Die generierte Zeitmaschine – Historisierung auf Knopfdruck

Dani Schnider
Trivadis AG
Zürich/Glattbrugg, Schweiz

Schlüsselworte:

Historisierung, Versionierung, Revisionsfähigkeit, Data Warehouse, Data Store, View Layer, ETL, Oracle Context, Oracle Warehouse Builder, OMB*Plus, PL/SQL, Generator

Einleitung

Eine typische Anforderung an ein Data Warehouse ist die lückenlose Historisierung der Daten basierend auf einem oder mehreren Quellsystemen. Dies ist die Grundvoraussetzung, um die Nachvollziehbarkeit von Auswertungen sicherstellen zu können. Ein Anwender, der einen Report im März dieses Jahres ausgeführt hat und nun denselben Report mit den gleichen Parametern nochmals ausführt, erwartet identische Resultate. Und zwar auch dann, wenn die Basisdaten im Quellsystem seither geändert oder sogar gelöscht wurden.

Diese Nachvollziehbarkeit von Datenänderungen ist nicht nur notwendig für die Reproduzierbarkeit von Auswertungen, sondern oft auch gesetzlich vorgeschrieben. In vielen Betrieben kann dadurch die Revisionsfähigkeit eines Systems gewährleistet werden.

Die Historisierung von Daten ist weitgehend unabhängig von fachlichen Anforderungen. Zwar muss – wie in jedem guten Data Warehouse – basierend auf den Anforderungen festgelegt werden, welche Daten aus den Quellsystemen ins Data Warehouse übernommen werden sollen. Steht der Umfang der Daten fest, so erfolgt die Historisierung immer nach dem gleichen Muster.

Was liegt also näher, als die benötigten Datenstrukturen und die zugehörigen Ladeprozesse automatisch zu generieren, statt mit viel Aufwand manuell zu implementieren?

Data Store: Historisierung der Quelldaten

Unsere Anforderung ist, dass alle Datenänderungen, die in einem Quellsystem durchgeführt werden, im Data Warehouse abgespeichert und zu jedem späteren Zeitpunkt abgefragt werden können. Dazu wird ein Data Store – eine historisierte Datenablage – eingeführt. Der Data Store ist ein Abbild der Daten aus den Quellsystemen. Damit alle relevanten Informationen lückenlos historisiert werden können, werden die Daten versioniert. Jede Änderung eines oder mehrerer Attribute eines Datensatzes führt zu einer neuen Version des Datensatzes. Im Data Store wird nicht der ursprüngliche Datensatz überschrieben, sondern ein neuer Eintrag eingefügt. Jeder Datensatz in diesem historisierten Datenmodell wird mit einem Gültigkeitsintervall ergänzt, definiert durch Anfangs- und Endzeitpunkt. Durch die Versionierungslogik wird sichergestellt, dass zu jedem Zeitpunkt immer genau eine Version eines Datensatzes gültig ist.

Schauen wir ein konkretes Beispiel an: Am 10. Juli 2009 wurde im Quellsystem eine neue Kundin eingegeben: Heidi Huber, Kundennummer 4711, geboren am 10. März 1984. Am 23. März 2010 heiratet Heidi Huber und heißt nun Heidi Müller-Huber. Dies führt zu einer neuer Version im Data Store, gültig ab dem Änderungszeitpunkt bis „unendlich". Um die Abfragen zu vereinfachen, wird der Endzeitpunkt „unendlich" durch das Datum 31. Dezember 9999 abgebildet. Die bisher gültige Version ist bis kurz vor der durchgeführten Änderung gültig. Entsprechend wir der Endzeitpunkt dieser Version auf eine Sekunde vor der Änderung gesetzt.

KUNDEN_NR	GUELTIG_VON	GUELTIG_BIS	NACHNAME	VORNAME	GEBURTSDATUM
4711	10.07.2009 16:13:06	23.03.2010 10:48:24	Huber	Heidi	10.03.1984
4711	23.03.2010 10:48:25	31.12.9999	Müller-Huber	Heidi	10.03.1984

Abb. 1: Versionierung von Daten

Solange nur eine Tabelle betrachtet wird, ist das Prinzip sehr einfach. Doch wie funktioniert die Historisierung von Fremdschlüsselbeziehungen zwischen Tabellen? Auf welche Version eines Datensatzes der referenzierten Tabelle soll verwiesen werden? Die Versionen in den verschiedenen Tabellen werden zu unterschiedlichen Zeitpunkten geschrieben. Um Beziehungen zwischen Tabellen trotzdem fehlerfrei und lückenlos abspeichern zu können, wählen wir folgenden Ansatz: Für jede Quelltabelle werden im Data Store zwei Tabellen erstellt, eine *Objekt-Tabelle* und eine *Versions-Tabelle*. Die Objekt-Tabelle enthält einen Eintrag pro Datensatz (Objekt) des Quellsystems. Darin werden der Primary Key des Quellsystems oder ein fachlicher Schlüssel sowie alle unveränderbaren (statischen) Attribute gespeichert. Statische Attribute können sich im Laufe der Zeit nicht ändern und müssen somit nicht versioniert werden. Alle anderen Attribute sind veränderbar und werden als dynamische Attribute bezeichnet. Sie werden zusammen mit dem Gültigkeitsintervall in der Versions-Tabelle gespeichert. Als Primary Key wird für jede Tabelle ein künstlicher Schlüssel, die Objekt-Id bzw. die Version-Id eingeführt. Da sich jede Version auf ein Objekt bezieht, wird in der Versions-Tabelle auch die Objekt-Id gespeichert und eine Fremdschlüsselbeziehung auf die Objekt-Tabelle implementiert.

Wichtig ist, dass der Aufbau für alle Tabellen gleich ist. Für jede Quelltabelle gibt es eine Objekt-Tabelle mit den statischen und eine Versions-Tabelle mit den dynamischen Attributen. Das ist deshalb zentral, weil dieser einheitliche Aufbau Voraussetzung dafür ist, dass die Tabellen und die zugehörigen ETL-Prozesse später generiert werden können.

Für unser Beispiel wird in der Objekt-Tabelle neben der Kundennummer auch das Geburtsdatum gespeichert, da es sich dabei um ein statisches Attribut handelt. Nachname und Vorname sind dynamische Attribute und somit Bestandteil der Versions-Tabelle.

Objekt-Tabelle		
OBJEKT_ID	KUNDEN_NR	GEBURTSDATUM
21	4711	10.03.1984

Versions-Tabelle					
VERSION_ID	OBJEKT_ID	GUELTIG_VON	GUELTIG_BIS	NACHNAME	VORNAME
101	21	10.07.2009 16:13:06	23.03.2010 10:48:24	Huber	Heidi
102	21	23.03.2010 10:48:25	31.12.9999	Müller-Huber	Heidi

Abb. 2: Prinzip von Objekt- und Versions-Tabelle

Durch die Aufteilung in Objekt- und Versions-Tabelle können nun auch Fremdschlüsselbeziehungen problemlos abgebildet werden. Da ein Fremdschlüssel ein Objekt referenziert und nicht eine zu einem bestimmten Zeitpunkt gültige Version, verweisen die Fremdschlüssel immer auf die Objekt-Tabellen. Ob der Fremdschlüssel in der Objekt- oder in der Versions-Tabelle gespeichert wird, hängt davon ab, ob die Fremdschlüsselbeziehung statisch oder dynamisch ist. Auch dazu wieder ein Beispiel: Jeder Kunde hat eine Adresse, an der gleichen Adresse können mehrere Kunden wohnen. Ein Kunde kann ein oder mehrere Konti besitzen, und jedes Konto gehört genau einem Kunden. Das Datenmodell des Quellsystems besteht somit aus drei Tabellen und zwei Fremdschlüsselbeziehungen.

Abb. 3: Datenmodell des Quellsystems

Im historisierten Data Store werden gemäß obiger Definition drei Paare von Objekt- und Versions-Tabellen erstellt. Statische Attribute werden in den Objekt-Tabellen gespeichert, dynamische Attribute in den Versions-Tabellen. Genau gleich verhält es sich mit den Fremdschlüsselattributen. Ein Kunde kann seine Adresse im Laufe der Zeit mehrmals ändern. Deshalb wird der Fremdschlüssel auf die Adresse in der Versions-Tabelle des Kunden gespeichert. Anders sieht es mit dem Konto aus. Ein Konto gehört immer zum gleichen Kunden und kann nicht auf einen anderen Kunden übertragen werden. Deshalb ist diese Beziehung statisch, und das entsprechende Fremdschlüsselattribut wird in der Objekt-Tabelle des Kontos abgelegt.

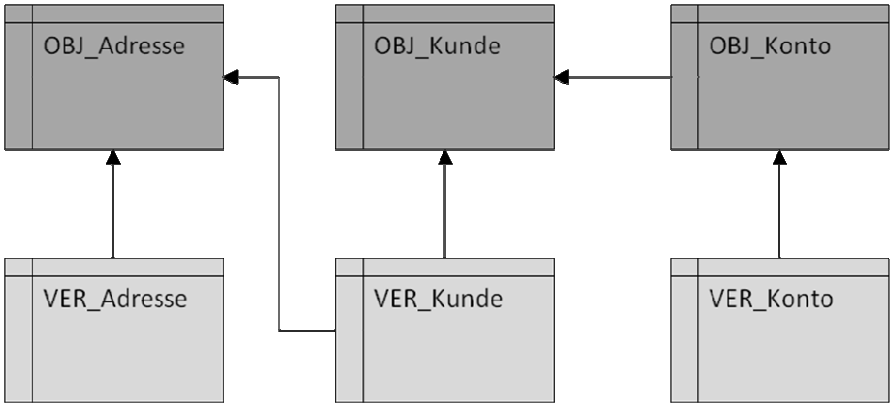

Abb. 4: Historisiertes Datenmodell im Data Store

ETL: Der Weg vom Quellsystem in den Data Store

Um alle Datenänderungen im Data Store lückenlos zu historisieren, müssen für jede Datenmanipulation im Quellsystem entsprechende ETL-Operationen im Data Store durchgeführt werden. Wann und wie häufig diese Datenübernahme in den Data Store erfolgt, spielt für die Historisierung keine Rolle, wie wir gleich sehen werden.

Datenmanipulation im Quellsystem	ETL-Operationen im Data Store
Einfügen eines Datensatzes	Erstellen eines neuen Objekts Erstellen einer ersten Version zu diesem Objekt, gültig ab Erstellungszeitpunkt

Ändern eines Datensatzes	Erstellen einer neuen Version für das existierende Objekt, gültig ab Änderungszeitpunkt Abschließen der Gültigkeitsperiode der bisher aktuellen Version auf den Änderungszeitpunkt
Löschen eines Datensatzes	Abschließen der Gültigkeitsperiode der bisher aktuellen (d.h. der letzten) Version auf den Änderungszeitpunkt

Abb. 5: Datenmanipulationen im Quellsystem und im Data Store

Was heißt nun genau „lückenlose Historisierung“? Die Anforderungen sind nicht in jedem Data Warehouse gleich definiert. Oft genügt es, wenn einmal täglich, wöchentlich oder monatlich der aktuelle Stand der geänderten Daten ins DWH geladen wird. Für ein revisionsfähiges System ist es jedoch notwendig, dass alle Änderungen festgehalten werden – also auch alle Zwischenstände, die seit der letzten Datenlieferung durchgeführt wurden.

Nehmen wir an, ein Quellsystem liefert jede Nacht die Änderungen des laufenden Tages an das DWH. Am 23. März morgens um 10 Uhr wird ein neuer Kunde erfasst. Um 11 Uhr wird seine Adresse ergänzt, und um 14 Uhr wird die Adresse korrigiert. Welcher Stand der Daten soll nun um Mitternacht ans DWH übermittelt werden? Um eine lückenlose Historisierung zu garantieren, lautet die Antwort: Alle Stände, d.h. jede durchgeführte Änderung.

Es gibt verschiedene Möglichkeiten, diese Anforderung zu implementieren. So kann Oracle Change Data Capture (CDC) verwendet werden, oder auf dem Quellsystem wird für jede relevante Tabelle ein DML-Trigger definiert, welcher die Änderungen in eine Journaltabelle schreibt. Im ersten Fall dient die CDC Subscriber View als Quelle für die ETL-Prozesse, im zweiten Fall die Journaltabelle. Bei beiden Verfahren werden die Datenmanipulationen im Quellsystem zwischengespeichert – entweder in der CDC Change Table oder in der Journaltabelle – und können zu jedem späteren Zeitpunkt vom ETL-Prozess „abgeholt“ und in den Data Store geschrieben werden. Durch eine geeignete Delta-Logik kann sichergestellt werden, dass jede Änderung nur einmal übertragen wird und dass die Änderungen lückenlos übermittelt werden. Im Falle von CDC wird dies durch die Subscriber View gewährleistet.

Nehmen wir an, alle Datenmanipulationen werden vom Quellsystem über DML-Triggers in Journaltabellen geschrieben. Für jede Quelltabelle gibt es eine zugehörige Journaltabelle, in welcher alle Datenmanipulationen chronologisch gespeichert werden. Neben allen Attributen der Quelltabelle enthält die Journaltabelle eine Sequenznummer (ID), um die Reihenfolge zu definieren, den Zeitpunkt, wann die Änderung durchgeführt wurde, und den Typ der Operation.

ID	DML-Zeitpunkt	DML-Operation	Kunden_Nr	Nachname	Vorname	Geburtsdatum	Adr_Nr
423	23.03.2010 10:17:12	INSERT	9317	Fröhlich	Peter	16.04.1959	
424	23.03.2010 10:48:25	UPDATE	4711	Müller-Huber	Heidi	10.03.1984	
425	23.03.2010 11:05:21	UPDATE	9317	Fröhlich	Peter	16.04.1959	5903
426	23.03.2010 13:54:47	DELETE	8207				
427	23.03.2010 14:07:58	UPDATE	1410	Tell	Wilhelm	01.08.1291	7659
428	23.03.2010 14:28:02	UPDATE	9317	Fröhlich	Peter	16.04.1959	6111

Abb. 6: Aufbau einer Journaltabelle

Der ETL-Prozess hat nun die Aufgabe, den Inhalt der Journaltabelle in den Data Store überzuführen. Dazu wird das ETL-Tool Oracle Warehouse Builder (OWB) verwendet. Um die einzelnen OWB-Mappings möglichst einfach zu halten, wird der ETL-Ablauf in mehrere Teilschritte unterteilt. Ein *Stage-Mapping* liest die Daten aus der Journaltabelle und speichert sie in der Staging Area des Data Warehouses. In diesem Schritt wird ein Deltaabgleich implementiert, der sicherstellt, dass nur jene Einträge aus der Journaltabelle übernommen werden, die nicht schon in einem vorhergehenden ETL-Lauf geladen wurden. Außerdem werden hier die Gültigkeitsintervalle pro Version ermittelt. Das nachfolgende *Objekt-Mapping* hat die Aufgabe, für neue Datensätze, die im Quellsystem erfasst wurden, einen Eintrag in die Objekt-Tabelle zu schreiben. Dazu wird ein Abgleich zwischen Stage-Tabelle und Objekt-Tabelle durchgeführt. Das *Versions-Mapping* schließlich schreibt für jeden im Quellsystem eingefügten oder geänderten Datensatz eine neue Version in die Versions-Tabelle und setzt für jeden geänderten oder gelöschten Datensatz den Endzeitpunkt der Vorgängerversion des entsprechenden Objekts.

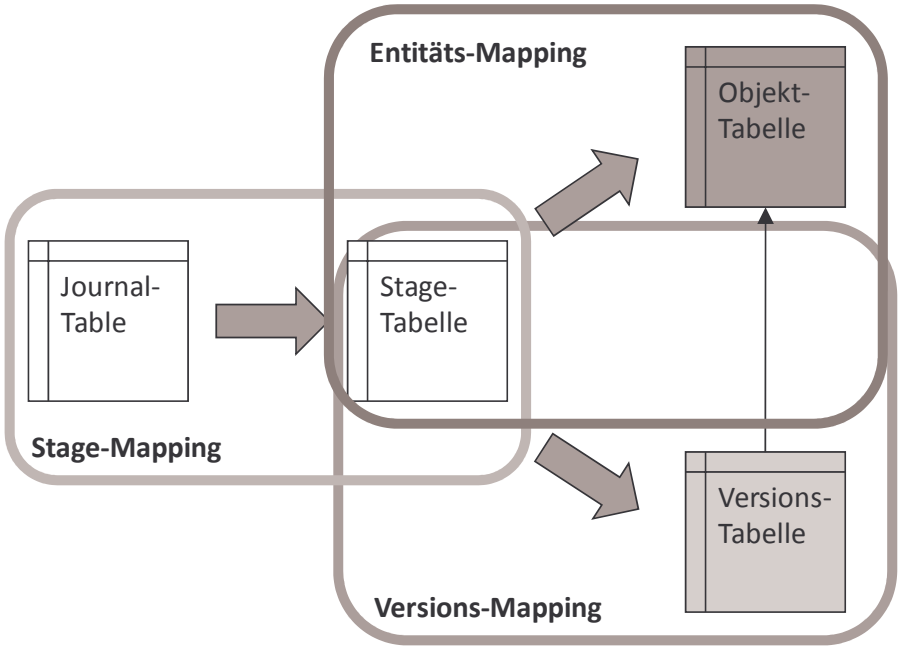

Abb. 7: ETL-Ablauf von der Journaltabelle in den Data Store

Diese drei Mappings müssen für jede Quelltabelle erstellt werden. Der logische Aufbau der Mappings ist zwar immer gleich, aber die Quell- und Zieltabellen sowie die Attribute und Datentypen sind in jedem Mapping unterschiedlich.

View Layer: Fenster in die Vergangenheit

Nun sind die Daten vollständig und lückenlos historisiert im Data Store gespeichert. Doch wie finden wir sie dort wieder? Um den Datenbestand zu einem bestimmten Zeitpunkt in der Vergangenheit zu ermitteln, müssten entsprechende Abfragen auf die Objekt- und Versions-Tabellen durchgeführt werden, welche die jeweils gültige Version zum entsprechenden Zeitpunkt zurückgeben – und das für jedes Paar von Objekt- und Versions-Tabelle. Um solche Abfragen zu vereinfachen, wird ein View Layer zur Verfügung gestellt, der eine konsistente Sicht in die Vergangenheit sicherstellt.

Für jede Quelltabelle, also für je ein zusammengehörendes Paar von Objekt- und Versions-Tabelle, werden zwei Views erstellt. Die Stand-View ermittelt den Stand der Daten zu einem definierten Zeitpunkt in der Vergangenheit und stellt sicher, dass pro Objekt immer nur eine Version zurückgegeben wird, nämlich die, welche zum definierten Abfragezeitpunkt gültig war. Die Intervall-View gibt alle Versionen zurück, die innerhalb eines definierten Abfrageintervalls gültig sind. In beiden Fällen findet ein Join zwischen Objekt- und Versions-Tabelle statt, das heißt, die physische Speicherung im Data Store wird hinter dem View Layer „versteckt".

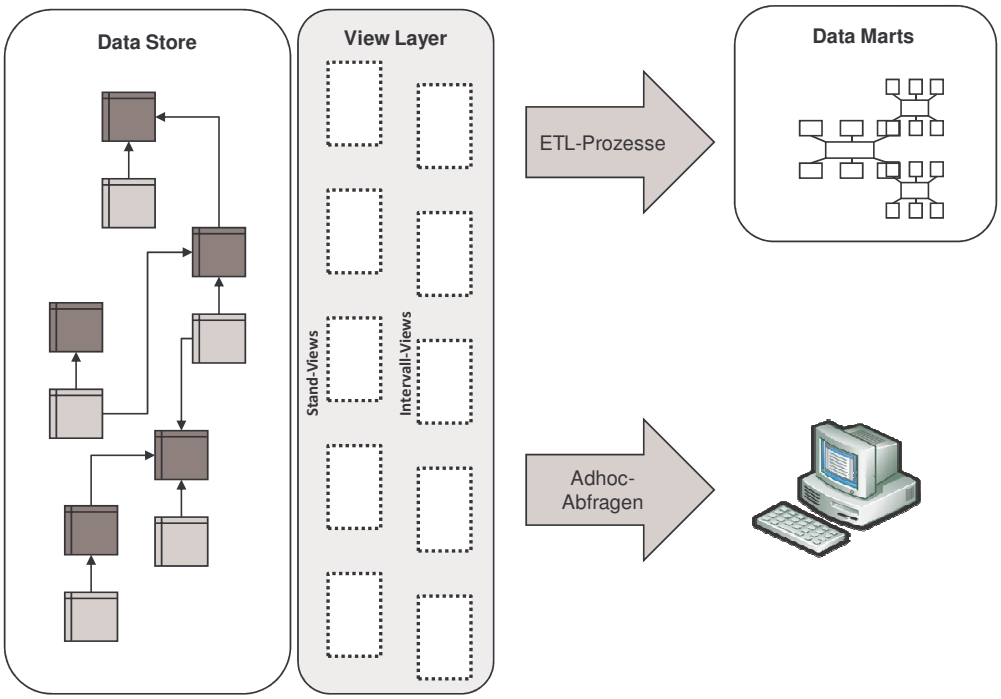

Abb. 8: Prinzip des View Layers

Der View Layer dient als Basis für die ETL-Prozesse, mit denen die Data Marts geladen werden. Soll zum Beispiel ein Data Mart einmal pro Monat mit dem Datenbestand per Ende des Monats geladen werden, wird der Abfragezeitpunkt im View Layer auf den letzten Tag des abgeschlossenen Monats gesetzt. Danach können über den View Layer die Daten gelesen werden, ohne dass sich die ETL-Prozesse um die Historisierung der Daten kümmern müssen. Die Stand-Views des View Layers zeigen die Daten so an, wie sie zum definierten Zeitpunkt ausgesehen haben.

Angenommen, der Data Mart wird um eine zusätzliche Kennzahl erweitert. Weil sich diese Kennzahl nicht aus den bestehenden Daten im Data Mart berechnen lässt, müssen zusätzliche Attribute aus dem Data Store geladen werden, und zwar rückwirkend für alle vergangenen Monate. Kein Problem! Nach der Erweiterung des Data Marts und der entsprechenden ETL-Prozesse werden die vergangenen Monate in den Data Mart nachgeladen. Weil die Daten im Data Store vollständig historisiert sind, kann jeder Stand in der Vergangenheit rekonstruiert werden. Dazu wird einfach der Abfragezeitpunkt im View Layer entsprechend gesetzt.

Auch für Adhoc-Abfragen auf den Data Store kann der View Layer verwendet werden. Abfragen auf die Stand-Views sind relativ einfach. Die Views können gleich wie die Tabellen im Quellsystem miteinander verknüpft werden, zeigen aber nicht zwingend den aktuellen Stand der Daten an, sondern

den Stand zum definierten Abfragezeitpunkt. Abfragen auf Intervall-Views sind einiges komplexer, wenn mehrere Views gejoined werden. Hierbei muss berücksichtigt werden, dass die Gültigkeitsintervalle der einzelnen Versionen überschneidend sein können und so durch den Join mehrere zusätzliche Zwischenversionen entstehen können.

Der Abfragezeitpunkt für die Stand-Views oder das Abfrageintervall für die Intervall-Views werden über einen Oracle Context gesetzt. Ein Context ist vergleichbar mit einer globalen Variablen, die innerhalb einer Session sichtbar ist. Er kann über ein PL/SQL-Package gesetzt und über die SQL-Funktion `SYS_CONTEXT` abgefragt werden. Der Context wird bei der Erstellung einem PL/SQL-Package zugeordnet, in welchem die Prozedur `dbms_session.set_context` aufgerufen werden muss. Für den View Layer wird ein Oracle Context und ein zugehöriges PL/SQL-Package implementiert.

```
CREATE OR REPLACE CONTEXT view_layer_context
USING view_layer_parameter;

CREATE OR REPLACE PACKAGE view_layer_parameter
IS
   PROCEDURE set_abfragezeitpunkt
      (p_abfrage_datum IN DATE DEFAULT NULL);
   PROCEDURE set_abfrageintervall
      (p_int_von_datum IN DATE DEFAULT NULL
      ,p_int_bis_datum IN DATE DEFAULT NULL);
END view_layer_parameter;
```

Auf diese Weise können nun parametrisierbare Views erstellt werden, die jeweils die aktuellen Werte aus dem Context lesen und für die Abfrage verwenden können. Die Benutzer wählen durch Aufruf der Prozedur `view_layer_parameter.set_abfragezeitpunkt` einen Abfragezeitpunkt und können danach auf die Stand-Views des View Layers zugreifen. Die Views liefern dann die jeweils gültige Version zu diesem Zeitpunkt zurück.

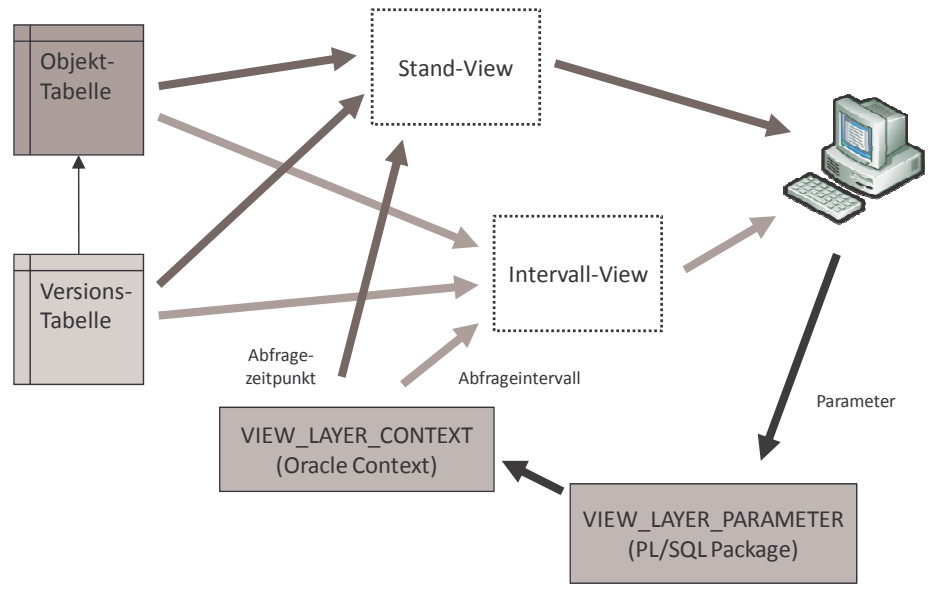

Abb. 9: Parametrisierbare View über Oracle Context

Da ein Context nur innerhalb der Session sichtbar ist, können gleichzeitig mehrere Benutzer (oder ETL-Prozesse) Abfragen zu unterschiedlichen Abfragezeitpunkten auf den View Layer machen. Um den Context in einem ETL-Prozess zu verwenden, kann er in einem Pre-Mapping Process in Oracle Warehouse Builder gesetzt werden.

Eine Stand-View sieht folgendermaßen aus:

```
CREATE OR REPLACE VIEW v_stand_kunde AS
SELECT obj.kunden_nr
     , ver.gueltig_von
     , ver.gueltig_bis
     , ver.nachname
     , ver.vorname
     , ver.geburtsdatum
  FROM obj_kunde obj
  JOIN ver_kunde ver ON (ver.objekt_id = obj.objekt_id)
WHERE SYS_CONTEXT('VIEW_LAYER_CONTEXT','ABFRAGEZEITPUNKT')
      BETWEEN ver.gueltig_von AND ver.gueltig_bis;
```

Für jede Quelltabelle muss eine solche Stand-View erstellt werden. Alle Stand-Views haben den gleichen Aufbau und bestehen aus einem einfachen Join zwischen Objekt- und Versions-Tabelle sowie einer Einschränkung nach dem Gültigkeitsintervall basierend auf dem definierten Context. Die Intervall-Views sehen fast gleich aus, haben aber eine andere WHERE-Bedingung:

```
WHERE ver.gueltig_von <= SYS_CONTEXT('VIEW_LAYER_CONTEXT','INTERVALL_BIS')
  AND ver.gueltig_bis => SYS_CONTEXT('VIEW_LAYER_CONTEXT','INTERVALL_VON')
```

TOGO: Automatische Generierung der OWB-Objekte

Für den Data Store müssen eine grosse Anzahl von Tabellen, OWB-Mappings und Views implementiert werden. Die meisten dieser Datenbankobjekte haben jedoch den gleichen Aufbau und unterscheiden sich nur bezüglich Namen und zugehörigen Attributen und Datentypen. Anstatt alle Tabellen, Mappings und Views von Hand zu erstellen, sollen sie automatisch generiert werden. Dies hat verschiedene Vorteile: Die mühsame und zeitaufwändige Fleissarbeit zum Implementieren vieler ähnlicher Tabellen, Mappings und Views entfällt. Die Gefahr von Flüchtigkeitsfehlern ist kleiner – die generierten Datenbankobjekte haben alle den gleichen Aufbau. Bei Änderungen oder Erweiterungen der Quellsystem-Tabellen müssen die Mappings und Views nicht manuell angepasst werden, sondern können neu generiert werden.

Um Datenbankobjekte generieren zu können, ist natürlich ein Grundaufwand notwendig, um entsprechende Generatoren zu implementieren. Für eine kleine Anzahl von Objekten lohnt sich dieser Aufwand kaum, doch je mehr gleichartige Objekte implementiert werden müssen, desto eher zahlt sich die Entwicklung eines entsprechenden Generators aus. Dies soll anhand einer einfachen Rechnung aufgezeigt werden: Für jede Tabelle eines Quellsystems werden im Data Store eine Objekt- und eine Versions-Tabelle erstellt. Für die ETL-Prozesse werden mehrere Mappings, Stage-Tabellen sowie eventuell Views oder External Tables benötigt. Der View Layer enthält je eine Stand- und eine Intervall-View. Im Rahmen eines Kundenprojekts wurde ein historisierter Data Store für ein Quellsystem mit 90 Tabellen entwickelt. Die Daten werden als Flat Files geliefert und über External

Tables und Stage-Tabellen in den Data Store geladen. Dazu mussten insgesamt 360 Tabellen, 270 Mappings und 180 Views erstellt werden. Diese Größenordnungen und die oben aufgeführten Gründe zeigen, dass es nicht zweckmässig ist, diese Tabellen, Mappings und Views von Hand zu erstellen. Mit Hilfe eines geeigneten Generators können diese Routinearbeiten vereinfacht und beschleunigt werden.

Neben der interaktiven Entwicklung im OWB-Client bietet der Oracle Warehouse Builder die Möglichkeit, über die spezielle Scriptsprache OMB*Plus alle Aktionen auszuführen, die im OWB-Client möglich sind. Häufig wird dies für einfache Aktionen wie das Setzen von Properties für mehrere Mappings verwendet. Es ist aber auch möglich, beispielsweise ein Mapping mit Hilfe von OMB*Plus-Befehlen zu implementieren. Normalerweise ist dies nicht sinnvoll, da der Aufwand für ein einzelnes Mapping viel höher ist als bei der manuellen Entwicklung im Mapping-Editor. Bei vielen gleichartigen Mappings ist es aber zweckmässig, die OMB*Plus-Befehle, die für jedes Mapping ähnlich sind, generieren zu lassen. OMB*Plus basiert auf der allgemeinen Scriptsprache *tcl* und wurde um Oracle-spezifische OMB*Plus-Befehle erweitert.

Basierend auf dem *Trivadis Object Generator for OWB* (TOGO) wurde ein *Data Store Generator* entwickelt, der basierend auf einem Definitionsfile alle benötigten OWB-Objekte generiert. Das Definitionsfile enthält alle für den Generator wesentlichen Informationen wie Tabellen- und Attributnamen, Datentypen, Primary und Foreign Keys, Historisierungstyp, etc. Bei Strukturänderungen des Quellsystems muss dieses Definitionsfile angepasst und der Data Store komplett oder auszugsweise neu generiert werden.

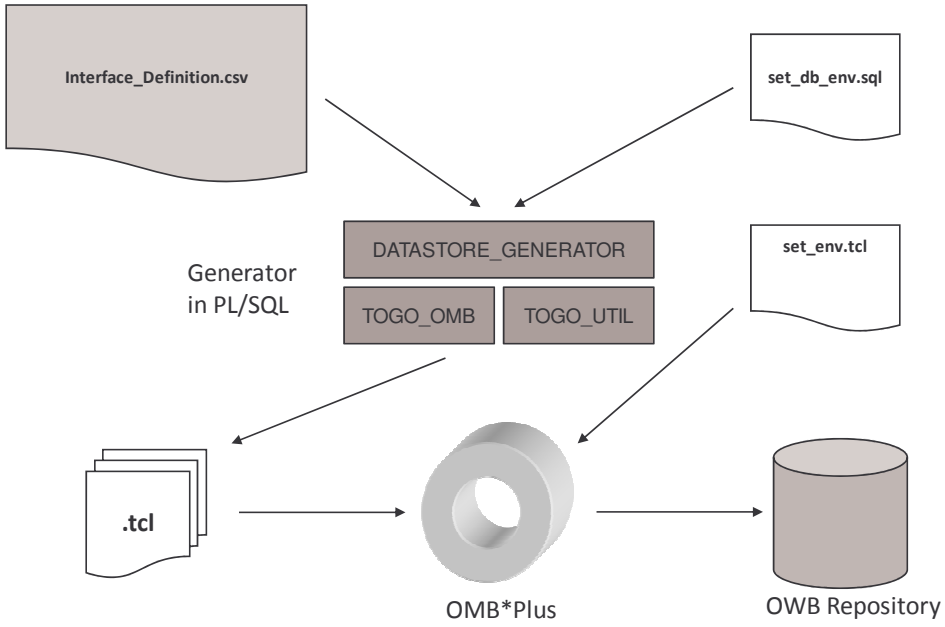

Abb. 10: Funktionsweise von TOGO und Data Store Generator

Der *Data Store Generator* besteht aus einem PL/SQL-Package `DATASTORE_GENERATOR`, welches die spezifischen OWB-Objekte für den Data Store generiert. Die allgemeine Funktionalität des OWB-Generators ist in zwei weiteren PL/SQL-Packages `TOGO_OMB` und `TOGO_UTIL` gekapselt. Der Generator liest das Definitionsfile ein und generiert mehrere *tcl*-Scripts, welche die OMB*Plus-Befehle zum Erstellen der entsprechenden OWB-Objekte enthalten. Pro Quelltabelle wird ein Script

generiert, das alle zu dieser Quelltabelle gehörenden Tabellen, Sequences, Mappings und Views erstellt. Ein zusätzliches *tcl*-Script erstellt die Foreign Key Constraints zwischen den Tabellen. Schliesslich wird ein Hauptscript `main.tcl` generiert, dass alle Scripts nacheinander aufruft. Wird dieses Script im OMB*Plus-Fenster von Oracle Warehouse Builder gestartet, macht der OWB das gleiche wie ein ETL-Entwickler – nur viel schneller und mit weniger Fehlern.

Fazit

Wiederkehrende Routinetätigkeiten sind nicht nur langweilig für die Entwickler, die sie ausführen müssen, sondern auch eine Fehlerquelle. Muss ein ETL-Entwickler Dutzende oder Hunderte von ähnlichen Mappings erstellen, lässt irgendwann die Konzentration nach und es schleichen sich Flüchtigkeitsfehler ein, die nur durch aufwändige Tests erkannt und behoben werden können. Werden die Datenbankobjekte und ETL-Mappings generiert, wird nicht nur die Entwicklungsdauer verkürzt, sondern auch der Testaufwand und die Fehlerquote minimiert, da der Ablauf immer gleich ist.

Die Anforderungen an einen historisierten Data Store sind für alle Tabellen gleich und nicht abhängig von fachlichen Regeln und Spezialfällen. Historisierung von Daten ist deshalb ein gutes Anwendungsbeispiel, wie die Implementation durch den Einsatz eines Generators vereinfacht und beschleunigt werden kann.

Trotzdem zwei Warnungen zum Schluss: Auch wenn mit Hilfe eines Generators die Anzahl der Fehler verringert werden kann, lassen sich Fehler nie ganz ausschließen, sei es durch falsche Schnittstellendefinitionen, unerwartete Datenlieferungen oder durch einen Programmierfehler im Generator selber. Auch generierte Software muss deshalb gewissenhaft getestet werden. Und schließlich darf der Aufwand für die Entwicklung des Generators nicht unterschätzt werden. Aufgrund von projektspezifischen Vorgaben wie Namenskonventionen oder der Verwendung bestehender Abläufe und Metadaten muss entweder ein entsprechender Generator für das geforderte Projektumfeld entwickelt werden, oder ein bestehender Generator muss entsprechend angepasst werden.

Kontaktadresse:

Dani Schnider
Trivadis AG
Europa-Strasse 5
CH-8152 Glattbrugg

Telefon: +41(0)44-808 70 20
Fax: +41(0)44-808 70 21
E-Mail dani.schnider@trivadis.com
Internet: www.trivadis.com