Data Vault – Code-Generierung mit dem ODI

Markus Schneider
PRODATO Integration Technology GmbH
Nürnberg, Deutschland

Schlüsselworte

Data Vault, Code Generator, Oracle Data Integrator, Knowledge Modules, odiRef Substitution API, ODI SDK Public API, Data Modeling, Data Warehousing, Java

Einleitung

Der hohe Grad der Standardisierung, die vergleichsweise einfache und einheitliche Importlogik (Insert-Only) und die klare Trennung der Schichten ermöglicht eine weitgehend automatische Erzeugung der Importlogik für ein Data Vault-Modell (Raw Vault und Teile der Business Vault).

Obwohl der Oracle Data Integrator (ODI) nicht primär als Modellierungstool für Data Vault konzipiert wurde, ist er als Zielsystem des Code-Generators besonders geeignet. Seine über die Substitution API flexibel anpassbaren Knowledge Module reduzieren die Komplexität der generierten Mappings und sorgen für schnelle Erfolge bei der Entwicklung eines eigenen Code-Generators.

Der hier vorgestellte Generator wird im Rahmen von Kundenprojekten entwickelt, in denen sowohl komplett neue DWHs auf Data Vault Basis erstellt werden, als auch existierende dimensionale Warehouses um eine Data Vault-Zwischenschicht erweitert werden.

Ziel des Vortrags ist es zu zeigen, wie vergleichsweise einfach die Data Vault-Codegenerierung für den ODI funktioniert und welche Erfahrungen wir bei der Entwicklung gemacht haben.

Data Vault - Architektur

Die DV-Architektur orientiert sich an klassischen DWH-Architekturen (Inmon, Kimball). Das eigentliche Data Vault – Datenmodell liegt als Schicht zwischen dem Staging und den Data Marts.

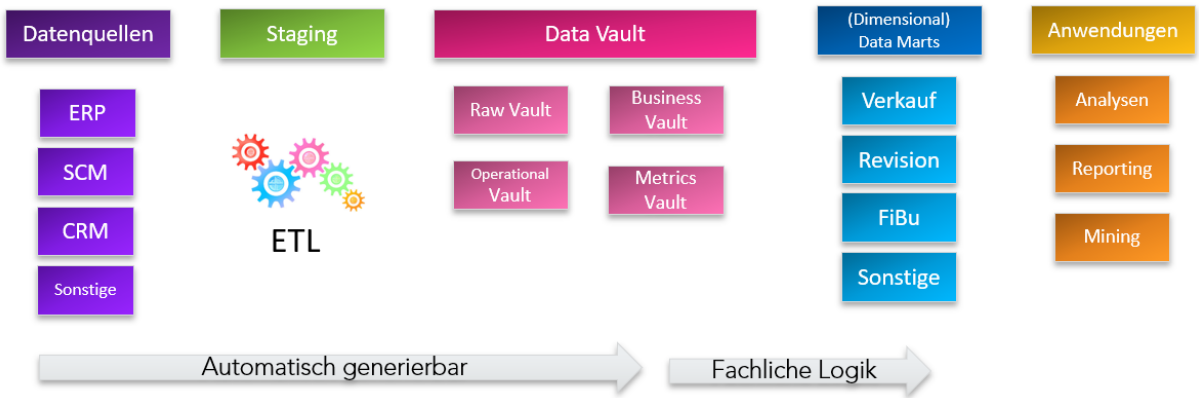

Abbildung 1: Data Vault - Architektur

Die Ladelogik zwischen Staging und Raw Vault kann vollständig automatisch generiert werden, zwischen Raw und Business Vault aber nur partiell, da individuelle Geschäftslogik nicht vollständig automatisierbar ist. Der Vortrag beschränkt sich daher auf die Raw Vault-Befüllung, mit einem kurzen Einblick die Business Vault-Strukturen Bridges und Point-in-Time-Tables, die ebenfalls automatisch erzeugt und befüllt werden können.

Data Vault – Datenmodell: Hubs, Links und Satellites

Ein Hub repräsentiert ein Geschäftsobjekt und enthält primär seine fachlichen Schlüsselattribute. Hubs enthalten keine deskriptiven Informationen, keine Abhängigkeiten durch Fremdschlüssel, keine Historie und damit auch keine zeitliche Gültigkeit.

Jedem Objekt wird eine Surrogat-ID zugewiesen, die das Objekt im Data Vault-Schema eindeutig identifiziert. Ab DV 2.0 handelt es sich dabei um einen Hash-Key, der aus dem fachlichen Schlüssel berechnet wird. Das reduziert die Abhängigkeiten beim Laden (keine Key-Lookups notwendig) und unterstützt eine heterogene und dabei konsistente Warehouse-Architektur (Kombination Datenbank + Big Data).

H_ARTICLE	
H_ARTICLE_HK	Surrogat-Schlüssel (Hash-Key)
ARTICLE_NUMBER	Fachlicher Schlüssel (Unique-Key Constraint)
LOAD_DTS	(Erst-) Ladezeitpunkt
RECORD_SOURCE	Quellsystem

Abbildung 2: Hub (Beispiel)

Außerdem sind noch zwei Metadaten enthalten: der Ladezeitpunkt des Datensatzes (LOAD_DTS), der später nicht mehr geändert wird und damit den Erstladezeitpunkt darstellt und das Quellsystem (RECORD_SOURCE) bzw. eine Job-ID, um die Datenherkunft nachvollziehen zu können.

Links repräsentieren Beziehungen, wie zum Beispiel klassische Hierarchien, Ereignisse oder Transaktionen. Dabei verbinden sie mindestens zwei Hubs miteinander.

L_ARTICLE_SEGMENT	
L_ART_SEG_HK	Surrogat-Schlüssel (HK)
H_ARTICLE_HK	Surrogat-Schlüssel (HK) des Artikels
H_SEGMENT_HK	Surrogat-Schlüssel (HK) des Segments
ARTICLE_NUMBER	Fachlicher Schlüssel des Kunden
SEGMENT_NUMBER	Fachlicher Schlüssel des Auftrags
LOAD_DTS	(Erst-) Ladezeitpunkt
RECORD_SOURCE	Quellsystem

Abbildung 3: Link (Beispiel)

Die fachlichen Schlüssel der verbundenen Hubs dienen zur Berechnung des Link Hash-Keys und werden seit DV 2.0 auch mit im Link gespeichert, um Auswertungen im Data Vault-Datenmodell zu

erleichtern. Auch wenn Links ebenfalls den Erst-Ladezeitpunkt einer Beziehung (LOAD_DTS) enthalten, so hat diese Beziehung selbst keinen zeitlichen Kontext.

Satelliten enthalten die deskriptiven Informationen des Geschäftsobjekts und sind dabei immer genau einem Hub oder einem Link zugeordnet. Umgekehrt können Hubs und Links beliebig viele Satelliten besitzen.

S_ARTICLE_MAIN	
H_ARTICLE_HK	Surrogat-Schlüssel (HK) des Kunden (PK-Constraint)
LOAD_DTS	Ladezeitpunkt (PK-Constraint)
LOAD_END_DTS	Gültigkeitsende
HASH_DIFF	Change Data Capture (HK)
NAME	Artikelbezeichnung
BRAND	Marke
RECORD_SOURCE	Quellsystem

Abbildung 4: Satellite (Beispiel)

Der Eintrag im Satelliten hat keinen eigenen Surrogatschlüssel, er wird über die ID des Hubs bzw. Links und dem Ladezeitpunkt identifiziert. Bei jeder Änderung wird ein neuer Eintrag geschrieben, die Daten sind historisiert.

Bei Satelliten werden Hash Keys auch verwendet, um schnell eine Änderung feststellen zu können. Dazu wird ein Change Data Capture Hash-Key (HASH_DIFF) aus den deskriptiven Feldern errechnet und gespeichert.

Code-Generierung – Lohnt sich das?

Ja, sie lohnt sich nicht nur, sie ist bei Data Vault notwendig. Durch die Aufspaltung in Hub, Links und Satellites multipliziert sich die Anzahl der Datenstrukturen und der damit verbundenen Ladeprozesse. Das ist mit manueller Modellierung nur schwer zu bewältigen.

Code-Generierung dient primär der Entlastung der Entwickler von repetitiven Tätigkeiten und schafft die Voraussetzung für schnelle und agile Projekte. Darüber hinaus ist generierter Code standardisiert, er hat eine einheitliche Qualität.

Die Nachteile von automatisch generiertem Code, dass er z.B. überkomplex, schlecht wartbar und inperformant ist, fallen im Data Vault – Bereich weniger stark ins Gewicht. Das liegt am hohen Grad der Standardisierung, die eine sehr einfache und damit gut optimierbare Ladelogik ermöglicht.

Code-Generierung – Ist der ODI dafür das richtige Tool?

Bei der Abwägung zwischen den beiden Möglichkeiten: Einen auf Data Vault spezialisierten (SQL-) Code-Generator zuzukaufen oder einen eigenen zu entwickeln, der die Ladelogik Oracle Data Integrator erzeugt, sind viele Faktoren zu berücksichtigen.

In unserem Projekt war für uns die Flexibilität entscheidend, die nur ein eigener Generator bieten kann. Da es sich um ein Migrationsprojekt handelt, bei dem zunächst Teile eines klassischen

dimensionalen Warehouses schrittweise um eine Data Vault-Zwischenschicht erweitert werden, kommen hier einige Besonderheiten hinzu. Wir erzeugen zum Beispiel ein Kombi-Modell aus DV 1.0 und DV 2.0, das heißt jeder Hub bzw. Link besitzt sowohl einen Hash-Key, als auch eine numerische ID aus dem bestehenden Warehouse zur Identifikation. Das ermöglicht ein schrittweises Release, da die numerischen IDs der Referenzen auf Dimensionen in den großen Faktentabellen nicht sofort auf Hash-Keys umgestellt werden müssen.

Der zweite große Vorteil ist der verhältnismäßig kleine Entwicklungsaufwand: eine erste Version des Generators, ohne Komfort-Features, ist schnell funktionsfähig. Das liegt zum einen an den einfachen Data Vault - Lade-Algorithmen und zum anderen an den ausgereiften Schnittstellen des Oracle Data Integrators.

Mit seiner Substitution API ermöglicht er die Entwicklung spezialisierter Integration Knowledge Module zum Befüllen der Hubs, Links und Satellites. Alle Aufgaben, die bereits von diesen Modulen erledigt werden, müssen auch nicht im eigentlichen Code-Generator implementiert werden. Die Folge sind strukturell einfache Mappings, die nur aus Quelle und Ziel bestehen. Diese Mappings werden über die ODI SDK Public API erzeugt, die jede Aktion im ODI-Studio abbilden kann.

Darüber hinaus vermeidet dieser Ansatz Medienbrüche: Die komplette Ladelogik befindet sich im Oracle Data Integrator; spezialisierte Generatoren erzeugen dagegen oft nur SQL. Es muss kein neues Tool gekauft werden, es sind keine neuen Schulungen notwendig und alle ODI-Mechanismen wie Logging, Fehlerbehandlung und Monitoring sind bereits abgedeckt.

Code-Generator: Meta-Modell

Der Code-Generator erzeugt DDL-Statements zum Anlegen und Ändern der Hub-, Link- und Satellite-Tabellen. Außerdem werden die Tabellen-Objekte im ODI DataStore und die Mappings mit ihrer Ladelogik generiert.

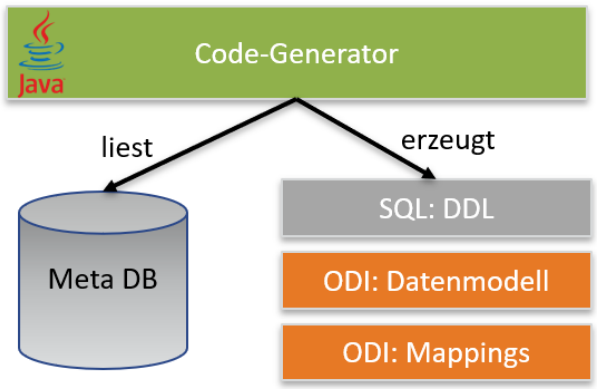

Abbildung 5: Code-Generator

Das Metamodell ist die Basis auf deren Grundlage der Generator Code erzeugt. Es gibt viele Möglichkeiten ein Metamodell zu beschreiben und abzulegen. Wir haben uns entschieden, ein relationales Datenschema anzulegen, das wir bei neuen Anforderungen an den Generator erweitern.

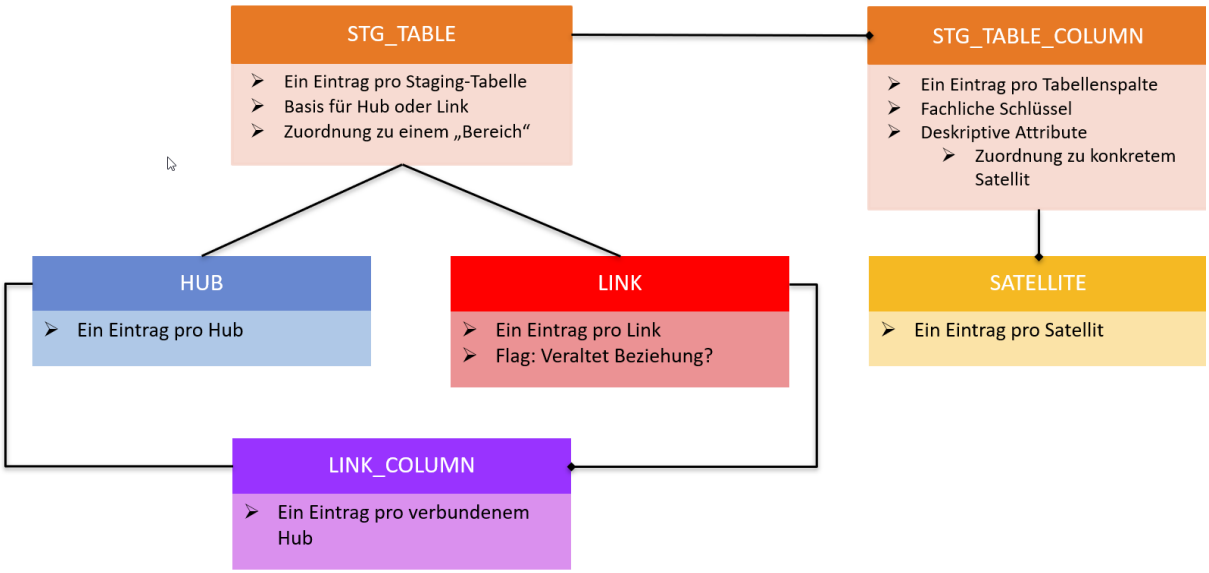

Abbildung 6: Code-Generator - Metamodell

Folgende Fragen des Code-Generators muss das Modell mindestens beantworten:

- Welche Datenquellen in Form von Staging-Tabellen müssen angebunden werden?
- Welche Hubs, Links und Satellites müssen als Zieltabellen angelegt werden?
- Wie sieht die Zuordnung zwischen Staging-Tabellen und Data Vault-Tabellen aus?
- Welche Hubs verbindet ein Link?
- Welches Feld der Staging-Tabelle ist Teil eines fachlichen Schlüssels und welches ist eine deskriptive Information?

Der Generator hat aktuell noch folgende Einschränkungen:

- Eine Staging-Tabelle ist Quelle für genau einen Hub oder Link (feste 1:1-Beziehung)
- Point-In-Time Tables und Bridges werden nicht erzeugt

Code-Generator – Algorithmus zum Beladen eines Hubs (bzw. Links)

Der Algorithmus zum Befüllen eines Hubs ist trivial: Nimm einen Eintrag aus der Staging-Tabelle, identifiziere den fachlichen Schlüssel und schaue nach ob der Eintrag bereits im Hub vorhanden ist. Fall ja, dann mache nichts und falls nein, dann füge ihn hinzu.

Für Links ist der Algorithmus identisch: Es wird geprüft, ob eine Beziehung bereits im Link vorhanden ist. Aus diesem Grund gibt es aus Sicht des Code-Generators keinen großen Unterschied zwischen Hubs und Links, die können fast gleich implementiert werden.

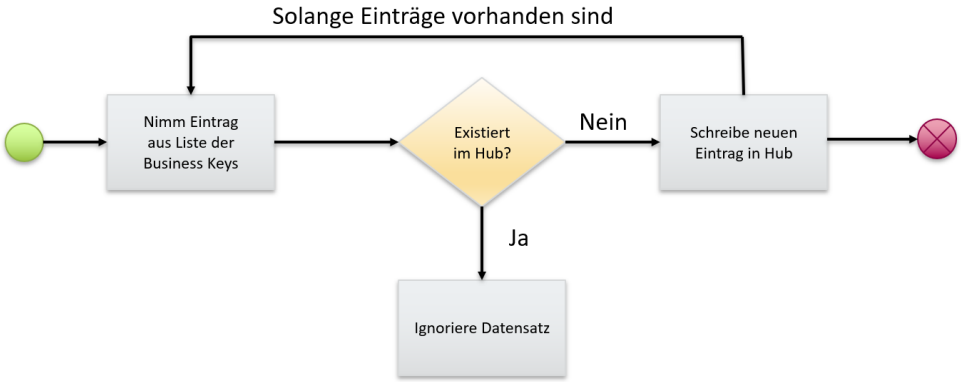

Abbildung 7: Algorithmus zur Hub-Befüllung

Der Algorithmus lässt sich in einem eigenen Integration Knowledge Module (IKM) zur Hub-Befüllung vollständig abbilden. Wir verwenden die User Defined Markers (Flags Udx), um Tabellenspalten zu markieren: UD1 ist der Hash-Key des Hubs und UD2 markiert alle fachlichen Schlüsselattribute.

Der erste rot markierte Block im Abbildung 8 berechnet den Hash-Key des Hubs mit Hilfe der *dbms_crypto.hash*-Funktion der Datenbank. Alle mit UD2 markierten Felder (= fachliche Schlüssel) werden konkateniert und als Parameter der Hash-Funktion übergeben. Diese Berechnung erfolgt nur für das UD1-markierte Feld (=Hash-Key ID).

```
insert /*+ <%=odiRef.getOption("HINT_INSERT_TARGET")%> */ into  <%=odiRef.getTable("L","TARG_NAME","A")%>
...
select  <%=odiRef.getPop("DISTINCT_ROWS")%>

  <%=odiRef.getColList(i,"", odiRef.getColList(
    "dbms_crypto.hash(src => utl_i18n.string_to_raw(data =>  ", "[EXPRESSION]", " || '|' || ", "), typ => 2)",
    "(UD2)") + " [COL_NAME]", ",\n\t", "", "(UD1)")%>

  <%=odiRef.getColList(i,",", "[EXPRESSION] [COL_NAME]", ",\n\t", "", "(UD2)")%>
from    <%=odiRef.getFrom(i)%>
...
) SRC

left outer join <%=odiRef.getTable("L","TARG_NAME","A")%> TRG
    on <%=odiRef.getColList("", "SRC.[COL_NAME] = TRG.[COL_NAME]", " AND \n\t", "", "UD1")%>
where <%=odiRef.getColList("", "TRG.[COL_NAME] IS NULL", " AND \n\t", "", "UD1")%>
```

Abbildung 8: IKM zur Hub-Befüllung (Auszug)

Der untere rot umrahmte Block prüft, ob die Entität im Hub bereits enthalten ist. Das ist mit einem Left-Outer-Join über den Hash-Key des Hubs gelöst. Das vom Generator erzeugte Mapping besteht nur aus Quelle und Ziel:

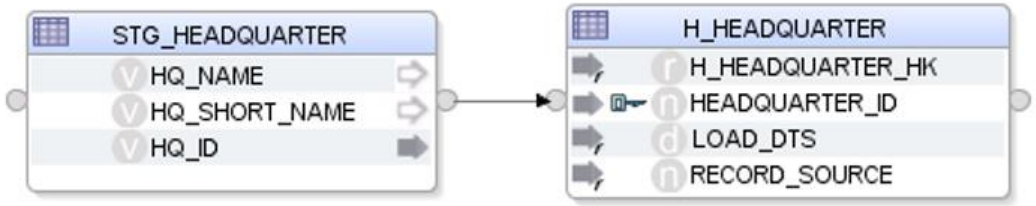

Abbildung 9: Mapping zur Hub-Beladung (Beispiel)

Code-Generator – Algorithmus zum Beladen eines Satellites

Die Satelliten-Befüllung ist minimal komplexer: Es muss der aktuellste Eintrag im Satelliten als Vergleichspartner ermittelt werden und der Vergleich erfolgt über den CDC-Hash-Key, der aus den deskriptiven Feldern der Staging-Tabelle berechnet wird.

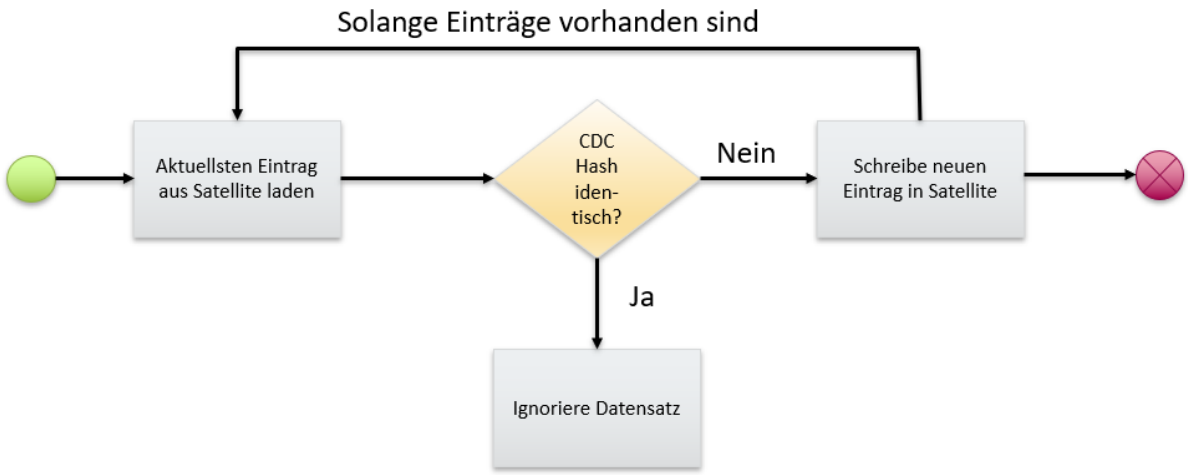

Abbildung 10: Algorithmus zur Satelliten-Befüllung

Falls ein Load-End-Date, also das Gültigkeitsende des Datensatzes, notwendig ist, dann muss der Algorithmus noch um ein Update des Vorgängerdatensatzes erweitert werden.

UDx	Beschreibung
UD1	CDC Hash-Key
UD2	Deskriptive Attribute
UD3	Hub Hash-Key
UD4	Hub Surrogate-ID (DV 1.0)
UD5	Load Date
UD6	Load End Date

Abbildung 11: Übersicht der Marker zur Satelliten-Befüllung

Von zentraler Bedeutung ist der CDC-Hash-Key (UD1) der aus den deskriptiven Informationen (UD2) berechnet wird. Das geschieht im ersten roten Block in Abbildung 12. Im zweiten Block wird der Hash-Key des Hubs berechnet und im dritten Block wird geprüft, ob ein neuer Eintrag im Satelliten notwendig ist. In dieser Variante ohne Load End Date wird eine analytische Funktion (ROW_NUMBER) verwendet, um den aktuellsten Eintrag im Hub zu ermitteln. Der Vergleich erfolgt wieder per Outer Join über die Hub-ID und den CDC-Hash-Key.

```
insert /*+ <%=odiRef.getOption("HINT_INSERT_TARGET")%> */ into  <%=odiRef.getTable("L","TARG_NAME","A")%>
...
select  <%=odiRef.getPop("DISTINCT_ROWS")%>

        <%=odiRef.getColList(i,"",odiRef.getColList(
            "dbms_crypto.hash(src => utl_i18n.string_to_raw(data =>  ",
            "[TO_CHAR_FORMAT_PREFIX] [EXPRESSION] [TO_CHAR_FORMAT_SUFFIX]", " || '|' || ", "), typ => 2)",
            "(UD2)") + " [COL_NAME]", ",\n\t", "", "(INS and REW and UD1)")%>

        <%=odiRef.getColList(i,",","[EXPRESSION] [COL_NAME]", ",\n\t", "", "(UD2)")%>

        <%=odiRef.getColList(i,",","dbms_crypto.hash(src => utl_i18n.string_to_raw(data => [EXPRESSION]),
                                    typ => 2) [COL_NAME]", ",\n\t", "", "(UD3)")%>

        <%=odiRef.getColList(i,",","[EXPRESSION] [COL_NAME]", ",\n\t", "", "(UD4)")%>
from    <%=odiRef.getFrom(i)%>
...
) SRC
```

```
left outer join (
  SELECT
         <%=odiRef.getColList("", "TRG2.[COL_NAME]", ",\n", "\n", "(UD1 OR UD2 OR UD3 OR UD4)")%>
  FROM (
    SELECT
         <%=odiRef.getColList("", "TRG1.[COL_NAME]", ",\n", ",\n", "(UD1 OR UD2 OR UD3 OR UD4)")%>
         ROW_NUMBER() OVER (PARTITION BY <%=odiRef.getColList("", "TRG1.[COL_NAME]", ",", "", "(UD3 OR UD4)")%>
                            ORDER BY <%=odiRef.getColList("", "TRG1.[COL_NAME] DESC", ",", "", "(UD5)")%>) AS ROW_NR
    FROM <%=odiRef.getTable("L","TARG_NAME","A")%> TRG1
  ) TRG2
  WHERE TRG2.ROW_NR = 1
) TRG
ON <%=odiRef.getColList("", "SRC.[COL_NAME] = TRG.[COL_NAME]", " AND \n\t", "", "(UD1 OR UD3 OR UD4)")%>
where <%=odiRef.getColList("", "TRG.[COL_NAME] IS NULL", ",", "", "(UD1)")%>
```

Abbildung 12: IKM zur Satelliten-Befüllung (Auszug)

Das vom Code-Generator erzeugt Mapping besteht ebenfalls nur aus Quelle und Ziel:

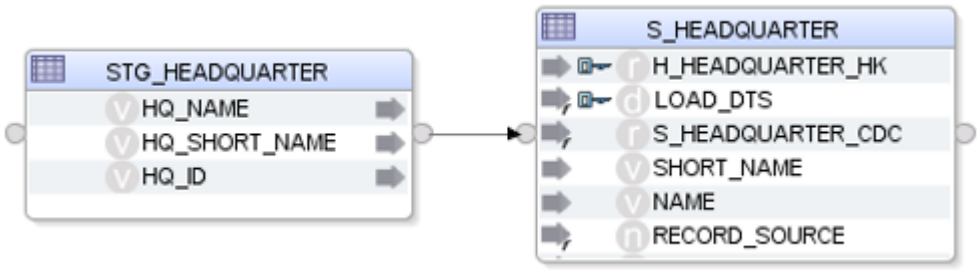

Abbildung 13: Mapping zur Satelliten-Beladung (Beispiel)

Fazit

Obwohl der ODI kein spezialisiertes Data Vault-Modellierungstool ist, kann er durch seine flexibel anpassbaren Knowledge Module dazu gemacht werden. Der Modellierungsaufwand für die Lade-Mappings sinkt durch diesen Template-Mechanismen enorm.

Die ODI SDK Public API erlaubt es Code-Generatoren alle ODI-Strukturen, insbesondere das Datenmodell und die Mappings automatisiert zu erstellen. Damit kann der Raw Vault automatisch erzeugt und beladen werden.

Kontaktadresse:
Markus Schneider
PRODATO Integration Technology GmbH
Herderstraße 5-9
D-90427 Nürnberg

Telefon: +49 (0) 911-994 730 54
Mobile: +49 (0) 151-269 060 42
E-Mail markus.schneider@prodato.de
Internet: www.prodato.de