper case and meaningful enough as it is used to prefix the name of variables.</p>	<p>Name: EDW – Financial</p> <p>Code: EDW_FINANCE</p>

Object Type	Naming Rule	Examples
Folder	Folders can be named freely. Try to avoid nested folders and references to objects across folders.	Financial Accounts
Variable	Variables' names should be in upper case. You can choose to include the data type of the variable in its name for better readability. Types can be V for Varchar, D for Date, and N for numeric. <TYPE>_<VAR_NAME>	V_FILE_NAME N_MAX_AGE D_LAST_UPDATE
Sequence	Sequences' names should be in upper case. To avoid confusion with variables, you can prefix your names with SEQ_. SEQ_<SEQ_NAME>	SEQ_PRD_COUNTER
User Function	User functions' names should be in upper case. <FUNCTION_NAME>	COMPUTE_COST() GET_FULL_NAME()
Knowledge Module	<p>Loading KMs: LKM <SOURCE> to <TARGET> (<METHOD>)</p> <p>Integration KMs; IKM <STAGING> to <TARGET> <METHOD> (<DETAILS>)</p> <p>IKM <TARGET> <METHOD> (<DETAILS>)</p> <p>Reverse KMs: RKM <TARGET></p> <p>Check KMs: CKM <TARGET> (<DETAILS>)</p>	<p>LKM File to Teradata (FastLoad) LKM Oracle to Teradata (Unload – FastLoad)</p> <p>IKM SQL to Teradata Append (MultiLoad)</p> <p>IKM Teradata Incremental Update</p> <p>RKM Oracle</p> <p>CKM Teradata</p>

Object Type	Naming Rule	Examples
	Journalizing KMs: JKM <TARGET> (Simple or Consistent)	JKM Teradata Simple JKM Oracle Consistent
Interface	For readability, you may wish to include the target table name in the name of your interfaces: Int. <TARGET_TABLE> <DETAILS>	Int. EDW_SALES Append Int. PRODUCT Upsert
Procedure	Procedure names are free	Update Statistics
Package	Package names should be in upper case and white spaces should be avoided. The scenario names are derived from package names. <PACKAGE_NAME>	LOAD_DIM_PRD
Scenario	Scenario names should always be in upper case with no spaces. Their name should match the package name that generated them. <PACKAGE_NAME>	LOAD_DIM_PRD

ODI KNOWLEDGE MODULES

Introduction to Knowledge Modules

Knowledge modules (KMs) are templates of code that generate code depending on how they are used. The code in the KMs is described exactly as it would be in a procedure, except that it is generically dedicated to a specific task using the ODI Substitution API. Like procedures, KMs contain sequences of commands (in SQL, shell, native language, etc.) and substitution tags. They are used to generate code from business rules and metadata.

However, there are some important differences between a KM and a procedure:

- The same KM can be reused across several interfaces or models. To modify the behavior of hundreds of jobs using procedures, developers would need to open every procedure and modify it. The same effect would only require modifying the KM used by your interfaces or models once.
- KMs are generic. 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 as procedures. They require metadata from interfaces, datastores and models.

KMs fall into 5 different categories, each of them solving a part of a process as summarized in the table below:

Knowledge Module	Description	Where used
Loading KM	Loads heterogeneous data to a staging area	Interfaces with heterogeneous sources
Integration KM	Integrates data from the staging area to a target	Interfaces
Check KM	Check consistency of data against constraints	<ul style="list-style-type: none">• Models, sub models and datastores for data quality audit• Interfaces for flow control or static control
Reverse-engineering KM	Retrieves metadata from a metadata provider to the ODI work repository	Models to perform a customized reverse-engineering
Journalizing KM	Creates the Change Data Capture framework objects in the staging area	Models, sub models and datastores to create, start and stop journals and to register subscribers.

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 business 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.

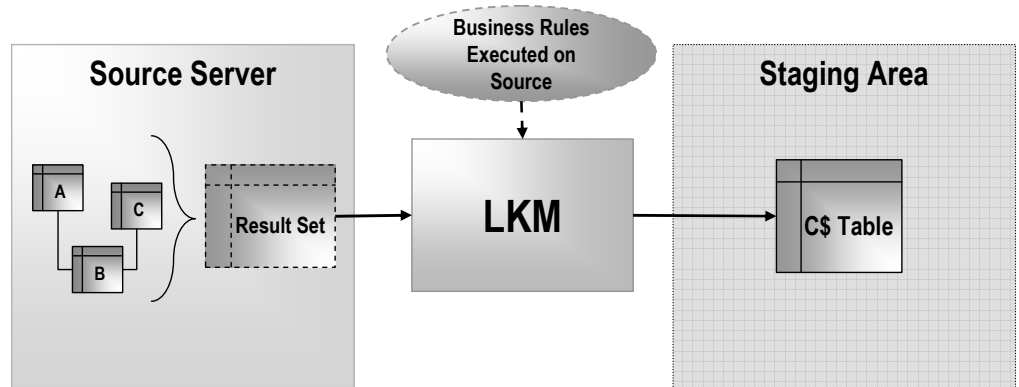


Figure 37: Loading Knowledge Module

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 in the “C\$” table of the staging area.

An interface may require several LKMs when it uses datastores from different sources. The section *Flow Diagram, Execution Plan and Staging Area* illustrates this. 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. 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

applying 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:

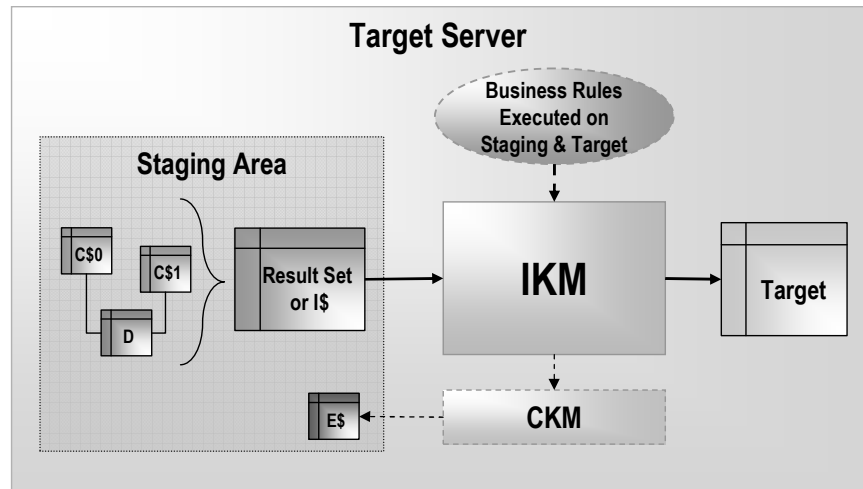


Figure 38: Integration Knowledge Module (Staging Area on Target)

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 business 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 (not the "I\$" table).

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

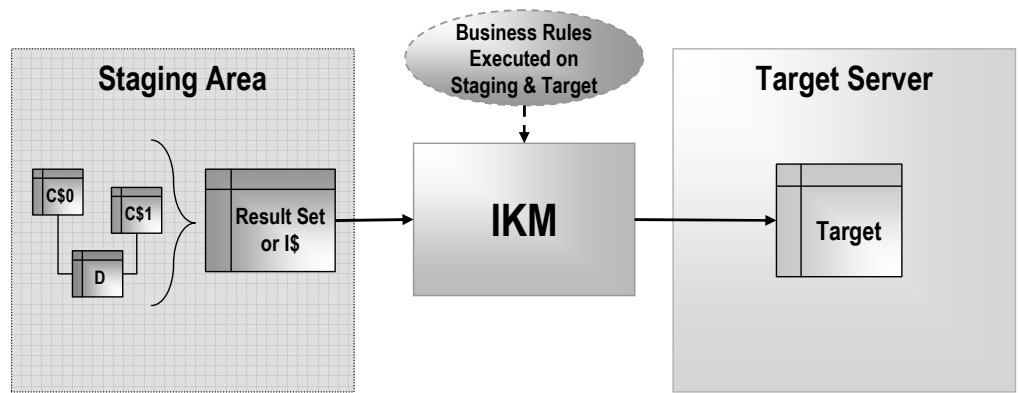


Figure 39: Integration Knowledge Module (Staging Area Different from Target)

When the staging area is different from the target server, as shown in Figure 39, the IKM usually follows these steps:

1. The IKM executes a single set-oriented SELECT statement to carry out business 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 quality 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.

Check Knowledge Modules (CKM)

The CKM is in charge of checking that records of a data set are consistent with defined constraints. The CKM can be triggered 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, regardless of any transformations. In the second case, data in the target datastore of the interface after loading the flow data is checked.
- To check consistency in the incoming flow before applying the records to a target datastore of an interface, 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 “I\$” table previously created by an IKM, which represents the records in the flow.

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 usually remove the erroneous records from the checked result set.

The following figure shows how a CKM operates.

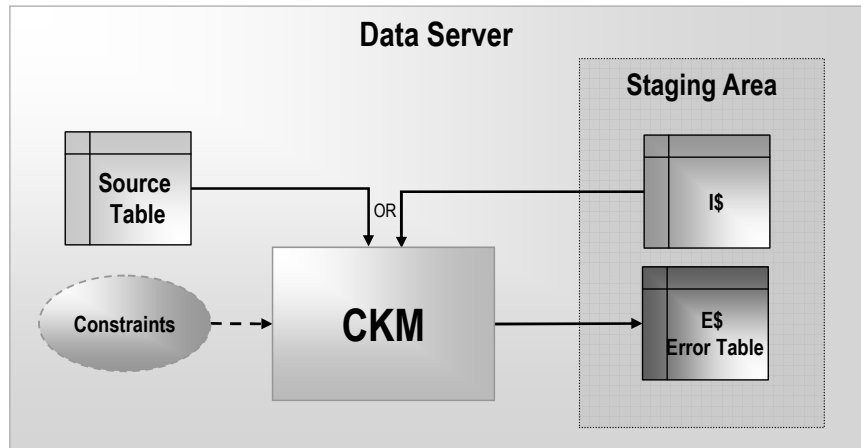


Figure 40: Check Knowledge Module

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 all 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 need to be checked.
3. If required, remove erroneous records from the table that has been checked.

Reverse-engineering Knowledge Modules (RKM)

The RKM is a type of KM not used in interfaces. Its 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 transform and write the resulting metadata into ODI'S repository, in the dedicated tables SNP_REV_xx. It then calls the ODI API to read from these tables and write to the actual ODI'S metadata tables of the work repository in incremental update mode. This is illustrated below:

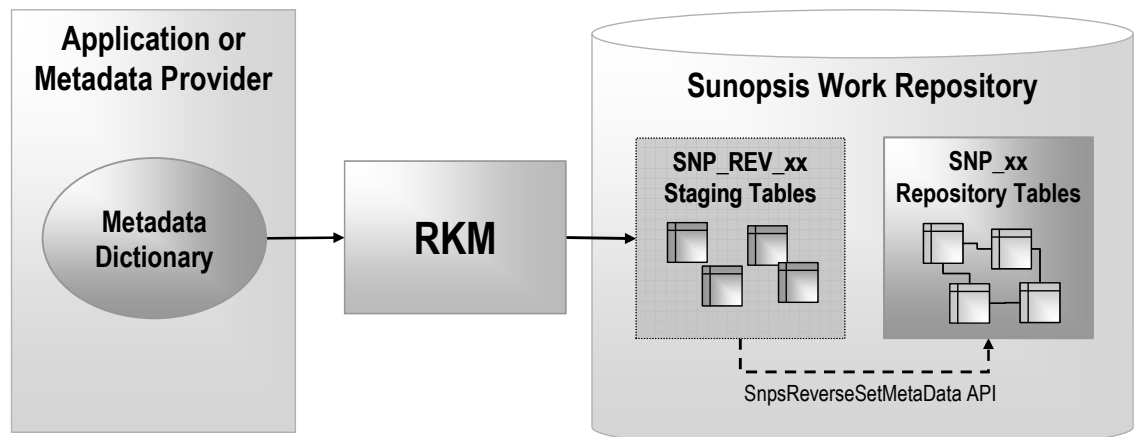


Figure 41: Reverse-engineering Knowledge Module

A typical RKM follows these steps:

1. Cleans up the SNP_REV_xx tables from previous executions using the SnpsReverseResetTable 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 SnpsReverseSetMetaData API.

Journalizing Knowledge Modules (JKM)

JKM's are in charge of creating infrastructure to set up Change Data Capture on a model, a sub model or a datastore. JKM's 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.

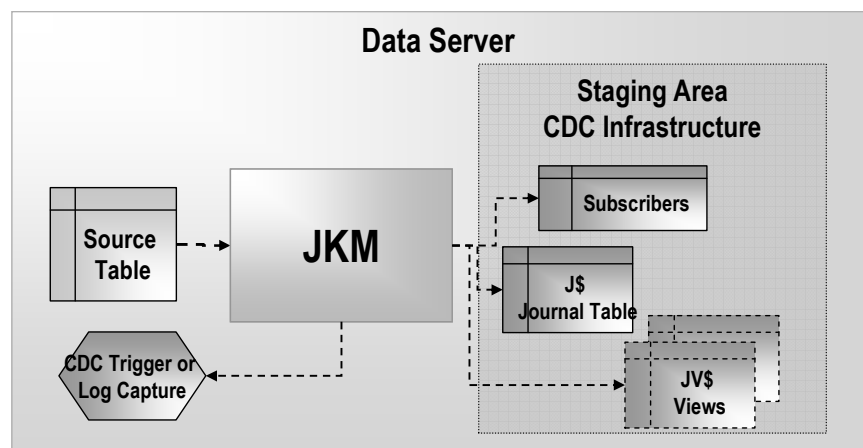


Figure 42: Journalizing Knowledge Module

Details on how JKM's work are beyond the scope of this document.

ODI Substitution API

KMs are written in a generic way by using the ODI 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 "snpRef". The same method may not return the same value depending on the type of KM that invokes it. That's why they are classified by type of KM.

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	<pre>Create table <%=snpRef.getTable("L", "INT_NAME", "A")%> (<%=snpRef.getColList("", "\t[COL_NAME] [DEST_CRE_DT]", "", "\n", "", "")%>)</pre>
Generated code for the PRODUCT datastore	<pre>Create table db_staging.I\$_PRODUCT (PRODUCT_ID numeric(10), PRODUCT_NAME varchar(250), FAMILY_ID numeric(4), SKU varchar(13), LAST_DATE timestamp)</pre>
Generated code for the CUSTOMER datastore	<pre>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))</pre>

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

The following topics cover some of the main substitution APIs and their use within KMs. Note that for better readability the "<%" and "%>" tags as well as the "snpRef" 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.

To help you achieve that, 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	<code>getObjectName("L", "MY_OBJECT", "D")</code>	All KMs and procedures
The target datastore	<code>getTable("L", "TARG_NAME", "A")</code>	LKM, CKM, IKM, JKM
The "I\$" datastore	<code>getTable("L", "INT_NAME", "A")</code>	LKM, IKM
The "C\$" datastore	<code>getTable("L", "COLL_NAME", "A")</code>	LKM
The "E\$" datastore	<code>getTable("L", "ERR_NAME", "A")</code>	LKM, CKM, IKM
The checked datastore	<code>getTable("L", "CT_NAME", "A")</code>	CKM
The datastore referenced by a foreign key	<code>getTable("L", "FK_PK_TABLE_NAME", "A")</code>	CKM

Working with Lists of Tables, Columns and Expressions

Generating code from a list of items often requires a "while" or "for" loop. ODI 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
<code>getColList()</code>	Probably the most frequently-used method in ODI. It returns	LKM, CKM,

Method	Description	Where Applicable
	<p>a list of columns and expressions that need to be executed in the context where used. You can use it for example to generate a list of:</p> <ul style="list-style-type: none"> 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 Etc. <p>This method accepts a “selector” as a 5th parameter to let you filter items as desired.</p>	IKM, JKM
<code>getTargetColList()</code>	<p>Returns the list of columns in the target datastore.</p> <p>This method accepts a selector as a 5th parameter to let you filter items as desired.</p>	LKM, CKM, IKM, JKM
<code>getAKColList()</code>	Returns the list of columns defined for an alternate key. It can only be used within a CKM step for alternate key checks.	CKM
<code>getPKColList()</code>	Returns the list of columns in a primary key. It can only be used within a CKM step for Primary Key control. You can alternatively use <code>getColList</code> with the selector parameter set to “PK” instead.	CKM
<code>getFKColList()</code>	Returns the list of referencing columns and referenced columns of the current foreign key. It can only be used within a CKM step for foreign key checks.	CKM
<code>getSrcTablesList()</code>	Returns the list of source tables of an interface. Whenever possible, use the <code>getFrom</code> method instead.	LKM, IKM
<code>getFilterList()</code>	Returns the list of filter expressions in an interface. The <code>getFilter</code> method is usually more appropriate;	LKM, IKM
<code>getJoinList()</code>	Returns the list of join expressions in an interface. The <code>getJoin</code> method is usually more appropriate.	LKM, IKM
<code>getGrpByList()</code>	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 <code>getGrpBy</code> method is usually more appropriate.	LKM, IKM

Method	Description	Where Applicable
<code>getHavingList()</code>	Returns the list of expressions that should appear in the having clause when aggregate functions are detected in the filters of an interface. The <code>getHaving</code> method is usually more appropriate.	LKM, IKM
<code>getSubscriberList()</code>	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 your KM	Create table MYTABLE <pre><%=snpRef.getTargetColList("(\\n", "\\t[COL_NAME] [DEST_WRI_DT]", ", \\n", "\\n)") %></pre>
Code Generated	Create table MYTABLE <pre>(CUST_ID numeric(10), CUST_NAME varchar(250), ADDRESS varchar(250), CITY varchar(50), ZIP_CODE varchar(12), COUNTRY_ID varchar(3))</pre>

- 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 <pre>(<%=snpRef.getColList("", "[COL_NAME]", ", ", "", "\\n", "INS AND NOT TARG") %> <%=snpRef.getColList("", "[COL_NAME]", ", ", "", "", "INS AND TARG") %>) Values (<%=snpRef.getColList("", ":[COL_NAME]", ", ", "", "\\n", "INS AND NOT TARG") %> <%=snpRef.getColList("", "[EXPRESSION]", ", ", "", "", "INS AND TARG") %>)</pre>
-----------------	---

Code Generated	<pre> 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) </pre>
----------------	--

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	<pre> Insert into MYLOGTABLE (INTERFACE_NAME, DATE_LOADED, SOURCE_TABLES) values ('<%=snpRef.getPop("POP_NAME")%>', current_date, '' <%=snpRef.getSrcTablesList(" ", "'[RES_NAME]'", " ',' ", "'')%>) </pre>
Code Generated	<pre> Insert into MYLOGTABLE (INTERFACE_NAME, DATE_LOADED, SOURCE_TABLES) values ('Int. CUSTOMER', current_date, '' 'SRC_CUST' ',' 'AGE_RANGE_FILE' ',' 'C\$0_CUSTOMER') </pre>

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. However, for the IKM, 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. ODI has some advanced substitution methods, including `getColList`, that help you generate this code:

Method	Description	Where Applicable
<code>getFrom()</code>	<p>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.</p> <ul style="list-style-type: none"> • 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
<code>getFilter()</code>	<p>Returns filter expressions separated by an “AND” operator.</p> <ul style="list-style-type: none"> • 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
<code>getJrnFilter()</code>	<p>Returns the special journal filter expressions for the journalized source datastore. This method should be used with the CDC framework.</p>	LKM, IKM
<code>getGrpBy()</code>	<p>Returns the GROUP BY clause when aggregation functions are</p>	LKM, IKM

Method	Description	Where Applicable
	<p>detected in the mappings.</p> <p>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.</p>	
getHaving()	<p>Returns the HAVING clause when aggregation functions are detected in filters.</p> <p>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.</p>	LKM, IKM

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

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

To obtain the result set from any ISO-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  <%=snpRef.getPop("DISTINCT_ROWS")%>
        <%=snpRef.getColList("", "[EXPRESSION]", "", "\n\t", "", "(not TRG) and REW")%>
from    <%=snpRef.getFrom()%>
where   (1=1)
<%=snpRef.getJoin()%>
<%=snpRef.getFilter()%>
<%=snpRef.getJrnFilter()%>
<%=snpRef.getGrpBy()%>
<%=snpRef.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
<code>getPop()</code>	Returns information about the current interface.	LKM, IKM
<code>getInfo()</code>	Returns information about the source or target server.	Any procedure or KM
<code>getSession()</code>	Returns information about the current running session	Any procedure or KM
<code>getOption()</code>	Returns the value of a particular option	Any procedure or KM
<code>getFlexFieldValue()</code>	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
<code>getJrnInfo()</code>	Returns information about the CDC framework	JKM, LKM, IKM
<code>getTargetTable()</code>	Returns information about the target table of an interface	LKM, IKM, CKM
<code>getModel()</code>	Returns information about the current model during a reverse-engineering process.	RKM

Advanced Techniques for Code Generation

Although it is not documented, you can use conditional branching and advanced programming techniques to generate code. The code generation in ODI 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
<pre><% String myTableName; myTableName = "ABCDEF"; %> drop table <%=myTableName.toLowerCase()%></pre>	<pre>drop table abcdef</pre>
<pre><% String myOptionValue=snpRef.getOption("Test"); if (myOption.equals("TRUE")) { out.print("/* Option Test is set to TRUE */"); } else { %></pre>	<pre>When option Test is set to TRUE: /* Option Test is set to TRUE */ ... Otherwise: /* The Test option is not properly set */ ...</pre>

<pre> /* The Test option is not properly set */ <% } %> ... </pre>	
<pre> Create table XYZ (<% String s; s = "ABCDEF"; for (int i=0; i < s.length(); i++) { %> <%=s.charAt(i)%> char(1), <% } %> G char(1)) </pre>	<pre> Create table XYZ (A char(1), B char(1), C char(1), D char(1), E char(1), F char(1), G char(1)) </pre>

Loading Strategies (LKM)

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 in section *Using Select on Source / Action on Target*. As discussed in that section, 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.	<pre>drop table <%=snpRef.getTable("L", "COLL_NAME", "A")%></pre>
Create the “C\$” table in the staging area	<pre> create table <%=snpRef.getTable("L", "COLL_NAME", "A")%> (<%=snpRef.getColList("", "[CX_COL_NAME]\t[DEST_WRI_DT]" + snpRef.getInfo("DEST_DDL_NULL"), ",\n\t", "", "")%>) </pre>
Load the source result set to the “C\$” table using a SELECT/INSERT command. The SELECT is executed on	<p>Code on the Source tab executed by the source server:</p> <pre> select <%=snpRef.getPop("DISTINCT_ROWS")%> <%=snpRef.getColList("", "[EXPRESSION]\t[ALIAS_SEP]" [CX_COL_NAME]", ", "\n\t", "", "")%> from <%=snpRef.getFrom()%> where (1=1) </pre>

the source and the INSERT on the staging area. The agent performs data type translations in memory using the JDBC API.	<pre> <%=snpRef.getFilter()%> <%=snpRef.getJrnFilter()%> <%=snpRef.getJoin()%> <%=snpRef.getGrpBy()%> <%=snpRef.getHaving()%> Code on the Target tab executed in the staging area: insert into <%=snpRef.getTable("L", "COLL_NAME", "A")%> (<%=snpRef.getColList("", "[CX_COL_NAME]", ",\n\t", "", "")%>) values (<%=snpRef.getColList("", ":[CX_COL_NAME]", ",\n\t", "", "")%>) </pre>
After the IKM has terminated 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.	<pre> drop table <%=snpRef.getTable("L", "COLL_NAME", "A")%> </pre>

ODI delivers the following Knowledge Modules that use this strategy:

LKM	Description
LKM SQL to SQL	Loads data through the agent from any SQL RDBMS to any SQL RDBMS staging area
LKM SQL to Oracle	Loads data through the agent from any SQL RDBMS to an Oracle staging area.
LKM SQL to DB2 UDB	Loads data through the agent from any SQL RDBMS to an IBM DB2 UDB staging area.
LKM SQL to MSSQL	Loads data through the agent from any SQL RDBMS to a Microsoft SQL Server staging area.
LKM File to SQL	Loads data through the agent from a flat file accessed using the ODI JDBC File Driver to any SQL RDBMS staging area.
LKM JMS to SQL	Loads data through the agent from any JMS message to any SQL RDBMS staging area.
LKM JMS XML to SQL	Loads data through the agent from any XML based

LKM	Description
	JMS message to any SQL RDBMS staging area.

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. Teradata suggests 3 different utilities: FastLoad for loading large files simply to append data, 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 Teradata (TPUMP-FASTLOAD-MULTILOAD)” that uses this strategy. Refer to the KM for the complete code:

Step	Example of code
Genera Fast or Mload or TPump script. This step uses the SnpsOutFile Tool to generate the script. The Java “if” statement is used to verify which type of	<pre> SnpsOutFile -File=<%=snpRef.getSrcTablesList("", "[WORK_SCHEMA]", "", "")%>/<%=snpRef.getTable("L", "COLL_NAME", "W")%>.script <% if (snpRef.getOption("TERADATA UTILITY").equals("fastload")) {%> SESSIONS <%=snpRef.getOption("SESSIONS")%> ; LOGON <%=snpRef.getInfo("DEST_DSERV_NAME")%>/<%=snpRef.getInfo("DEST_USER_NAME")%>, <%=snpRef.getInfo("DEST_PASS")%> ; BEGIN LOADING <%=snpRef.getTable("L", "COLL_NAME", "W")%> ERRORFILES <%=snpRef.getTable("L", "COLL_NAME", "W")%>_ERR1, <%=snpRef.getTable("L", "COLL_NAME", "W")%>_ERR2 </pre>

script to generate.	<pre> ; <%if (snpRef.getSrcTablesList("", "[FILE_FORMAT]", "", "").equals("F")) {%> SET RECORD TEXT; DEFINE <%=snpRef.getColList("", "\t[CX_COL_NAME] (char([LONGC]))", ",\n", "", "")%> <%} else {%> SET RECORD VARTEXT "<%=snpRef.getSrcTablesList("", "[SFILE_SEP_FIELD]", "", "")%>"; DEFINE <%=snpRef.getColList("", "\t[CX_COL_NAME] (varchar([LONGC]))", ",\n", "", "")%> <%}%> FILE=<%=snpRef.getSrcTablesList("", "[SCHEMA]/[RES_NAME]", "", "")%> ; INSERT INTO <%=snpRef.getTable("L", "COLL_NAME", "W")%> (<%=snpRef.getColList("", "\t[CX_COL_NAME]", ",\n", "", "")%>) VALUES (<%=snpRef.getColList("", "\t:[CX_COL_NAME] [COL_FORMAT]", ",\n", "", "")%>) ; END LOADING ; LOGOFF ; <%}%> <% if (snpRef.getOption("TERADATA UTILITY").equals("multiload")) {%> ... etc. ... <%}%> </pre>
Execute the load utility. This step uses Jython to execute an operating system command rather than the built-in “OS Command” technology	<pre> import os logname = "<%=snpRef.getSrcTablesList("", "[WORK_SCHEMA]", "", "")%>/<%=snpRef.getTable("L", "COLL_NAME", "W")%>.log" scriptname = "<%=snpRef.getSrcTablesList("", "[WORK_SCHEMA]", "", "")%>/<%=snpRef.getTable("L", "COLL_NAME", "W")%>.script" utility = "<%=snpRef.getOption("TERADATA UTILITY")%>" if utility == "multiload": utility="mload" loadcmd = '%s < %s > %s' % (utility,scriptname, logname) if os.system(loadcmd) <> 0 : raise "Load Error", "See %s for details" % logname </pre>

ODI delivers the following Knowledge Modules that use this strategy:

LKM	Description
LKM File to Teradata (TPUMP-FASTLOAD-MULTILOAD)	Loads a flat file to the staging area using Tump, TPT, FastLoad or MultiLoad. The utilities must be installed on the machine hosting the ODI Agent.
LKM File to DB2 UDB (LOAD)	Uses the DB2 LOAD command.
LKM File to MSSQL (BULK)	Uses Microsoft SQL Server BULK INSERT command.
LKM File to Oracle (EXTERNAL TABLE)	Uses Oracle External table command
LKM File to Oracle (SQLLDR)	Uses Oracle SQL*LOADER utility

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 more efficient than using the agent 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 SnpsSqlUnload 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 SnpsSqlUnload 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 SnpsSqlUnload	<pre> SnpsSqlUnload "--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()%> </pre>

ODI delivers the following Knowledge Modules that use this strategy:

LKM	Description
LKM SQL to Teradata (TPUMP-FASTLOAD-MULTILOAD)	Loads a remote RDBMS result set to the staging area using Unload and T pump, TPT, FastLoad or MultiLoad. These utilities must be installed on the machine hosting the ODI Agent.
LKM DB2 UDB to DB2 UDB (EXPORT_IMPORT)	Unloads and loads using IBM DB2 export and import utilities.
LKM MSSQL to MSSQL (BCP)	Unloads and loads using Microsoft Bulk Copy Program (bcp)
LKM SQL to DB2 400 (CPYFRMIMPF)	Unloads using SnpsSqlUnload from any RDBMS to a DB2/400 staging area using the CPYFRMIMPF utility of the iSeries.
LKM SQL to DB2 UDB (LOAD)	Unloads using SnpsSqlUnload from any RDBMS to a DB2/UDB staging area using the IBM LOAD utility.
LKM SQL to MSSQL (BULK)	Unloads using SnpsSqlUnload from any RDBMS to a Microsoft SQL Server staging area using the BULK INSERT command.
LKM SQL to Sybase ASE (BCP)	Unloads using SnpsSqlUnload from any RDBMS to a Sybase ASE Server staging area

LKM	Description
	using the bcp utility.
LKM Sybase ASE to Sybase ASE (BCP)	Unloads and loads using Sybase ASE bcp utility.

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 `SnpsSqlUnload` 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.

ODI 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 `SnpsSqlUnload` 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	<pre>import com.sunopsis.dwg.tools.SnpsSqlUnload as JSnpsSqlUnload import java.util.Vector as JVector</pre>

Step	Example of code
used to trigger the SnpsSqlUnload command	<pre> import java.lang.String from jararray 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 (l=1) <%=snpRef.getJoin()%> <%=snpRef.getFilter()%> <%=snpRef.getJrnFilter() %> <%=snpRef.getGrpBy()%> <%=snpRef.getHaving()%> """ ... def snpssqlunload(): snpunload = JSnpsSqlUnload() # Set the parameters cmdline = JVector() 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)) snpunload.setParameters(cmdline) # Start the unload process snpunload.execute() </pre>
Main function that runs the piped load	<pre> ... 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 </pre>

Step	Example of code
	<pre> 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") 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", "(%) load process not started" % loadcmd # Start the SQLUnload process try: snpssqlunload() 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() </pre>

The following standard Knowledge Modules use this strategy:

LKM	Description
LKM SQL to Teradata (pipelined PUMP-FAST-MULTILOAD)	Loads a remote RDBMS result set to the staging area using a pipelined Unload and Tpump, TPT, FastLoad or MultiLoad. The utilities must be installed on the Unix machine hosting the ODI Agent.

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
- Etc.

Have a look at the following LKMs, which make use of these features:

LKM	Description
LKM DB2 400 to DB2 400	Uses the DRDA mechanism for sharing data across iSeries
LKM Informix to Informix (SAME SERVER)	Move data across Informix database inside the same server.
MSSQL to MSSQL (LINKED SERVERS)	Uses Microsoft SQL Server's linked servers technology
Oracle to Oracle (DBLINK)	Uses Oracle's data base links technology

Integration Strategies (IKM)

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 server.

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

This very simple strategy is not provided as is in the default ODI 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	<pre>create table <%=snpRef.getTable("L", "INT_NAME", "A")%> (<%=snpRef.getColList("", "[COL_NAME]\t[DEST_WRI_DT] " + snpRef.getInfo("DEST_DDL_NULL"), ",\n\t", "", "INS")%>)</pre>	FLOW_CONTR OL is set to YES
Insert flow data	<pre>insert into <%=snpRef.getTable("L", "INT_NAME", "A")%> (</pre>	FLOW_CONTR

Step	Example of code	Execute if
in the “I\$” table	<pre> <%=snpRef.getColList("", "[COL_NAME]", "", "\n\t", "", "((INS and !TRG) and REW)")%>) select <%=snpRef.getPop("DISTINCT_ROWS")%> <%=snpRef.getColList("", "[EXPRESSION]", "", "\n\t", "", "((INS and !TRG) and REW)")%> from <%=snpRef.getFrom()%> where (1=1) <%=snpRef.getJoin()%> <%=snpRef.getFilter()%> <%=snpRef.getJrnFilter()%> <%=snpRef.getGrpBy()%> <%=snpRef.getHaving()%> </pre>	OL is set to YES
Recycle previous rejected records	<pre> insert into <%=snpRef.getTable("L", "INT_NAME", "A")%> (<%=snpRef.getColList("", "[COL_NAME]", "", "\n\t", "", "INS and REW")%>) select <%=snpRef.getColList("", "[COL_NAME]", "", "\n\t", "", "INS and REW")%> from <%=snpRef.getTable("L", "ERR_NAME", "A")%> <%=snpRef.getInfo("DEST_TAB_ALIAS_WORD")%> E where not exists (select 'X' from <%=snpRef.getTable("L", "INT_NAME", "A")%> <%=snpRef.getInfo("DEST_TAB_ALIAS_WORD")%> T where <%=snpRef.getColList("", "T.[COL_NAME]\t= E.[COL_NAME]", "\n\t\tand\t", "", "UK")%>) and E.ORIGIN = '<%=snpRef.getInfo("CT_ORIGIN")%>' and E.ERR_TYPE = '<%=snpRef.getInfo("CT_ERR_TYPE")%>' </pre>	RECYCLE_ERR ORS is set to Yes
Call the CKM to perform data quality check	<%@ INCLUDE CKM_FLOW DELETE_ERRORS%>	FLOW_CONTR OL is set to YES
Remove all records from the target table	delete from <%=snpRef.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.	<pre> <%if (snpRef.getOption("FLOW_CONTROL").equals("1")) { %> insert into <%=snpRef.getTable("L", "TARG_NAME", "A")%> (<%=snpRef.getColList("", "[COL_NAME]", "", "\n\t", "", "((INS and !TRG) and REW)")%> <%=snpRef.getColList("", "[COL_NAME]", "", "\n\t", "", "((INS and TRG) and REW)")%>) select <%=snpRef.getColList("", "[COL_NAME]", "", "\n\t", "", "((INS and !TRG) and REW)")%> <%=snpRef.getColList("", "[EXPRESSION]", "", "\n\t", "", "((INS and TRG) and REW)")%> from <%=snpRef.getTable("L", "INT_NAME", "A")%> <% } else { %> insert into <%=snpRef.getTable("L", "TARG_NAME", "A")%> </pre>	INSERT is set to Yes

Step	Example of code	Execute if
	<pre> (<%=snpRef.getColList("", "[COL_NAME]", "", "\n\t", "", "(INS and REW)")%>) select <%=snpRef.getPop("DISTINCT_ROWS")%> <%=snpRef.getColList("", "[EXPRESSION]", "", "\n\t", "", "(INS and REW)")%> from <%=snpRef.getFrom()%> where <% if (snpRef.getPop("HAS_JRN").equals("0")) { %> (1=1) <%} else {%> JRN_FLAG <> 'D' <% } %> <%=snpRef.getJoin()%> <%=snpRef.getFilter()%> <%=snpRef.getJrnFilter()%> <%=snpRef.getGrpBy()%> <%=snpRef.getHaving()%> <% } %> </pre>	

ODI delivers the following Knowledge Modules that use this strategy:

IKM	Description
IKM SQL Control Append	Integrates data in any ISO-92 compliant database target table in TRUNCATE/INSERT (append mode.) Data quality can be checked. Invalid data is rejected in the “E\$” error table and can be recycled.

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 can be used as an update key, except when it is automatically calculated using an increment such as an identity column, a rank function, or a sequence.

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	<pre> create <%=snpRef.getTable("L", "INT_NAME", "A")%> table <%=snpRef.getTable("L", "INT_NAME", "A")%>, <%=snpRef.getTable("L", "INT_NAME", "A")%> (<%=snpRef.getColList("", " [COL_NAME]\t[DEST_WRI_DT] " + snpRef.getInfo("DEST_DDL_NULL"), " ,\n\t", "", "")%>, IND_UPDATE char(1)) primary index (<%=snpRef.getColList("", " [COL_NAME]", " ,", " ", "UK")%>) </pre>	
Determine what to update (using the update key)	<pre> update <%=snpRef.getTable("L", "INT_NAME", "A")%> from <%=snpRef.getTable("L", "TARG_NAME", "A")%> T set IND_UPDATE = 'U' where <%=snpRef.getColList("", snpRef.getTable("L", "INT_NAME", "A") + ". [COL_NAME]\t= T. [COL_NAME]", "\nand\t", "", "UK")%> </pre>	INSERT or UPDATE are set to Yes
Determine what shouldn't be updated by comparing the data	<pre> update <%=snpRef.getTable("L", "INT_NAME", "A")%> from <%=snpRef.getTable("L", "TARG_NAME", "A")%> T set IND_UPDATE = 'N' where <%=snpRef.getColList("", snpRef.getTable("L", "INT_NAME", "A") + ". [COL_NAME]\t= T. [COL_NAME]", "\nand\t", "", "UK")%> and <%=snpRef.getColList("", "(" + snpRef.getTable("L", "INT_NAME", "A") + ". [COL_NAME] = T. [COL_NAME]) or (" + snpRef.getTable("L", "INT_NAME", "A") + ". [COL_NAME] IS NULL and T. [COL_NAME] IS NULL))", " \nand\t", "", "((UPD and !TRG) and !UK) ")%> </pre>	UPDATE is set to Yes
Update the target with the existing records	<pre> update <%=snpRef.getTable("L", "TARG_NAME", "A")%> from <%=snpRef.getTable("L", "INT_NAME", "A")%> S set <%=snpRef.getColList("", " [COL_NAME]\t= S. [COL_NAME]", " ,\n\t", "", "((UPD and !UK) and !TRG) and REW) "%> <%=snpRef.getColList("", " [COL_NAME]=[EXPRESSION]", " ,\n\t", "", "((UPD and !UK) and TRG) and REW) "%> where <%=snpRef.getColList("", snpRef.getTable("L", "TARG_NAME", "A") + ". [COL_NAME]\t= S. [COL_NAME]", "\nand\t", "", " (UK) "%> and S.IND_UPDATE = 'U' </pre>	UPDATE is set to Yes
Insert new records	<pre> insert into <%=snpRef.getTable("L", "TARG_NAME", "A")%> (<%=snpRef.getColList("", " [COL_NAME]", " ,\n\t", "", "((INS and !TRG) and REW) "%> <%=snpRef.getColList("", " [COL_NAME]", " ,\n\t", "", "((INS and TRG) and REW) "%>) select <%=snpRef.getColList("", " [COL_NAME]", " ,\n\t", "", "((INS and !TRG) and REW) "%> <%=snpRef.getColList("", " [EXPRESSION]", " ,\n\t", "", "((INS and TRG) and REW) "%> from <%=snpRef.getTable("L", "INT_NAME", "A")%> where IND_UPDATE = 'I' </pre>	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 would have been written this way:

```
<%=snpRef.getColList("", "coalesce(" + snpRef.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 applied to the target.

It is recommended that you have a look at the following IKMs as they all may implement a similar strategy:

IKM	Description
IKM Teradata Incremental Update	Set-based incremental update for Teradata
IKM DB2 400 Incremental Update	Set-based incremental update for DB2/400
IKM DB2 400 Incremental Update (CPYF)	Incremental update for DB2/400 using the CPYF utility to compare the records
IKM DB2 UDB Incremental Update	Set-based incremental update for DB2/UDB
IKM Informix Incremental Update	Set-based incremental update for Informix
IKM MSSQL Incremental Update	Set-based incremental update for Microsoft SQL Server
IKM Oracle Incremental Update (MERGE)	Set-based incremental update for Oracle using the MERGE command introduced in Oracle 9i
IKM Oracle Incremental Update (PL SQL)	Cursor-based incremental update for Oracle using PL/SQL

IKM	Description
IKM Oracle Incremental Update	Set-based incremental update for Oracle
IKM SQL Incremental Update	Row-by-row-based incremental update for any SQL compliant database. The flow is fetched by the Agent and inserted or updated row by row.
IKM Sybase ASE Incremental Update	Set-based incremental update for Sybase Adaptive Server
IKM Access Incremental Update	Set-based incremental update for Microsoft Access

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 following 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.

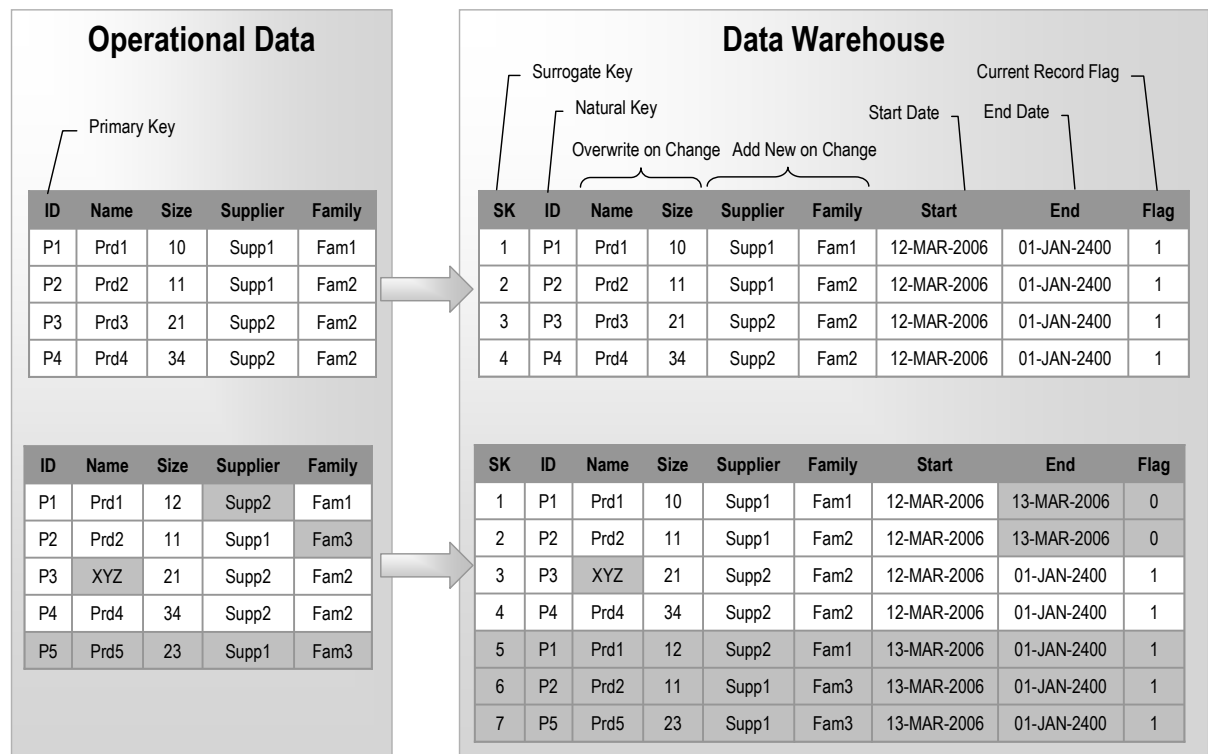


Figure 43: Slowly Changing Dimension Example

1. 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:
2. The supplier is updated for product P1
3. The family is updated for product P2
4. The name is updated for product P3
5. 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. ODI can set

this information in additional metadata fields for every column of the target slowly changing dimension datastore as described in the figure below.

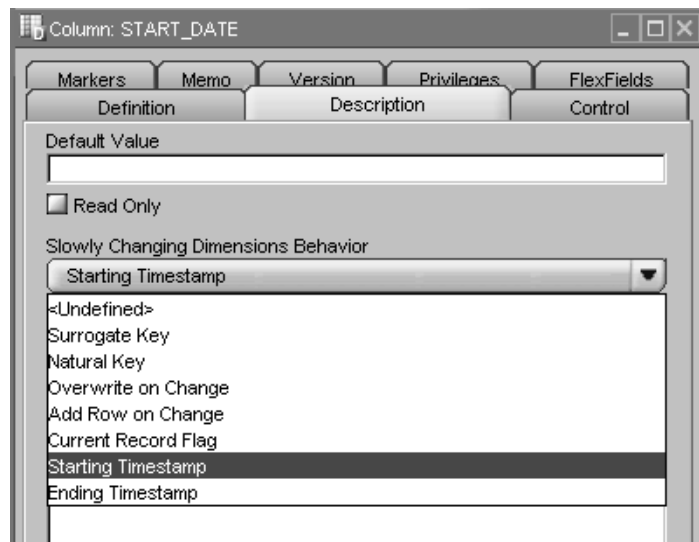


Figure 44: Slowly Changing Dimension Column Behavior

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 ODI 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 EXCEPT statement	<pre> insert into <%=snpRef.getTable("L","INT_NAME","A")%> (<%=snpRef.getColList("", "[COL_NAME]", "",\n\t", "", "(((SCD_NK or SCD_INS or SCD_UPD) and !TRG) and REW)")%> <%=snpRef.getColList("", "[COL_NAME]", "",\n\t", "", "(SCD_START and REW)")%> <%=snpRef.getColList("", "[COL_NAME]", "",\n\t", "", "(SCD_END and REW)")%> IND_UPDATE) select <%=snpRef.getPop("DISTINCT_ROWS")%> <%=snpRef.getColList("", "[EXPRESSION]", "",\n\t", "", "(((SCD_NK or SCD_INS or SCD_UPD) and !TRG) and REW)")%> <%=snpRef.getColList("", "[EXPRESSION]", "",\n\t", "", "(SCD_START and REW)")%> <%=snpRef.getColList("", "cast('2400-01-01' as [DEST_WRI_DT])", ",\n\t", "", "(SCD_END and REW)")%> <%if (snpRef.getPop("HAS_JRN").equals("0")) {%> 'I' <%}else{%> JRN_FLAG <%}%> from <%=snpRef.getFrom()%> where (1=1) <%=snpRef.getJoin()%> <%=snpRef.getFilter()%> <%=snpRef.getJrnFilter()%> <%=snpRef.getGrpBy()%> <%=snpRef.getHaving()%> except select <%=snpRef.getColList("", "[COL_NAME]", "",\n\t", "", "(((SCD_NK or SCD_INS or SCD_UPD) and !TRG) and REW)")%> <%=snpRef.getColList("", "[COL_NAME]", "",\n\t", "", "(SCD_START and REW)")%> <%=snpRef.getColList("", "[COL_NAME]", "",\n\t", "", "(SCD_END and REW)")%> 'I' from <%=snpRef.getTable("L","TARG_NAME","A")%> where <%=snpRef.getColList("", "[COL_NAME]\t= 1", "\nand\t", "", "SCD_FLAG")%> </pre>
Flag records that require an update on the target	<pre> update S from <%=snpRef.getTable("L", "INT_NAME", "A")%> as S, <%=snpRef.getTable("L", "TARG_NAME", "A")%> as T set IND_UPDATE = 'U' where <%=snpRef.getColList("", "S.[COL_NAME]\t= T.[COL_NAME]", "", "", "SCD_NK")%> and <%=snpRef.getColList("", "((S.[COL_NAME] = T.[COL_NAME]) or ((S.[COL_NAME] is null) and (T.[COL_NAME] is null)))", "\nand\t", "", "SCD_INS and !TRG")%> and <%=snpRef.getColList("", "T.[COL_NAME]\t= 1", "\n\t\tand\t", </pre>

Step	Example of code
	<pre> ", "SCD_FLAG")%> </pre>
Update the updatable columns on the target	<pre> update T from <%=snpRef.getTable("L", "TARG_NAME", "A")%> as T, <%=snpRef.getTable("L", "INT_NAME", "A")%> as S set <%=snpRef.getColList("", "[COL_NAME]\t= S.[COL_NAME]", "\n\t", ",", "(((SCD_UPD) and !TRG) and REW)")%> <%=snpRef.getColList("", "[COL_NAME]\t= [EXPRESSION]", ",\n\t", "", "(((SCD_UPD) and TRG) and REW)")%> where <%=snpRef.getColList("", "T.[COL_NAME]\t= S.[COL_NAME]", "\nand\t", "", "SCD_NK")%> and <%=snpRef.getColList("", "T.[COL_NAME]\t= 1", "\nand\t", "", "SCD_FLAG")%> and S.IND_UPDATE = 'U' </pre>
Close obsolete records	<pre> update T from <%=snpRef.getTable("L", "TARG_NAME", "A")%> as T, <%=snpRef.getTable("L", "INT_NAME", "A")%> as S set <%=snpRef.getColList("", "[COL_NAME]\t= 0", "\n\t", "", "(SCD_FLAG and REW)")%> <%=snpRef.getColList("", "[COL_NAME]\t= ", "", "", "(SCD_END and REW)")%><%=snpRef.getColList("", "S.[COL_NAME]", "\n\t", "", "(SCD_START and REW)")%> where <%=snpRef.getColList("", "T.[COL_NAME]\t= S.[COL_NAME]", "\nand\t", "", "SCD_NK")%> and <%=snpRef.getColList("", "T.[COL_NAME]\t= 1", "\nand\t", "", "SCD_FLAG")%> and S.IND_UPDATE = 'I' </pre>
Insert new records	<pre> insert into <%=snpRef.getTable("L", "TARG_NAME", "A")%> (<%=snpRef.getColList("", "[EXPRESSION]", "", "", "(((SCD_SK and TRG) and REW)").equals("0"))){out.print(snpRef.getColList("", "[COL_NAME]", "", "", "(((SCD_SK and TRG) and REW))");}%> <%=snpRef.getColList("", "[COL_NAME]", "\n\t", "", "(((SCD_NK or SCD_INS or SCD_UPD) and !TRG) and REW)")%> <%=snpRef.getColList("", "[COL_NAME]", "\n\t", "", "(((SCD_NK or SCD_INS or SCD_UPD) and TRG) and REW)")%> <%=snpRef.getColList("", "[COL_NAME]", "\n\t", "", "(SCD_FLAG and REW)")%> <%=snpRef.getColList("", "[COL_NAME]", "\n\t", "", "(SCD_START and REW)")%> <%=snpRef.getColList("", "[COL_NAME]", "\n\t", "", "(SCD_END and REW)")%>) select <%=snpRef.getColList("", "[EXPRESSION]", "", "", "(((SCD_SK and TRG) and REW)").equals("0"))){out.print(snpRef.getColList("", "\t[EXPRESSION]", "", "", "(((SCD_SK and TRG) and REW))");}%> <%=snpRef.getColList("", "[COL_NAME]", "\n\t", "", "(((SCD_NK or SCD_INS or SCD_UPD) and !TRG) and REW)")%> <%=snpRef.getColList("", "[EXPRESSION]", "\n\t", "", "(((SCD_NK or SCD_INS or SCD_UPD) and TRG) and REW)")%> <%=snpRef.getColList("", "1", "\n\t", "", "(SCD_FLAG AND REW)")%> <%=snpRef.getColList("", "[COL_NAME]", "\n\t", "", "(SCD_START and REW)")%> <%=snpRef.getColList("", "[COL_NAME]", "\n\t", "", "(SCD_END and REW)")%> from <%=snpRef.getTable("L", "INT_NAME", "A")%> as S where S.IND_UPDATE = 'I' </pre>

See also the following IKMs which may implement the slowly changing dimension strategy:

IKM	Description
IKM Teradata Slowly Changing Dimension	Slowly Changing Dimension for Teradata
IKM DB2 400 Slowly Changing Dimension	Slowly Changing Dimension for DB2/400
IKM DB2 UDB Slowly Changing Dimension	Slowly Changing Dimension for DB2 UDB
IKM MSSQL Slowly Changing Dimension	Slowly Changing Dimension for Microsoft SQL Server
IKM Oracle Slowly Changing Dimension	Slowly Changing Dimension for Oracle
IKM Sybase ASE Slowly Changing Dimension	Slowly Changing Dimension for Sybase Adaptive Server

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!

Another, 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 “T”
- 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	<code>Drop table <%=snpRef.getTable("L", "TARG_NAME", "A")%>_BCK</code>
Create the backup table	<code>Create table <%=snpRef.getTable("L", "TARG_NAME", "A")%>_BCK (<%=snpRef.getTargetColList("", "\t[COL_NAME] [DEST_CRE_DT]", "", \n", "")%>) primary index (<%=snpRef.getTargetColList("", "[COL_NAME]", "", "", "", "PK")%>)</code>
Backup the target data	<code>insert into <%=snpRef.getTable("L", "TARG_NAME", "A")%>_BCK (<%=snpRef.getTargetColList("", "[COL_NAME]", "", "", "")%>) select <%=snpRef.getTargetColList("", "[COL_NAME]", "", "", "")%> from <%=snpRef.getTable("L", "TARG_NAME", "A")%></code>

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

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, ODI 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 shouldn’t need to have the file loaded in the staging “C\$” table before being applied 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 would imply defining a staging area different from the target in your interfaces and setting it to the file logical schema. By doing so, ODI 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

The following table shows some excerpts from the IKM File to Teradata (TPUMP-FAST-MULTILOAD), which allows the generation of appropriate scripts for each of these Teradata utilities depending on your integration strategy:

Step	Example of code
Generate Fastload, MultiLoad or Tpump script	<pre>SnpsOutFile -File=<%=snpRef.getSrcTablesList("", "[WORK_SCHEMA]", "", "")%>/<%=snpRef.getTargetTable("SCHEMA")%>.<%=snpRef.getTargetTable("RES_ _NAME")%>.script <% if (snpRef.getOption("TERADATA UTILITY").equals("fastload")) {%> SESSIONS <%=snpRef.getOption("SESSIONS")%> ; LOGON <%=snpRef.getInfo("DEST_DSERV_NAME")%>/<%=snpRef.getInfo("DEST_USER_NAME ")%>,<%=snpRef.getInfo("DEST_PASS")%> ; BEGIN LOADING <%=snpRef.getTargetTable("SCHEMA")%>.<%=snpRef.getTargetTable("RES_NAME")%> ERRORFILES <%=snpRef.getTargetTable("WORK_SCHEMA")%>.<%=snpRef.getTargetTable("RES_ NAME")%>_ERR1, <%=snpRef.getTargetTable("WORK_SCHEMA")%>.<%=snpRef.getTargetTable("RES_ NAME")%>_ERR2 ; </pre>

Step	Example of code
	<pre> <%if (snpRef.getSrcTablesList("", "[FILE_FORMAT]", "", "").equals("F")) {%> SET RECORD TEXT; DEFINE <%=snpRef.getColList("", "\t[CX_COL_NAME] (char([LONGC]))", ",\n", "","")%> <%} else {%> SET RECORD VARTEXT "<%=snpRef.getSrcTablesList("", "[SFILE_SEP_FIELD]", "","")%>"; DEFINE <%=snpRef.getColList("", "\t[CX_COL_NAME] (varchar([LONGC]))", ",\n", "","")%> <%}%> FILE=<%=snpRef.getSrcTablesList("", "[SCHEMA]/[RES_NAME]", "", "")%> ; INSERT INTO <%=snpRef.getTargetTable("SCHEMA")%>.<%=snpRef.getTargetTable("RES_NAME")%> (<%=snpRef.getColList("", "\t[CX_COL_NAME]", ",\n", "", "")%>) VALUES (<%=snpRef.getColList("", "\t:[CX_COL_NAME] [COL_FORMAT]", ",\n", "","")%>) ; END LOADING ; LOGOFF ; <%}%> <% if (snpRef.getOption("TERADATA UTILITY").equals("multiload")) {%> [etc.] <%}%> </pre>
Start the load using a Jython script	<pre> import os logname = "<%=snpRef.getSrcTablesList("", "[WORK_SCHEMA]", "", "")%>/<%=snpRef.getTargetTable("SCHEMA")%>.<%=snpRef.getTargetTable("RES _NAME")%>.log" scriptname = "<%=snpRef.getSrcTablesList("", "[WORK_SCHEMA]", "", "")%>/<%=snpRef.getTargetTable("SCHEMA")%>.<%=snpRef.getTargetTable("RES _NAME")%>.script" utility = "<%=snpRef.getOption("TERADATA UTILITY")%>" if utility == "multiload": utility="mload" loadcmd = '%s < %s > %s' % (utility,scriptname, logname) if os.system(loadcmd) <> 0 : raise "Load Error", "See %s for details" % logname </pre>

This type of Knowledge Module has the following restrictions:

- Your interface contains a single file as source
- The staging area is set to your file logical schema
- The data from the file is not transformed (simple one-to-one mappings)
- The flow can't be checked against data quality rules

The following IKM implements this strategy:

IKM	Description
IKM File to Teradata (TPUMP-FAST-MULTILOAD)	Integrates data from a File to a Teradata table without staging the file data. The Utility used for loading can be either Fastload, TPT, Multiload or Tpump.

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 following IKM implements this strategy:

IKM	Description
IKM SQL to SQL Append	Integrates data into an ANSI-SQL92 target database from any remote ANSI-SQL92 compliant staging area in replace or append mode.

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

The following IKMs implement this strategy:

IKM	Description
IKM SQL to Teradata (pipelined TPUMP-FAST-MULTILOAD)	Integration from any SQL compliant staging area to Teradata using either FASTLOAD, TPT, MULTILOAD or TPUMP. Data is extracted using SnpsSqlUnload into a Unix FIFO and loaded using the appropriate utility.
IKM SQL to Teradata (TPUMP-FAST-MULTILOAD)	Integration from any SQL compliant staging area to Teradata using either FASTLOAD, TPT, MULTILOAD or TPUMP. Data is extracted using SnpsSqlUnload into a temporary file and loaded using the appropriate utility.

Server to File or JMS Append

When the target datastore is a file or JMS queue or topic you will obviously 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 specific “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 ODI 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

You can refer to the following IKMs for further details:

IKM	Description
IKM Teradata to File (FASTEXPORT)	Exports data from a Teradata staging area to an ASCII file using FastExport
IKM SQL to File Append	Exports data from any SQL compliant staging area to an ASCII or BINARY file using the

IKM	Description
	ODI File driver
IKM SQL to JMS Append	Exports data from any SQL compliant staging area and posts ASCII or BINARY messages into JMS queue or topic (such as IBM Websphere MQ using the Sunopsis JMS driver
IKM SQL to JMS XML Append	Exports data from any SQL compliant staging area and posts XML messages to a JMS queue or topic (such as IBM Websphere MQ using the Sunopsis JMS driver

Data Quality 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

Table	Column	Description
	FULL_RES_NAME	Fully qualified name of the checked table. For example <catalog>.<schema>.<table>
	ERR_TYPE	Type of error: <ul style="list-style-type: none"> • '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: <ul style="list-style-type: none"> • '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
"E\$" Error table	[Columns of the checked table]*	The error table contains all the columns of the checked datastore.
	ERR_TYPE	Type of error: <ul style="list-style-type: none"> • 'F' when the datastore is checked during flow control • 'S' when the datastore is checked using static control

Table	Column	Description
	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: <ul style="list-style-type: none"> • '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, ODI will use this command for every alternate key
- “Join”: The command defines the code needed to check a foreign key constraint. During code generation, ODI will use this command for every foreign key
- “Condition”: The command defines the code needed to check a condition constraint. During code generation, ODI will use this command for every check condition
- “Mandatory”: The command defines the code needed to check a mandatory column constraint. During code generation, ODI 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 Teradata are provided below:

Step	Example of code	Conditions to Execute
Create the summary table	<pre>create multiset table <%=snpRef.getTable("L", "CHECK_NAME", "W")%>, no fallback, no before journal, no after journal (CATALOG_NAME varchar(100) , SCHEMA_NAME varchar(100) , RESOURCE_NAME varchar(100) , FULL_RES_NAME varchar(100) , ERR_TYPE varchar(1) , ERR_MESS varchar(250) , CHECK_DATE timestamp , ORIGIN varchar(100) , CONS_NAME varchar(35) , CONS_TYPE varchar(2) , ERR_COUNT numeric(10))</pre>	Always. Error Tolerated
Create the error	<pre>create multiset table <%=snpRef.getTable("L", "ERR_NAME", "A")%>, </pre>	Always. Error

Step	Example of code	Conditions to Execute
table	<pre> no fallback, no before journal, no after journal (ERR_TYPE varchar(1) , ERR_MESS varchar(250) , CHECK_DATE timestamp , <%=snpRef.getColList("", "[COL_NAME]\t[DEST_WRI_DT]" + snpRef.getInfo("DEST_DDL_NULL"), "\n\t", "", "")%>, ORIGIN varchar(100) , CONS_NAME varchar(35) , CONS_TYPE varchar(2) ,) </pre>	Tolerated
Isolate PK errors	<pre> insert into <%=snpRef.getTable("L","ERR_NAME", "A")%> (ERR_TYPE, ERR_MESS, ORIGIN, CHECK_DATE, CONS_NAME, CONS_TYPE, <%=snpRef.getColList("", "[COL_NAME]", "\n\t", "", "MAP")%>) select '<%=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")%> as <%=snpRef.getTargetTable("TABLE_ALIAS")%> where (<%=snpRef.getColList("", snpRef.getTargetTable("TABLE_ALIAS") + ". [COL_NAME]", "\n\t\t", "", "PK")%>) in (select <%=snpRef.getColList("", "[COL_NAME]", "\n\t\t\t", "", "PK")%> from <%=snpRef.getTable("L", "CT_NAME", "A")%> as E group by <%=snpRef.getColList("", "E. [COL_NAME]", "\n\t\t\t", "", "PK")%> having count(*) > 1) <%=snpRef.getFilter()%> </pre>	Primary Key
Remove errors from checked table	<pre> delete from <%=snpRef.getTable("L", "CT_NAME", "A")%> where exists (select 'X' from </pre>	Remove Errors

Step	Example of code	Conditions to Execute
	<pre> <%=snpRef.getTable("L", "ERR_NAME", "A")%> as E <%if (snpRef.getInfo("CT_ERR_TYPE").equals("F")) {%> where <%=snpRef.getColList("", "(" +snpRef.getTable("L", "CT_NAME", "A")+ ".[COL_NAME]\t= E.[COL_NAME]) or (" +snpRef.getTable("L", "CT_NAME", "A")+ ".[COL_NAME] is null and E.[COL_NAME] is null))", "\n\t\tand\t", "", "UK")%> <%}else{%> where <%=snpRef.getColList("", "(" +snpRef.getTable("L", "CT_NAME", "A")+ ".[COL_NAME]\t= E.[COL_NAME]) or (" +snpRef.getTable("L", "CT_NAME", "A")+ ".[COL_NAME] is null and E.[COL_NAME] is null))", "\n\t\tand\t", "", "PK")%> <%}%> and ERR_TYPE = '<%=snpRef.getInfo("CT_ERR_TYPE")%>' and ORIGIN = '<%=snpRef.getInfo("CT_ORIGIN")%>') </pre>	

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”.

Refer to the following CKMs for further details:

CKM	Description
CKM Teradata	Check Knowledge Module for Teradata
CKM SQL	Check Knowledge Module for any SQL compliant database
CKM Oracle	Check Knowledge Module for Oracle using rowids
CKM HSQL	Check Knowledge Module for Hypersonic SQL

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

In some cases when loading the data warehouse, you may receive records that should reference data from other tables, but for specific reasons, 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 specific 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.

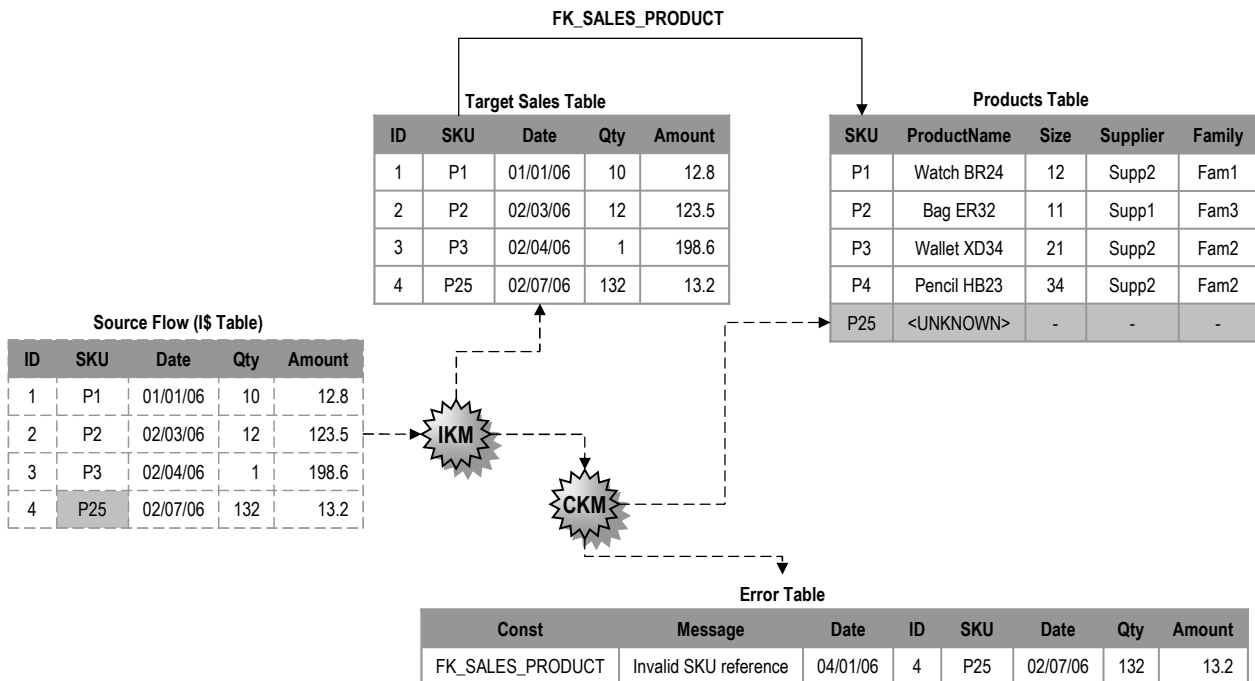


Figure 45: Creating References on the Fly

- 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 ODI 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 ODI 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 to section *Adding User-Defined Metadata with Flex Fields*.

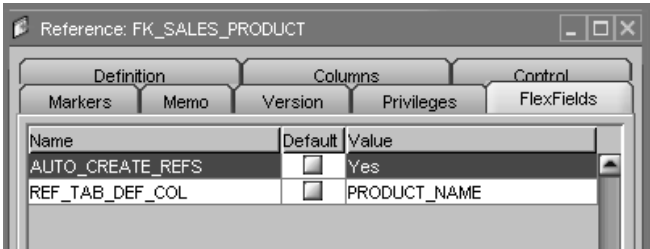


Figure 46: Adding Flex Fields to the FK_SALES_PRODUCT Foreign Key

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 of 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
Create the error	<code>create multiset table <%=snpRef.getTable("L", "ERR_NAME", "A")%>,</code>

Step	Example of code
table	<pre> no fallback, no before journal, no after journal (AUTO_CREATE_REFS varchar(3), ERR_TYPE varchar(1) , ERR_MESS varchar(250) , CHECK_DATE timestamp , <%=snpRef.getColList("", "[COL_NAME]\t[DEST_WRI_DT]" + snpRef.getInfo("DEST_DDL_NULL"), ",\n\t", "", "")%>, ORIGIN varchar(100) , CONS_NAME varchar(35) , CONS_TYPE varchar(2) ,) </pre>
Isolate FK errors	<pre> insert into <%=snpRef.getTable("L", "ERR_NAME", "A")%> (AUTO_CREATE_REFS, ERR_TYPE, ERR_MESS, CHECK_DATE, ORIGIN, CONS_NAME, CONS_TYPE, <%=snpRef.getColList("", "[COL_NAME]", ",\n\t", "", "MAP")%>) select '<%=snpRef.getFK("AUTO_CREATE_REFS")%>', '<%=snpRef.getInfo("CT_ERR_TYPE")%>', '<%=snpRef.getFK("MESS")%>', '<%=snpRef.getInfo("DEST_DATE_FCT")%>', '<%=snpRef.getInfo("CT_ORIGIN")%>', '<%=snpRef.getFK("FK_NAME")%>', 'FK', [... etc.] </pre>
Insert missing references	<pre> <% if (snpRef.getFK("AUTO_CREATE_REFS").equals("Yes")) { %> insert into <%=snpRef.getTable("L", "FK_PK_TABLE_NAME", "A")%> (<%=snpRef.getFKColList("", "[PK_COL_NAME]", "", "", "")%> , <%=snpRef.getFK("REF_TAB_DEF_COL")%>) select distinct <%=snpRef.getFKColList("", "[COL_NAME]", "", "", "")%> , '<UNKNOWN>' from <%=snpRef.getTable("L", "ERR_NAME", "A")%> where CONS_NAME = '<%=snpRef.getFK("FK_NAME")%>' And CONS_TYPE = 'FK' And ORIGIN = '<%=snpRef.getInfo("CT_ORIGIN")%>' And AUTO_CREATE_REFS = 'Yes' <%}%> </pre>
Remove the rejected records from the checked	<pre> delete from <%=snpRef.getTable("L", "CT_NAME", "A")%> where exists (select 'X' from <%=snpRef.getTable("L", "ERR_NAME", "A")%> as E where </pre>

Step	Example of code
table	<pre> <%=snpRef.getColList("", "(" + snpRef.getTable("L", "CT_NAME", "A") + ".[COL_NAME]\t= E.[COL_NAME]) or (" + snpRef.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.] </pre>

Reverse-engineering Knowledge Modules (RKM)

RKM Process

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

1. Call the SnpReverseResetTable 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 SnpsReverseSetMetaData command to apply the changes to the current ODI model.

As an example, the steps below are extracted from the RKM for Teradata:

Step	Example of code
Reset SNP_REV tables	<pre> SnpsReverseResetTable -MODEL=<%=snpRef.getModel("ID")%> </pre>
Get Tables and views	<pre> /*=====*/ /* Command on the source */ /*=====*/ select trim(T.TableName) TABLE_NAME, trim(T.TableName) RES_NAME, upper(substr(T.TableName, 1, 3)) TABLE_ALIAS, trim(T.TableKind) TABLE_TYPE, substr(T.CommentString, 1, 1000) TABLE_DESC from DBC.Tables T where T.DatabaseName = '<%=snpRef.getModel("SCHEMA_NAME")%>' and T.TableKind in ('T', 'V') and T.TableName like '<%=snpRef.getModel("REV_OBJ_PATT")%>' /*=====*/ </pre>

Step	Example of code
	<pre> /* Command on the target */ /*=====*/ insert into SNP_REV_TABLE (I_MOD, TABLE_NAME, RES_NAME, TABLE_ALIAS, TABLE_TYPE, TABLE_DESC, IND_SHOW) values (<%=snpRef.getModel("ID")%>, :TABLE_NAME, :RES_NAME, :TABLE_ALIAS, :TABLE_TYPE, :TABLE_DESC, '1') </pre>
Get Table Columns	<pre> /*=====*/ /* Command on the source */ /*=====*/ select trim(c.TableName) TABLE_NAME, trim(c.ColumnName) COL_NAME, substr(c.ColumnTitle,1,35) COL_HEADING, substr(c.CommentString,1,250) COL_DESC, CASE c.ColumnType WHEN 'BF' THEN 'BYTE' WHEN 'BV' THEN 'BYTE' WHEN 'CF' THEN 'CHAR' WHEN 'CV' THEN 'VARCHAR' WHEN 'D' THEN 'DECIMAL' WHEN 'DA' THEN 'DATE' WHEN 'F' THEN 'FLOAT' WHEN 'I' THEN 'INTEGER' WHEN 'I1' THEN 'INTEGER' WHEN 'I2' THEN 'SMALLINT' WHEN 'TS' THEN 'TIMESTAMP' WHEN 'SZ' THEN 'TIMESTAMP' ELSE c.ColumnType END DT_DRIVER, CASE WHEN c.ColumnId > 1024 THEN c.ColumnId - 1024 ELSE c.ColumnId END POS, CASE WHEN c.ColumnType = 'D' THEN c.DecimalTotalDigits WHEN c.ColumnType = 'TS' THEN c.DecimalFractionalDigits WHEN c.ColumnType = 'SZ' THEN c.DecimalFractionalDigits ELSE c.ColumnLength END LONGC, CASE WHEN c.ColumnType = 'D' THEN c.DecimalFractionalDigits ELSE 0 END SCALEC, CASE WHEN c.Nullable = 'Y' THEN '0' ELSE '1' END COL_MANDATORY, c.ColumnFormat COL_FORMAT, substr(c.DefaultValue, 1, 100) DEF_VALUE from DBC.Columns c, DBC.Tables t </pre>

Step	Example of code
	<pre> Where c.DatabaseName = '<%=snpRef.getModel("SCHEMA_NAME")%>' and c.DatabaseName = t.DatabaseName and c.TableName like '<%=snpRef.getModel("REV_OBJ_PATT")%>%' and c.TableName = t.TableName and t.TableKind = 'T' /*=====*/ /* Command on the target */ /*=====*/ insert into SNP_REV_COL (I_MOD, TABLE_NAME, COL_NAME, COL_HEADING, DT_DRIVER, COL_DESC, POS, LONGC, SCALEC, COL_MANDATORY, COL_FORMAT, DEF_VALUE, CHECK_STAT, CHECK_FLOW) values (<%=snpRef.getModel("ID")%>, :TABLE_NAME, :COL_NAME, :COL_HEADING, :DT_DRIVER, :COL_DESC, :POS, :LONGC, :SCALEC, :COL_MANDATORY, :COL_FORMAT, :DEF_VALUE, '1', '1') </pre>
Etc.	
Set Metadata	<pre> SnpsReverseSetMetaData -MODEL=<%=snpRef.getModel("ID")%> </pre>

Refer to the following RKMs for further details:

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 Oracle	Reverse-engineering Knowledge Module for Oracle
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 SnpsReverserSetMetadata 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)		Optional code of sub-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 ODI 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 (or foreign keys).

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
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: ODI 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

Column	Type	Mandatory	Description
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

Journalizing Knowledge Modules (JKM)

Journalizing Knowledge Modules are beyond the scope of this document. They are detailed further in the ODI guide for Change Data Capture.

Guidelines for Developing your own Knowledge Module

One of the main guidelines when developing your own KM is to never start from scratch. Oracle provides more than 80 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 ODI 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

ARCHITECTURE CASE STUDIES

Setting up Repositories

General Architecture for ODI Repositories

Section Architecture of Oracle Data Integrator (ODI)Sunopsis describes the general architecture of ODI repositories.

In a typical environment for a data warehouse project, you would create the following repositories:

- A single master repository that holds all the topology and security information. All the work repositories are registered in this master repository. This single master repository contains all the versions of objects that are committed by the designers.
- A **“Development”** work repository shared by all ODI designers. This repository holds all the projects and models under development.
- A **“Testing”** work repository shared by the IT testing team. This repository contains all the projects and models being tested for future release.
- A **“User Acceptance Tests”** work repository shared by the IT testing team and the business analysts. This repository contains all the projects and models about to be released. Business analysts will use the Metadata Navigator on top of this repository to validate the scenarios and transformations before releasing them to production.
- A **“Production”** work repository shared by the production team, the

operators and the business analysts. This repository contains all the projects and models in read-only mode for metadata lineage, as well as all the released scenarios.

- A **“Hot fix”** work repository shared by the maintenance team and the development team. This work repository is usually empty. Whenever a critical error happens in production, the maintenance team restores the corresponding projects and models in this repository and performs the corrections with the help of the development team. Once the problems are solved, the scenarios are released directly to the production repository and the new models and projects are versioned in the master repository.

This recommended architecture is described in the figure below:

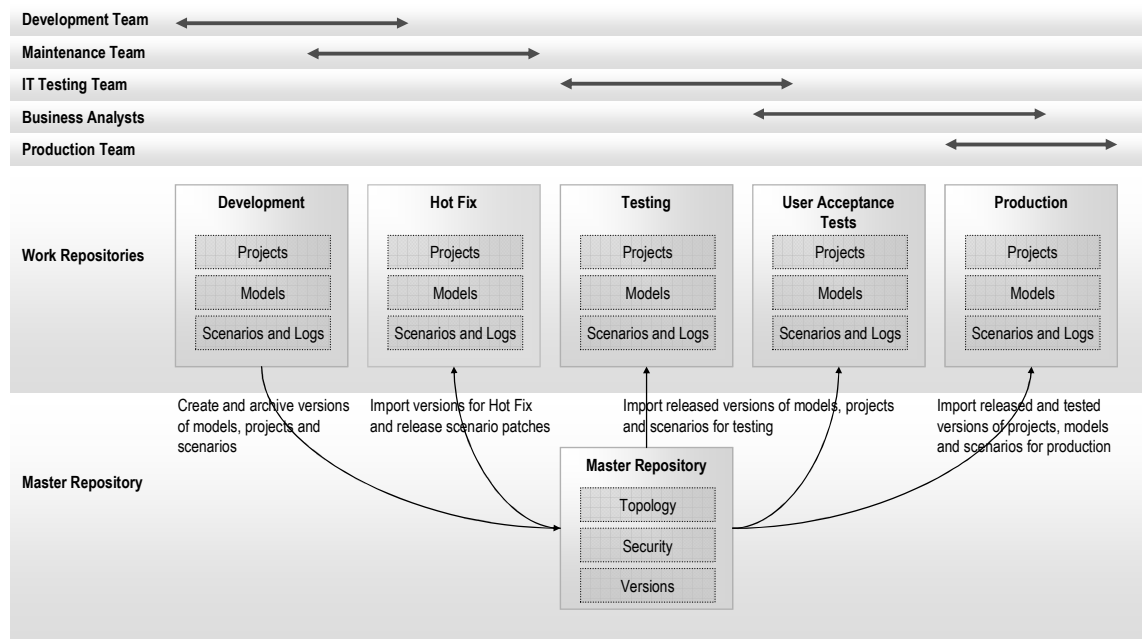


Figure 47: ODI Repositories and Teams Organization

The master repository and all work repositories are usually created in the same OLTP database instance in separate schemas or catalogs.

When developers have finished working on a project and decide to release it, they create a **version** for their projects and models and store it in the master repository (see section *Using Sunopsis ODI Version*). This version is then restored by the IT testing team in the testing repository. After the technical tests have completed, the testing team initializes the “user acceptance tests” repository for business analysts. They restore the same version they were working on to have it tested by business users. Once this version is functionally accepted, it is restored by the production team in the production repository.

When a critical bug is discovered in production, the developers are usually already working on the next release. Therefore they are usually not able to stop their

development and restore the previous version for corrections. The maintenance team is then in charge of restoring the version used in production into a separate empty work repository called “Hot Fix” and applying the appropriate fixes. Once done, the maintenance team releases its modified projects, models and scenarios into the master repository so that the production team can restore them as a patch in the production repository.

Creating a Separate Master Repository for Production

For some particular security requirements, you may not want to share the same master repository between development and production. In this case, the solution is to duplicate the master repository and to isolate the production environment from other environments as shown below:

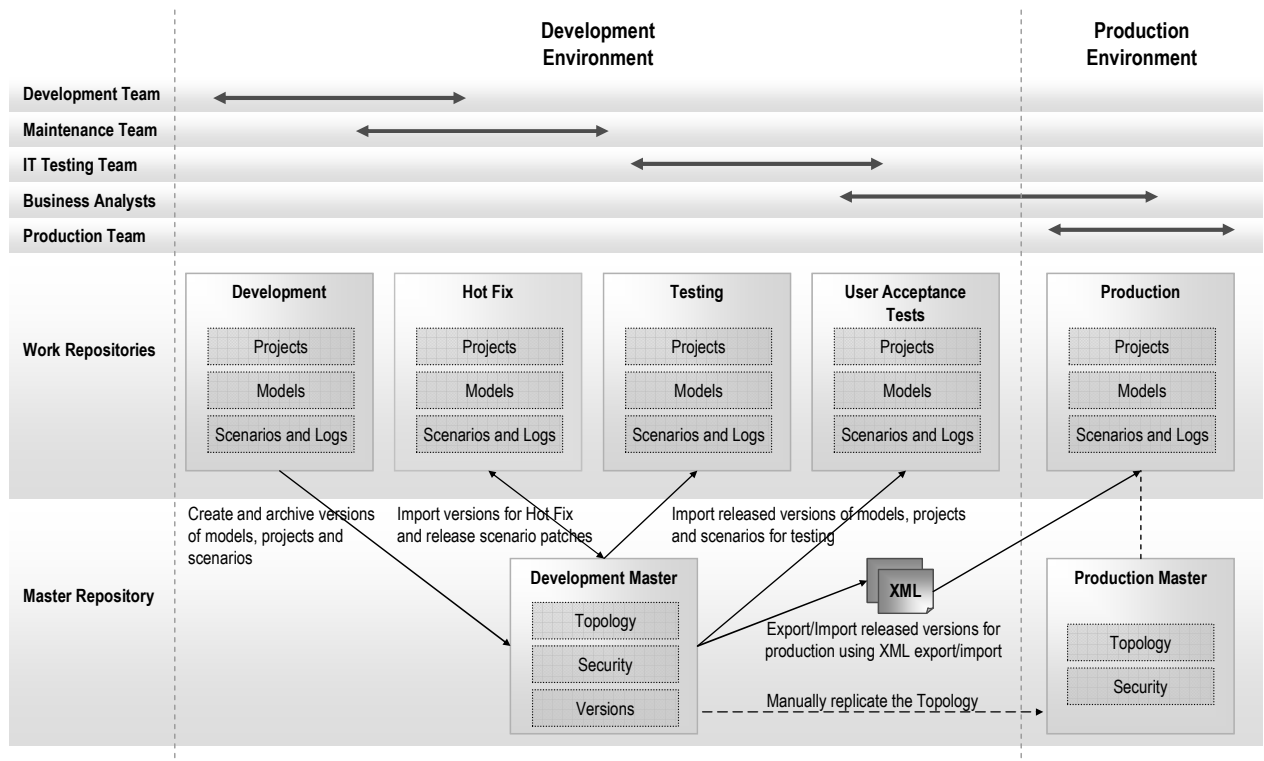


Figure 48: Several Master Repositories

To create a new master repository for your production environment, it is recommended that you use an ODI export of your master from your Topology and then use the master import utility “mimport”. You can find the “mimport” utility in the bin directory of ODI. When creating the new master repository, you should assign a new ID to it, different from the ID used by the development master repository.

Once created, do the following to set up the production environment:

- Create the production context

- Create all the production data servers and physical schemas in the physical architecture of the Topology
- Link your production physical schemas to the existing logical schemas defined by the designers in the production context.
- Do not change or remove existing contexts and logical schemas.
- Update the Security so that only production users and business analysts can access the repository.
- Create the production work repository and give it an ID different from any of the IDs used by the other work repositories (Development, Hot Fix, Testing and User Acceptance Tests). See *Understanding the Impact of Work Repository IDs*.
- Every time the development master repository is updated, manually replicate the changes in the production master repository.
- Export projects, models and scenarios that are ready for production from the development master repository into XML files. You can use the “Version Browser” to do so. Alternatively, if using Solutions, export your solutions into compressed files
- Connect to the production work repository with Designer and import the XML files or the solution’s compressed files.

Understanding the Impact of Work Repository IDs

When creating a master repository or a work repository, ODI asks you for a 3 digit ID for this repository. You should select this ID by following the rules below:

- Every master repository created in your enterprise should have a unique ID.
- Every work repository created in your enterprise should have a unique ID even if it belongs to a different master repository.

Every type of object in an ODI repository has a unique ID calculated according to the following rule:

<auto number> concatenated with the 3 digits of the repository ID.

For example, if the internal ID of an interface is 1256023, you can automatically guess that it was first created in the work repository ID 023.

The main goal of this rule is to enable export and import in “Synonym mode” of objects across work repositories without any risk of ID collision.

If 2 work repositories have the same ID, there is a chance that 2 different objects within these 2 repositories have the same ID. Therefore, importing the first object from the first repository to the second may overwrite the second object! The only way to avoid that is, of course, to have 2 different work repository IDs.

Using ODI Version Management

How Version Management Works

Version management in ODI is designed to allow you to work on different versions of objects across work repositories as shown below:

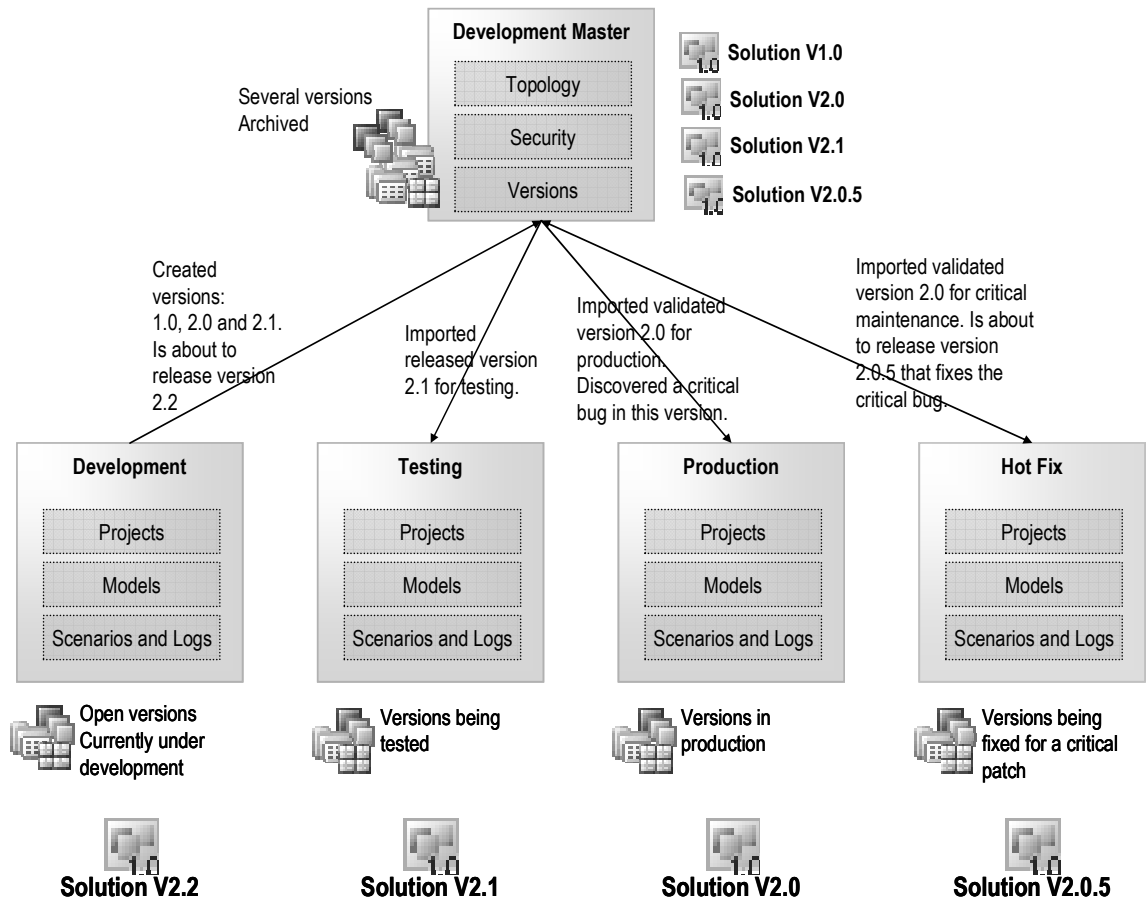


Figure 49: Version Management with ODI

Note:

The user acceptance tests repository does not appear in this diagram as the mechanism for populating it with new versions is similar to the one used for the testing repository.

The developers have already released versions 1.0, 2.0 and 2.1 of their development in the master repository. Every time they have released a version they have used a Solution. See *Using Solutions for Configuration Management* for details. They are now working on the next release as they are about to release solution version 2.2. Meanwhile, the IT testing team is testing version 2.1 in its work repository. The production team is still working on version 2.0 and it has discovered a critical bug in this version. Therefore, the maintenance team has restored version 2.0 in the Hot Fix work repository and is working on a patch release, v2.0.5. As soon as this patch release is committed to the master repository, the production team will have

to restore it into the production repository. Developers will have to manually update versions 2.1 and 2.2 so they reflect the changes made by the maintenance team to version 2.0.5.

Note:

Every work repository can only have a single version of an object.

Creating and Restoring Versions of Objects

To create a version of an object, simply right-click on the object and select “Version->Create...”.

When the version of the object is created, it becomes available in the “Version Browser” and in the “Version” tab of the object. ODI also updates all the “I” and “U” flags relating to this object to indicate that it is up to date with the version in the master repository. If you change your object in the work repository after committing it in the master, you will notice that its icon is changed and a small “U” appears besides it to indicate that its status is now “Updated”. This feature is very helpful as it shows you all the differences between the current version and the last version committed in the master repository.

When you create a version of an object, the object is exported to XML in memory, compressed and stored in the master repository as a binary large object. Therefore, you should consider creating a version of an object only when you are about to commit it definitely for a release.

To restore an object to one of its previous versions, simply right-click on the object and select “Version->Restore...”. Then select from the list of available versions which one to restore. You should use caution for the following reasons:

- All updates that you have done since the last time you have created a version of your current object will be lost
- The object that you are about to restore may reference other objects that were removed from your work repository. The typical situation is restoring an interface that references a column that doesn’t exist anymore in your models. If this happens, the restored object will be marked as “invalid” with a red exclamation mark and you will have to edit it to correct all missing references.

Using Solutions for Configuration Management

During design time you will probably create versions of your objects for backup. For example, you will create versions of your models, folders, interfaces, variables, Knowledge Modules, etc. Most of these objects are designed to work together as they may depend on one another. For example, interface version 2.1.0 requires

tables from model 3.1.2 and Knowledge Modules version 1.2 and 4.5. Maintaining the list of dependencies can be very tedious and time consuming. Therefore, when you are about to release what has been developed for the testing team, you would prefer to have a single object that manages these dependencies.

ODI solution objects are designed to manage these dependencies between a project and all models, global objects and scenarios it references. When you are about to release your development, you simply need to create a new solution and drag and drop your project into this solution. ODI will then create a new version of your project (if required) and a new version of every model, scenario, other project or global object referenced by this project. Therefore, instead of releasing several objects, you simply release the solution.

Notes:

For performance reasons, you should consider having small projects to improve the efficiency of the version creation mechanism. It is recommended that you split your development into several small projects containing less than 300 objects.

When calculating the dependencies for a project in order to build the solution, ODI creates a version of every model referenced by this project. Then it looks at every model to see what Knowledge Modules it references (RKM, JKM and CKM). And finally it creates a new version of every project to which these Knowledge Modules belong. So if your models reference Knowledge Modules from 3 different projects, the solution will reference these 3 projects as well. A best practice to avoid this is to create a project called “KMs” and to have in this project all the RKMs, CKMs and JKMs that you will reference in your models. Therefore every time you create a solution, this project will be added as a dependency, instead of 3 or 4 different projects.

Going to Production

Releasing Scenarios

Section *Using SunopsisODI Version Management* gives an overview on how to handle versions of objects to release them to the production environment. However, sometimes you will only want your scenarios in the production work repository without their corresponding projects and models. For that, you would create an “Execution” work repository rather than a “Development” work repository. When your scenarios are released into the master repository or as simple XML files, you would either restore them to your work repository or import them in “synonym” or “duplication” mode from the Operator user interfaces.

Executing Scenarios

Executing a Scenario Manually

You can execute your scenario manually from Operator. In this case if your scenario requires variables to be set, you would edit every variable used by the scenario and update its default value. To execute the scenario, you also need to

indicate the context for the execution as well as the logical agent in charge of this execution.

An alternative way to execute a scenario is to use Metadata Navigator to start the scenario from your web browser. This approach can be useful if you are considering having scenarios that will be executed on-demand by your business users and analysts. Should your scenario require variables, the user would be asked to fill in the variable values accordingly as described in the figure below.

The screenshot shows the Metadata Navigator web interface. The left sidebar contains a navigation menu with sections: Metadata, Execution (with sub-items: Scenarios, Execute a Scenario, Restart, Scheduler, All Sessions, Parent Sessions), Topology, and Other Links (with sub-items: Search, About, Logout). The main content area is titled 'Execute a Scenario' and features a 'Select a Scenario' section with dropdown menus for Scenario (UPDATE_FINANCIAL_RESULTS (v.001)), Agent (PULSAR1), Context (3 - San Francisco), and Log level (5). Below this is a 'Scenario Variables' section with a table for inputting variable values.

Variable name	Variable value	Datatype
ELT_DEMO.BUSINESS_UNIT_ID =	BU1253	Alphanumeric
ELT_DEMO.CLOSING_DATE =	02/05/2006	Date

An 'Execute' button is located at the bottom of the form.

Figure 50: Executing a Scenario from Metadata Navigator

Executing a Scenario Using an Operating System Command

You can also start a scenario from an operating system command using the “startscen” shell script. You would do that if you plan to use an external scheduler to schedule your jobs. Refer to the ODI on-line documentation for details on this operating system command

When starting a scenario from the operating system, you do not need a running agent. The “startscen” command has its own built-in agent that executes the scenario and stops at the end of the session.

Executing a Scenario Using an HTTP Request

The ODI on-line documentation gives you examples on how to start a scenario using an HTTP request. This mechanism requires both having Metadata Navigator properly set up and a running agent.

You can consider this way of starting a scenario if you plan to invoke ODI from your own application. Your application would simply format the HTTP post

request with appropriate parameters and wait for an HTTP reply from Metadata Navigator.

Assigning Session Keywords for a Scenario

You may occasionally find it difficult to browse the Operator log as it may contain a large number of sessions. To overcome this limitation, ODI allows keywords to be assigned to a session, that will let this session be stored automatically in an appropriate session folder that you create in Operator.

In the figure below, the folder “Product Dimensions” was created for the “PDIM” keyword. When starting the LOAD_FAMILY_HIERARCHY and LOAD_PRODUCTS scenarios, the “-KEYWORDS” parameter was set to “-KEYWORDS=PDIM”. Therefore when using the operator to access the executed sessions, they are automatically available under the “Product Dimensions” folder.

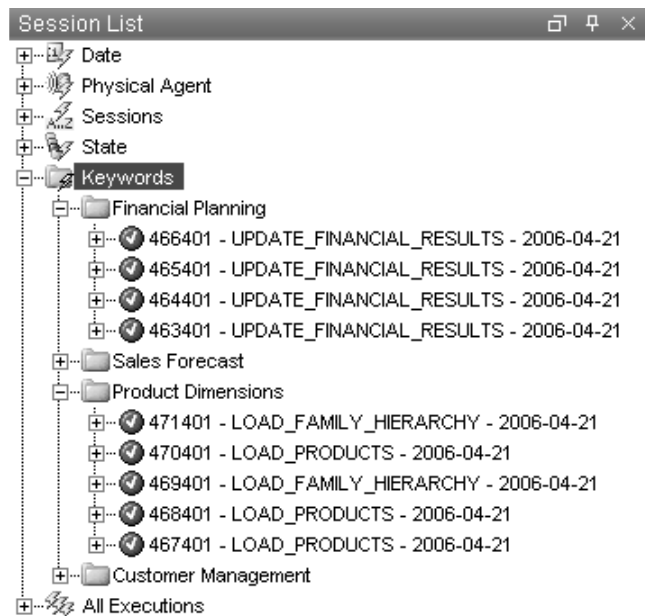


Figure 51: Session Folders and Keywords

You can use this feature to distribute your sessions across your business areas.

Retrieving ODI Logs after the Execution of a Scenario

Although both the Operator and Metadata Navigator offer an intuitive user interface to access ODI logs, there may be some cases where you would like to access the log information from your own code.

The quickest way to do this is to access the ODI work repository using a SQL connection and query the log tables. The key to these tables is the session ID that you obtain right after starting the scenario. When starting your scenario using HTTP, the session ID is returned in the HTTP reply. When starting your scenario

from an operating system command, you should set verbose mode to 1 (“-V=1”) and analyze standard output.

You can also execute a scenario that uses a procedure that dumps the ODI logs into formatted files. This scenario would be systematically triggered at the end of all your scenarios.

The following SQL queries give you a starting point to retrieving information about a session and its steps:

Obtaining information about a session:

```
select  SESS_NO "SessionID",
        SESS_NAME      "SessionName",
        SCEN_NAME      "Scenario",
        SCEN_VERSION    "Version",
        case
            when SESS_STATUS = 'E' then 'Error'
            when SESS_STATUS = 'W' then 'Waiting'
            when SESS_STATUS = 'D' then 'Done'
            when SESS_STATUS = 'Q' then 'Queued'
            when SESS_STATUS = 'R' then 'Running'
        end          "Session Status",
        SESS_RC        "Return Code",
        SESS_MESS       "Error Message",
        SESS_BEG        "Start Datetime",
        SESS_DUR        "Session Duration (s)"
from    SNP_SESSION
where   SESS_NO = $$SESS_NO$$
```

Obtaining information about session steps:

```
select  SST.SESSION_NO  "SessionID",
        SST.STEP_NAME   "Step Name",
        case
            when SST.STEP_TYPE like 'C%' then 'Data Quality Check'
            when SST.STEP_TYPE like 'J%' then 'Journalizing'
            when SST.STEP_TYPE like 'R%' then 'Reverse Engineering'
            when SST.STEP_TYPE like 'V%' then 'Variable'
            when SST.STEP_TYPE like 'F%' then 'Interface'
            when SST.STEP_TYPE like 'T%' then 'Procedure'
            when SST.STEP_TYPE like 'O%' then 'Operating System'
        end          "Step Type",
        SL.STEP_BEG      "Start Date",
        SL.STEP_DUR      "Step Duration (s)",
        case
            when SL.STEP_STATUS = 'E' then 'Error'
            when SL.STEP_STATUS = 'W' then 'Waiting'
            when SL.STEP_STATUS = 'D' then 'Done'
            when SL.STEP_STATUS = 'R' then 'Running'
        end          "Step Status",
        SL.NB_ROW        "Rows Processed",
        SL.NB_INS        "Inserts",
        SL.NB_UPD        "Updates",
        SL.NB_DEL        "Deletes",
        SL.NB_ERR        "Errors",
        SL.STEP_RC        "Return Code",
        SL.STEP_MESS      "Error Message"
from    SNP_SESSION_STEP SST, SNP_STEP_LOG SL
```

```

where SST.SESS_NO = $$SESS_NO$$
and    SST.SESS_NO = SL.SESS_NO
and    SST.NNO = SL.NNO
order by SL.STEP_BEG, SL.NNO

```

For a complete list of tables and columns with their description, ask for the ODI Repository Model documentation from oracledi-pm_us@oracle.com.

Scheduling Scenarios Using the Built-in Scheduler

You can use ODI's built-in scheduler to schedule your scenarios. However, if your enterprise is already using an enterprise-wide scheduler, you may wish to use this instead, as described in the previous sections.

Refer to the ODI online documentation for information on how to assign a schedule to a scenario.

The “agentscheduler” shell script allows you to start an ODI agent as a scheduler.

Using Operator in Production

ODI Operator is the module that is used the most in a production environment. From this module, operators have the ability to:

- Import scenarios into the production work repository
- Execute scenarios manually and assign default values to scenario variables
- Follow execution logs with advanced statistics including the number of row processed and the execution elapsed time
- Abort a currently-running session. Note that aborting a session is not always supported by the underlying JDBC drivers. Jobs already submitted to databases may sometimes continue running.
- Restart a session that has failed and possibly redefine the restart point by updating the task statuses.
- Create session folders to hold sessions with pre-defined keywords
- Access scheduling information

Using Metadata Navigator in Production

Metadata Navigator can be used in production for 2 purposes:

- To give business users, analysts and operators access to metadata lineage, metadata searching, scenario execution, and log monitoring. In this situation, the production work repository should be a “development” repository, and all the released projects, models and scenarios should be restored in this repository.

- To give operators access only to scenarios and logs. In this case, the production work repository would only be an “execution” work repository, and would only contain the released scenarios.

Note:

Not all operations that can be carried out in Operator can be done in Metadata Navigator. The following functions are not available in Metadata Navigator:

- Importing a scenario
- Aborting a session
- Redefining the restarting point of a session
- Accessing session folders
- Accessing scheduling information

Setting up Agents

Where to Install the Agent(s)?

A typical data warehouse implementation usually requires a single ODI agent in production. The agent is usually installed on the host machine that is used to load data in the data warehouse. The agent requires a connection to source databases or files, and triggers the appropriate load utilities. The operating system of this host machine can be Unix, Linux, Windows or zOS provided that a Java Virtual Machine 1.4 is installed.

The network bandwidth between this machine and the data warehouse should be large enough to handle the volume of data that will be loaded by the utilities in the warehouse database. On the other hand, as the agent may need to access other source servers, close attention should be paid to the network bandwidth to and from these source servers.

If your Knowledge Modules generate operating-system-specific commands, these must match the operating system on which the agent is installed.

In a normal environment, you will set up:

- 1 physical agent for the development team, preferably on the same operating system as the production agent
- 1 physical agent shared by the testing team, the maintenance team and the user acceptance team. Again, try to use the same operating system as the agent in production.
- 1 physical agent for the production

Using Load Balancing

There are cases when a single agent can become a bottleneck, especially when it has to deal with large numbers of sessions started in parallel. For example, suppose you want to retrieve source data from 300 stores. If you attempt to do this in parallel on

a single agent, it may lead to excessive overhead as it will require opening 300 threads (1 for every session). A way to avoid that is to set up load balancing across several agents.

To set up load balancing in ODI, you can follow these steps in the Topology:

- Define the agent that will be in charge of distributing the workload. In the following, this agent will be referred to as the “master agent”. The master agent usually has only 2 concurrent sessions. All the scenarios will be executed by this single agent by using the `-AGENT` parameter.
- Define several child agents that will be in charge of actually executing the scenarios. Set the maximum number of sessions of each agent to a value between 2 and 20.
- Edit the master agent in the Topology, and in the “Load Balancing” tab, select all your child agents.

The diagram below gives you an example of how you can setup this architecture:

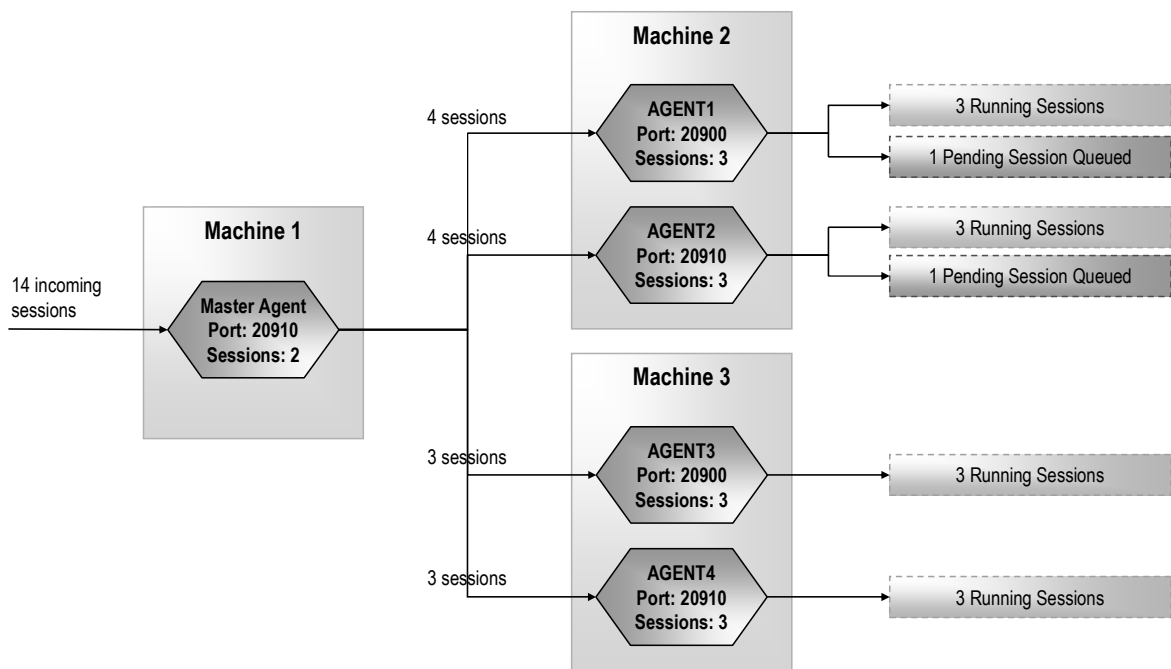


Figure 52: Example of Load Balancing

In this example, 4 agents are installed on 2 different machines. Each of them accepts a maximum of 3 sessions in parallel. The master agent is installed on another machine.

The master agent receives 14 requests to start several scenarios in parallel. It distributes these 14 requests to the available agents. Each agent will start 3 scenarios in parallel, and the first 2 agents will simply queue the remaining sessions for future execution. As soon as one of the agents becomes available, it will pick one of the queued sessions and begin executing it.

The master agent initially calculates the workload per agent according to the ratio given by the number of sessions currently being executed versus the maximum number of sessions allowed for this agent. It distributes incoming sessions one by one to every agent and then recalculates the workload's ratio. As the queued sessions for an agent can be picked up by another agent, the entire system becomes balanced.

This flexible architecture allows you to add as many agents as needed to reach the maximum scalability for your system. If you plan to start several scenarios in parallel, you can refer to section *Example4: Starting Multiple Scenarios in Parallel* for a detailed example on how to achieve that elegantly.

Notes:

- The number of agents required and the maximum number of sessions that they may support will depend on your environment settings. It should be tested carefully before going to production.
- The main bottleneck in this architecture is obviously the RDBMS hosting the ODI repositories as all agents access the same database tables. It is recommended that you dedicate a database server to the ODI repository if you plan to setup such architecture. You would therefore have more flexibility to tune this server in the case of locks or performance issues.

Backing up Repositories

ODI recommends that you backup your master and work repositories on a daily basis to avoid any loss of your work.

To backup your repositories, you simply need to backup every schema on which you have installed a master or work repository, using your standard database backup procedure.

Refer to your specific database documentation for information on how to backup a schema or catalog.

APPENDICES

APPENDIX I. ODI FOR TERADATA BEST PRACTICES

Architecture of ODI Repositories

It is recommended that you install the master and work repositories in an OLTP database distinct from the source or target applications. Teradata database is not recommended for hosting ODI repositories.

Reverse-engineering a Teradata Schema

RKM	Description
RKM Teradata	Reverse-engineering Knowledge Module for Teradata

The Teradata JDBC driver implements most metadata APIs. You can therefore use the standard JDBC reverse-engineering with ODI. Using this method,, you can capture metadata for:

- Tables and Views including table comments,
- Columns including data types, length and scale, and comments
- Primary Keys when they are defined using a PRIMARY KEY constraint statement in the databases

However, you will probably face the following limitations:

- Foreign Key metadata is not implemented in the JDBC driver (ITU 7 and 8). Therefore ODI will not retrieve foreign keys.
- Unique Primary Indexes (UPI) and Non Unique Primary Indexes (NUPI) are not imported. To have a UPI imported as a primary key in ODI, you need to define it as a PRIMARY KEY constraint at the database level.
- Other indexes are not imported
- Check constraints are not imported

You can bypass some of these limits by using the Reverse Knowledge Module for Teradata provided with the Open Connector for Teradata. This RKM is based on the DBC catalogue tables (DBC.Tables, DBC.Columns etc.) and is discussed further in this document. You may also enhance this Knowledge Module and adapt it to your needs.

ODI Agent accepts binary files and supports several native binary data types such as Binary Big and Little Endian, EbcDic, EbcDic zoned decimals, Packed decimals, etc. However, we recommend against using ODI Agent to load large binary files into the Teradata staging area. As the agent uses JDBC to write to the target, using this method would lead to significant performance issues compared to loading using the native Fastload utility. Refer to the Fastload documentation for a way to load binary files in a Teradata table. The LKM File to Teradata (Fastload) should meet your needs for generating and executing the appropriate Fastload scripts.

Teradata Loading Strategies

LKM	Description
LKM SQL to SQL	Loads data through the agent from any SQL RDBMS to any SQL RDBMS staging area
LKM File to Teradata (TPUMP-FASTLOAD-MULTILOAD)	Loads a flat file to the staging area using Tpump, TPT, FastLoad, MultiLoad, or Parallel Transporter. The utilities must be installed on the machine hosting the ODI Agent.
LKM SQL to Teradata (piped TPUMP-FAST-MULTILOAD)	Loads a remote RDBMS result set to the staging area using a piped Unload and Tpump, TPT, FastLoad, MultiLoad or or Parallel Transporter. The utilities must be installed on the Unix machine hosting the ODI Agent.

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”.

Step	Example of code
Genera Fast or Mload or TPump script. This step uses the SnpsOutFile Tool to generate the script. The Java “if” statement is used to verify	<pre> SnpsOutFile -File=<%=snpRef.getSrcTablesList("", "[WORK_SCHEMA]", "", "")%>/<%=snpRef.getTable("L", "COLL_NAME", "W")%>.script <% if (snpRef.getOption("TERADATA UTILITY").equals("fastload")) {%> SESSIONS <%=snpRef.getOption("SESSIONS")%> ; LOGON <%=snpRef.getInfo("DEST_DSERV_NAME")%>/<%=snpRef.getInfo("DEST_USER_NAME")%>,<%=snpRef.getInfo("DEST_PASS")%> ; </pre>

<p>which type of script to generate.</p>	<pre> BEGIN LOADING <%=snpRef.getTable("L", "COLL_NAME", "W")%> ERRORFILES <%=snpRef.getTable("L", "COLL_NAME", "W")%>_ERR1, <%=snpRef.getTable("L", "COLL_NAME", "W")%>_ERR2 ; <%if (snpRef.getSrcTablesList("", "[FILE_FORMAT]", "", "").equals("F")) {%> SET RECORD TEXT; DEFINE <%=snpRef.getColList("", "\t[CX_COL_NAME] (char([LONGC]))", ",\n", "", "")%> <%} else {%> SET RECORD VARTEXT "<%=snpRef.getSrcTablesList("", "[FILE_SEP_FIELD]", "", "")%>"; DEFINE <%=snpRef.getColList("", "\t[CX_COL_NAME] (varchar([LONGC]))", ",\n", "", "")%> <%}%> FILE=<%=snpRef.getSrcTablesList("", "[SCHEMA]/[RES_NAME]", "", "")%> ; INSERT INTO <%=snpRef.getTable("L", "COLL_NAME", "W")%> (<%=snpRef.getColList("", "\t[CX_COL_NAME]", ",\n", "", "")%>) VALUES (<%=snpRef.getColList("", "\t:[CX_COL_NAME] [COL_FORMAT]", ",\n", "", "")%>) ; END LOADING ; LOGOFF ; <%}%> <% if (snpRef.getOption("TERADATA UTILITY").equals("multiload")) {%> ... etc. ... <%}%> </pre>
<p>Execute the load utility. This step uses Jython to execute an operating system command rather than the built-in “OS Command” technology</p>	<pre> import os logname = "<%=snpRef.getSrcTablesList("", "[WORK_SCHEMA]", "", "")%>/<%=snpRef.getTable("L", "COLL_NAME", "W")%>.log" scriptname = "<%=snpRef.getSrcTablesList("", "[WORK_SCHEMA]", "", "")%>/<%=snpRef.getTable("L", "COLL_NAME", "W")%>.script" utility = "<%=snpRef.getOption("TERADATA UTILITY")%>" if utility == "multiload": utility="mload" loadcmd = '%s < %s > %s' % (utility,scriptname, logname) if os.system(loadcmd) <> 0 : raise "Load Error", "See %s for details" % logname </pre>

The following table gives you extracts from the “LKM File to Teradata (TPUMP-FASTLOAD-MULTILOAD)” that uses this strategy. Refer to the KM for the complete code:

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.

The “LKM SQL to Teradata (TPUMP-FASTLOAD-MULTILOAD)” follows these steps and uses the generic SnpsSqlUnload 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 SnpsSqlUnload	<pre> SnpsSqlUnload "-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() %> </pre>

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).

ODI 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 SnpsSqlUnload 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 SnpsSqlUnload command	<pre> import com.sunopsis.dwg.tools.SnpsSqlUnload as JSnpsSqlUnload 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 snpssqlunload(): snpunload = JSnpsSqlUnload() # Set the parameters cmdline = JVector() 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)) snpunload.setParameters(cmdline) # Start the unload process snpunload.execute() </pre>
Main function that runs the piped load	<pre> ... utility = "<%=snpRef.getOption("TERADATA UTILITY")%>" if utility == "multiload": </pre>

Step	Example of code
	<pre> 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") 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: snpssqlunload() 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 </pre>

Step	Example of code
	<pre> # At this point, the unload() process has finished, so we need to wait # for the load process to finish (join the thread) loadthread.join() </pre>

Teradata Integration Strategies

IKM	Description
IKM SQL Control Append	Integrates data in any ISO-92 compliant database target table in TRUNCATE/INSERT (append mode.) Data quality can be checked. Invalid data is rejected in the “E\$” error table and can be recycled.
IKM Teradata Incremental Update	Set-based incremental update for Teradata
IKM Teradata Slowly Changing Dimension	Slowly Changing Dimension for Teradata
IKM File to Teradata (TPUMP-FAST-MULTILOAD)	Integrates data from a File to a Teradata table without staging the file data. The Utility used for loading can be either Fastload, Multiload, Tpump, TPT or Parallel Transporter.
IKM SQL to SQL Append	Integrates data into an ANSI-SQL92 target database from any remote ANSI-SQL92 compliant staging area in replace or append mode.
IKM SQL to Teradata (piped TPUMP-FAST-MULTILOAD)	Integration from any SQL compliant staging area to Teradata using either FASTLOAD, TPT, MULTILOAD or TPUMP. Data is extracted using SnpsSqlUnload into a Unix FIFO and loaded using the appropriate utility.
IKM SQL to Teradata (TPUMP-FAST-MULTILOAD)	Integration from any SQL compliant staging area to Teradata using either FASTLOAD, TPT, MULTILOAD or TPUMP. Data is extracted using SnpsSqlUnload into a temporary file and loaded using the appropriate utility.
IKM Teradata to File	Exports data from a Teradata staging area to an

IKM	Description
(FASTEXPORT)	ASCII file using FastExport

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, ODI 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.

The following table shows some excerpts from the IKM File to Teradata (TPUMP-FAST-MULTILOAD), which allows the generation of appropriate scripts for each of these Teradata utilities depending on your integration strategy:

Step	Example of code
Generate Fastload, MultiLoad or T pump script	<pre> SnpsOutFile -File=<%=snpRef.getSrcTablesList("", "[WORK_SCHEMA]", "", "")%>/<%=snpRef.getTargetTable("SCHEMA")%>.<%=snpRef.getTargetTable("RES_NAME")%>.script <% if (snpRef.getOption("TERADATA UTILITY").equals("fastload")) {%> SESSIONS <%=snpRef.getOption("SESSIONS")%> ; LOGON <%=snpRef.getInfo("DEST_DSERV_NAME")%>/<%=snpRef.getInfo("DEST_USER_NAME")%>,<%=snpRef.getInfo("DEST_PASS")%> ; BEGIN LOADING <%=snpRef.getTargetTable("SCHEMA")%>.<%=snpRef.getTargetTable("RES_NAME")%> ERRORFILES <%=snpRef.getTargetTable("WORK_SCHEMA")%>.<%=snpRef.getTargetTable("RES_NAME")%>_ERR1, <%=snpRef.getTargetTable("WORK_SCHEMA")%>.<%=snpRef.getTargetTable("RES_NAME")%>_ERR2 ; <%if (snpRef.getSrcTablesList("", "[FILE_FORMAT]", "", "").equals("F")) {%> SET RECORD TEXT; DEFINE <%=snpRef.getColList("", "\t[CX_COL_NAME] (char([LONGC]))", "", "\n", "","")%> <%} else {%> SET RECORD VARTEXT "<%=snpRef.getSrcTablesList("", "[SFILE_SEP_FIELD]", "", "")%>"; DEFINE <%=snpRef.getColList("", "\t[CX_COL_NAME] (varchar([LONGC]))", </pre>

Step	Example of code
	<pre> ",\n", "", "")%> <%}%> FILE=<%=snpRef.getSrcTablesList("", "[SCHEMA]/[RES_NAME]", "", "")%> ; INSERT INTO <%=snpRef.getTargetTable("SCHEMA")%>.<%=snpRef.getTargetTable("RES _NAME")%> (<%=snpRef.getColList("", "\t[CX_COL_NAME]", ",\n", "", "")%>) VALUES (<%=snpRef.getColList("", "\t:[CX_COL_NAME] [COL_FORMAT]", ",\n", "","")%>) ; END LOADING ; LOGOFF ; <%}%> <% if (snpRef.getOption("TERADATA UTILITY").equals("multiload")) {%> [etc.] <%}%> </pre>
Start the load using a Jython script	<pre> import os logname = "<%=snpRef.getSrcTablesList("", "[WORK_SCHEMA]", "", "")%>/<%=snpRef.getTargetTable("SCHEMA")%>.<%=snpRef.getTargetTabl e("RES_NAME")%>.log" scriptname = "<%=snpRef.getSrcTablesList("", "[WORK_SCHEMA]", "", "")%>/<%=snpRef.getTargetTable("SCHEMA")%>.<%=snpRef.getTargetTabl e("RES_NAME")%>.script" utility = "<%=snpRef.getOption("TERADATA UTILITY")%>" if utility == "multiload": utility="mload" loadcmd = '%s < %s > %s' % (utility,scriptname, logname) if os.system(loadcmd) <> 0 : raise "Load Error", "See %s for details" % logname </pre>

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.

The following IKMs implement this strategy:

IKM	Description
IKM SQL to Teradata (pipelined TPUMP-FAST-MULTILOAD)	Integration from any SQL compliant staging area to Teradata using either FASTLOAD, TPT, MULTILOAD or TPUMP. Data is extracted using SnpsSqlUnload into a Unix FIFO and loaded using the appropriate utility.
IKM SQL to Teradata (TPUMP-FAST-MULTILOAD)	Integration from any SQL compliant staging area to Teradata using either FASTLOAD, TPT, MULTILOAD or TPUMP. Data is extracted using SnpsSqlUnload into a temporary file and loaded using the appropriate utility.

Server to File or JMS Append

When the target datastore is a file or JMS queue or topic you will obviously 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 specific “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 ODI features. Or you can use specific unload utilities such as Teradata FastExport if the target is not JMS based.

You can refer to the following IKMs for further details:

IKM	Description
IKM Teradata to File (FASTEXPORT)	Exports data from a Teradata staging area to an ASCII file using FastExport
IKM SQL to File Append	Exports data from any SQL compliant staging area to an ASCII or BINARY file using the ODI File driver

Knowledge Module Options

KM Option	Option Description
TERADATA_UTILITY	Choose the teradata utility to load data choose:

KM Option	Option Description
	<p>TPUMP - to use tpump utility MLOAD - to use multiload utility FASTLOAD - to use fastload utility TPT-LOAD - to use parallel transporter (load operator) TPT-SQL-INSERT to use parallel transporter (sql insert operator)</p> <p>For details and appropriate choice of utility/ load operator, see Teradata documentation: TPUMP Teradata Parallel Data Pump Reference MLOAD Teradata MultiLoad Reference FASTLOAD Teradata FastLoad Reference TPT-LOAD Teradata Parallel Transporter Reference Teradata Parallel Transporter User Guide TPT-SQL-INSERT Teradata Parallel Transporter Reference Teradata Parallel Transporter User Guide</p>
REPORT_NB_ROWS	Report the number of rows processed as an exception to the Sunopsis Log (a task reporting the number of rows processed should appear as a warning in the log).
SESSIONS	Number of sessions
DROP_TTU_TABLES	<p>This option is intended for development use only. Do NOT use in production!</p> <p>If activated, drops all Teradata utility tables like</p> <ul style="list-style-type: none"> - acquisition error table - application error table - log table - work table
MAX_ALLOWED_ERRORS	<p>Maximum number of tolerated errors. This corresponds to</p> <ul style="list-style-type: none"> - ERRLIMIT command in FastLoad, MultiLoad and TPump - ErrorLimit attribute in TPT Load operator <p>Note: TPT SQL Insert operator does not support any error limit. This option is ignored for TPT-SQL-INSERT.</p>

KM Option	Option Description
COLLECT_STATS	Collect statistics on primary index of flow and target table?
PRIMARY_INDEX	<p>Specify primary index for flow table. Teradata uses the PI to spread data across AMPs. It is important that the chosen PI has a high cardinality (many distinct values) to ensure evenly spread data to allow maximum processing performance. Please follow Teradata's recommendation on choosing a primary index, e.g. Teradata Database Design, Chapter "Primary Indexes"</p> <p>Valid options are [PK] Primary key of target table [UK] Update key set in interface <list of cols></p> <p>This comma separated list of target columns defines the primary index. If no value is set (empty), no index will be created. NB: Teradata will automatically choose the first column of the table for the PI.</p>
VALIDATE	<p>This option generates an extra validation step during development. Validations performed are:</p> <ul style="list-style-type: none"> - are all target columns mapped? - is physical length information set for all target columns? <p>This option should be turned off for all production use in particular for high-frequency execution. There is not added value in production, only processing overhead.</p>
ODI_DDL	<p>The generation of DDL commands can be controlled by setting ODI_DDL:</p> <p>DROP_CREATE - Drop and recreate all ODI temp tables at every execution CREATE_DELETE_ALL - Create all temp tables (ignore warnings) and use DELETE ALL DELETE_ALL - Only submit DELETE ALL for all temp tables NONE - Do not submit any DDL</p>

KM Option	Option Description
FLOW_TABLE_TYPE	SET or MULTiset table
FLOW_TABLE_OPTIONS	Additional options for the temporary I\$ table. See documentation for valid values: - [NO] FALLBACK - [NO DUAL] [BEFORE AFTER] JOURNAL etc.
NP_USE_NAMED_PIPE	Use named pipes for data transport?
NP_ACCESS_MODULE_NAME	Depending on platform the Teradata utility is used, different Teradata access modules need to be used for named pipes: For FastLoad, MultiLoad and TPump: - np_axsmo.sl on HP-UX platforms - np_axsmo.so on MP-RAS; IBM-AIX; Solaris SPARC and Solaris Opteron platforms - np_axsmo.dll on Windows platforms For TPT: - np_axsmoTWB.sl on HP-UX platforms - np_axsmoTWB.so on MP-RAS; IBM-AIX; Solaris SPARC and Solaris Opteron platforms - np_axsmoTWB.dll on Windows platforms Please refer to TTU and TD access module documentation for more details.
NP_TTU_STARTUP_TIME	When using named pipes, the TTU creates the named pipe and then accepts data. So before spooling out data into the pipe, the KM waits the specified number of seconds after starting the TTU before spooling data into pipe. The time depends on utility used, platform, CPU workload, network connectivity and so forth and need to be adapted to used systems. Please take into account that this time also needs to fit production environments.
NP_EXEC_ON_WINDOWS	When using named pipes, this options specifies, whether the agent running this interface is on Windows.
MULTILOAD_TPUMP_TYPE	Operation performed by load utility (only

KM Option	Option Description						
	<p>for MULTILoad and TPUMP)</p> <p>choose:</p> <p>INSERT (insert new data)</p> <p>UPSERT (insert new data and update existing)</p> <p>DELETE (delete existing data)</p> <p>WARNING: using fastload forces the KM to delete the content of the target table before fastloading the data in it.</p>						
COMPATIBLE	<p>This option affects the way the data is merged into the target table:</p> <table> <tr> <td>Values</td><td>Command</td></tr> <tr> <td>12</td><td>MERGE</td></tr> <tr> <td>6</td><td>INSERT, UPDATE</td></tr> </table> <p>Note: Only Teradata V12 provides full ansi-compliant merge statement.</p>	Values	Command	12	MERGE	6	INSERT, UPDATE
Values	Command						
12	MERGE						
6	INSERT, UPDATE						

Setting up Agents on Teradata environment

Where to Install the Agent(s) on Teradata environment?

A typical data warehouse implementation usually requires a single ODI agent in production. In a Teradata environment, the agent is usually installed on the host machine that is used to load data in the data warehouse. The agent requires a connection to source databases or files, and triggers the appropriate load utilities (fastload, multiload, tpump). The operating system of this host machine can be Unix, Linux, Windows or zOS provided that a Java Virtual Machine 1.4 is installed.

APPENDIX II: ADDITIONAL INFORMATION

Acronyms used in this book

3NF	3 rd Normal Form
API	Application Programming Interface
ASCII	American Standard Code for Information Interchange
CKM	Check Knowledge Module

DDL	Data Description Language
DML	Data Manipulation Language
DW	Data Warehouse
EBCDIC	Extended Binary-Coded Decimal Interchange Code
E-LT	Extract, Load and Transform
ETL	Extract, Transform and Load
GUI	Graphical User Interface
HTTP	Hypertext Transport Protocol
IKM	Integration Knowledge Module
IT	Information Technology
J2EE	Java 2 Enterprise Edition
JDBC	Java Database Connectivity
JKM	Journalizing Knowledge Module
JMS	Java Message Services
JNDI	Java Naming Directory Interface
JVM	Java Virtual Machine
KM	Knowledge Module
LDAP	Lightweight Directory Access Protocol
LKM	Loading Knowledge Module
MQ	Message Queuing
ODBC	Open Database Connectivity
ODS	Operational Data Store
OLTP	Online Transactional Processing
RDBMS	Relation Database Management System
RI	Referential Integrity
RKM	Reverse-engineering Knowledge Module
SOX	Sarbane-Oxley
SQL	Simple Query Language

URL	Unique Resource Locator
XML	Extended Markup Language



Oracle Data Integrator for Data Warehouses

May 2008

Author: ODI Product Management

Contributing Authors: ODI Development

Oracle Corporation

World Headquarters

500 Oracle Parkway

Redwood Shores, CA 94065

U.S.A.

Worldwide Inquiries:

Phone: +1.650.506.7000

Fax: +1.650.506.7200

oracle.com

Copyright © 2006, Oracle. All rights reserved.

This document is provided for information purposes only and the contents hereof are subject to change without notice.

This document is not warranted to be error-free, nor subject to any other warranties or conditions, whether expressed orally or implied in law, including implied warranties and conditions of merchantability or fitness for a particular purpose. We specifically disclaim any liability with respect to this document and no contractual obligations are formed either directly or indirectly by this document. This document may not be reproduced or transmitted in any form or by any means, electronic or mechanical, for any purpose, without our prior written permission. Oracle, JD Edwards, PeopleSoft, and Siebel are registered trademarks of Oracle Corporation and/or its affiliates. Other names may be trademarks of their respective owners.