Das generische DWH: Weniger Code => weniger Kosten

Tobias Braunschober
b.telligent GmbH & Co. KG
Georg-Brauchle-Ring 52-54, 80992 München

Schlüsselworte
Business Case: generisches DWH, DWH als Kostenfaktor, DWH Architektur, PL/SQL-ELT-Prozess, generischer Code

Einleitung
Häufig werden im Business-Intelligence-Umfeld Aussagen wie die folgenden getätigt, um damit eigene Ziele zu verfolgen: Ein DWH ist so komplex, dass es nur durch gute Tools oder aufwändige Programmierung beherrschbar bleibt. Insbesondere bei bestehenden DWHs sind Änderungen kaum noch möglich. Das DWH ist ein riesiger Kostenfaktor.

Das muss nicht sein.

Während der Kostendruck auf die IT stets steigt, verlieren bestehende DWHs im Laufe der Zeit an Performance und die Kosten für die Aufrechterhaltung und Weiterentwicklung steigen stetig. Ein wesentlicher Grund dafür liegt an einer unnötig hohen Komplexität. Diese Komplexität steigt, je mehr Aufgaben im DWH dazu implementiert werden und je mehr Personen daran beteiligt sind.

Oft wird diese Komplexität durch eine unklare Architektur, die schiere Menge an Codezeilen oder die an viele Stellen im Quellcode verteilte Businesslogik erzeugt.

Ein solches Vorgehen führt zwangsläufig dazu, dass Lösungen (bestenfalls identisch) an mehreren Stellen programmiert sind. Häufig wird der einzelne Entwickler in einem solchen Umfeld unersetzlich, da nur noch er einen bestimmten Themenkomplex ausreichend überblickt um an den notwendigen Stellen Änderungen vornehmen zu können. Dies wirkt sich ebenso negativ auf das Testen als den anderen großen Kostenfaktor aus. Je komplexer die Implementierung ist, desto aufwändiger sind Tests, insbesondere wenn durch Seiteneffekte die Auswirkungen von Änderungen undurchsichtig sind.

Im Folgenden wird, anhand eines realen Kundenprojekts, gezeigt wie ein Ausweg aus diesen Kostenfallen implementiert werden kann.

Setup des Kundenprojekts
Im Projekt wurde die folgende Ausgangslage vorgefunden: Es gibt ein bereits bestehendes DWH (~350 GB) das über viele Jahre gewachsen ist und einige Umfirmierungen, Migrationen und Personalfluktuation erlebt hat. Die direkten Quellsysteme sind Oracle Datenbanken, das DWH selbst ebenso. Als Output-Schnittstellen dienen unterschiedliche Zielsystem, wie SAB Business Objects, Cubeware, MS Analysis Services, .net-Eigenentwicklungen und technischen Datenschnittstellen, die externe Abnehmer beliefern. Dieses bestehende DWH soll durch ein neues DWH 2.0 ebenfalls in Oracle ersetzt werden. Für die Entwicklung sollen keine ETL/ELT-Tools verwendet werden. Das DWH 2.0 soll in einem Jahr von einer Person zur Produktionsreife gebracht werden können.

Architektur
Um in der kurzen Zeit das DWH 2.0 Implementieren zu können, muss zuerst Einigkeit über eine klare Architektur hergestellt werden, die im Laufe des Projekts unumstößlich ist. Einem idealen DWH liegt

ein Mehr-Schichten-Modell zugrunde, was im Folgenden beschrieben wird. Fehler die in der Architekturphase gemacht werden sind im späteren Projektverlauf nahezu nicht mehr zu korrigieren.

Das Mehr-Schichten-Modell
Ein DWH besteht diesem Modell nach aus mehreren Schichten. Dabei lässt sich das Grundprinzip in vier Paradigmen zusammenfassen: Jede der Schichten erfüllt ihren speziellen Zweck. Der Zweck ist innerhalb einer Schicht immer derselbe. Schichten dürfen nicht übersprungen werden und Daten fließen nur in eine Richtung.

Gelingt eine klare und überschneidungsfreie Definition der Schichten, bietet ein solcher Aufbau eine klargegliederte Struktur des DWH. Dies stellt sicher, dass die Komplexität für die folgenden Extract-Transform- und Load Prozesse nach dem generischen Prinzip auf ein Minimum reduziert werden kann.

Die folgende Abbildung zeigt einen umfassenden Architekturansatz, je nach Projektart und Größe werden nur Teile davon umgesetzt..

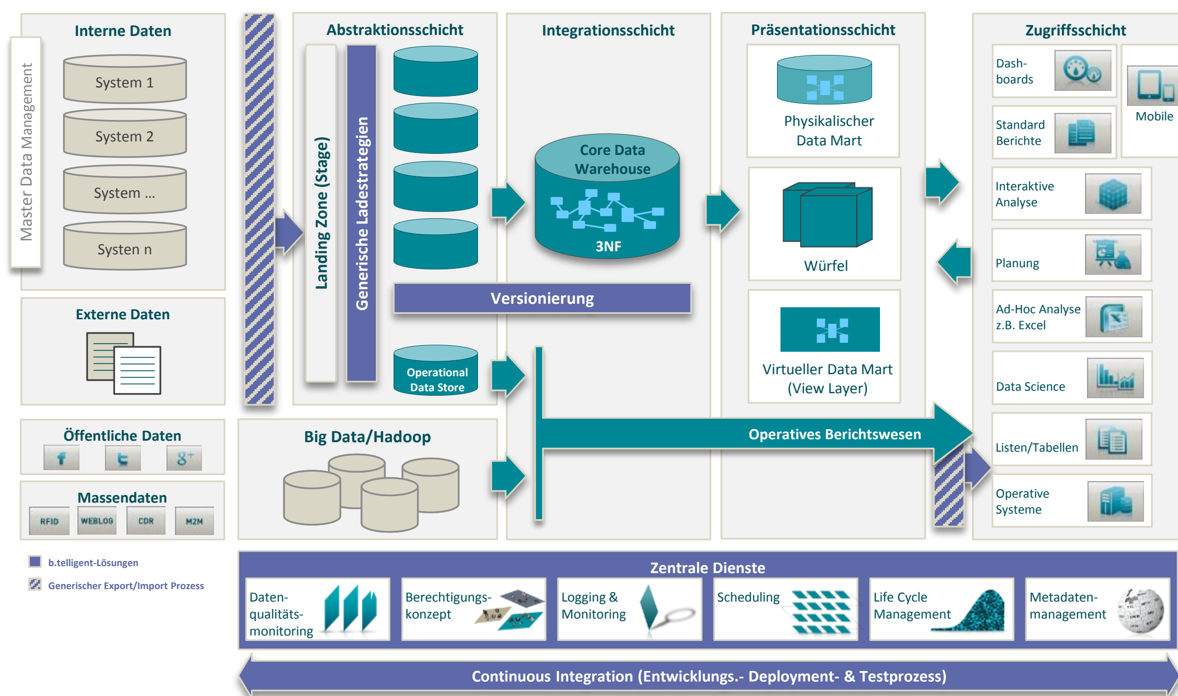

Abb. 1: DWH Referenzarchitektur

In diesem Projekt wurden diese Schichten wie folgt implementiert:

Staging Area:
Der Inhalt aller relevanter Spalten aller relevanter Tabellen aus dem Vorsystemen landen vollständig in dieser Schicht. Dabei bleiben Namen und Datentypen erhalten. Diese Schicht dient lediglich dazu die Daten den weiteren Prozessschritten einfach zugreifbar zu machen und dabei nur so kurz wie möglich vom Vorsystem abhängig zu sein.

Source Abstraction Area:

Alle Daten aus der Staging Area werden in die Source Abstraction Area übernommen. Dabei werden die Daten um weitere Informationen angereichert wie der Zeitstempel an dem die Daten übernommen wurden und ob es sich um eine Änderung handelt. Daten die in der Staging Area nicht mehr existieren werden ebenfalls kenntlich gemacht. Das bedeutet, es wird eine Deltaberechnung durchgeführt. Alle Namen und Datentypen bleiben weiterhin unverändert.

Cleansing Area:
An dieser Stelle findet der Bruch mit der Nomenklatur des Vorsystems statt. In Views werden die Tabellen aus der Source Abstraction Area zusammengeführt. Sämtliche Tabellen und Spalten bekommen nun Namen die im weiteren DWH eine sinnvolle Verwendung finden. Die Views entsprechen dabei den Entitäten, die in der nächsten Schicht aufgebaut werden. In den Views steckt die Logik wie das DWH zu beladen ist.

Core DWH:
Hier findet sich der Kern des DWHs. Alle Tabellen liegen in der dritten Normalform vor. Alle Tabellen werden SCD2 historisiert, das bedeutet jede Änderung führt zu einem neuen Historiensatz in der Tabelle.

Presentation Layer:
Dies ist die Zugriffsschicht für alle externen Anwendungen. Hier wird die Business Logik in Views implementiert.

ETL Technik
Eine herkömmliche PL/SQL Implementierung des DWHs führt in der Regel zu einer riesigen Menge an Codezeilen verteilt auf eine große Anzahl an Packages. Die Nachteile einer solchen Herangehensweise werden deutlich, wenn man sie mit einem generisch erzeugten DWH vergleicht.

Programmiertes DWH
Egal ob ETL/ELT-Tools eingesetzt werden oder sämtliche Prozesse in PL/SQL implementiert werden, es liegt üblicher Weise folgendes Szenario zu Grunde.

Für jede Zieltabelle gibt es eine Prozedur (oder ein Objekt im Tool), die die Logik enthält, aus welchen Tabellen welche Spalten auf welche Spalten der Zieltabelle wirken, wie diese Tabellen zusammen geführt werden müssen und welche Einschränkungen es dabei gibt.

- ➔ n Zieltabellen führt zu n Prozeduren

Beispiel: Eine Tabelle, die den Kunden repräsentiert, soll um eine Spalte für die neue IBAN erweitert werden. Ausgehend von der Quelldatei werden die Daten von der Staging Area durch die Source Abstraction und anschließend die Cleansing Area in das Core DWH geladen. Diese Änderung muss somit durch vier Vortabellen durchgeschleift werden. Die Folge davon ist, vier Tabellen müssen erweitert und vier Prozeduren geändert, getestet und ausgerollt werden.

Generisches DWH
Das Grundprinzip des Code Generators in Oracle ist, dass mittels einer PL/SQL-Prozedur ein dynamisches SQL generiert und dieses dann ausgeführt wird.

Beispielsweise lässt sich anhand dieses Codefragments ein einfacher Codegenerator programmieren:

```
DECLARE
BEGIN
   FOR rec IN (SELECT 'INSERT INTO '
|| owner
|| '.'
|| table_name
|| ' (start_time) VALUES (SYSDATE)' AS v_query
                 FROM sys.all_tables
                WHERE owner = 'CORE' AND table_name LIKE '%LOGGING')
   LOOP
      EXECUTE IMMEDIATE rec.v_query;
   END LOOP;

   COMMIT;
END;
```

Was passiert hier: Aus der Oracle Katalog-Tabelle sys.all_tables werden alle Tabellen im Schema CORE deren Tabellenname auf LOGGING enden gesucht.

Während der Laufzeit wird ein Insert Statement erzeugt, dass die aktuelle Zeit in eine Spalte start_time in jede dieser Tabellen schreibt.

Dadurch reduziert sich der Aufwand Prozeduren zu implemetieren von n auf 1.

Ebenso reduziert sich der Testaufwand auf ein Minimum sobald die generische Prozedur einmalig getestet wurde.

Solche Codegeneratoren lassen sich an diversen Stellen im DWH einsetzen. Einen Überblick gibt folgende Grafik:

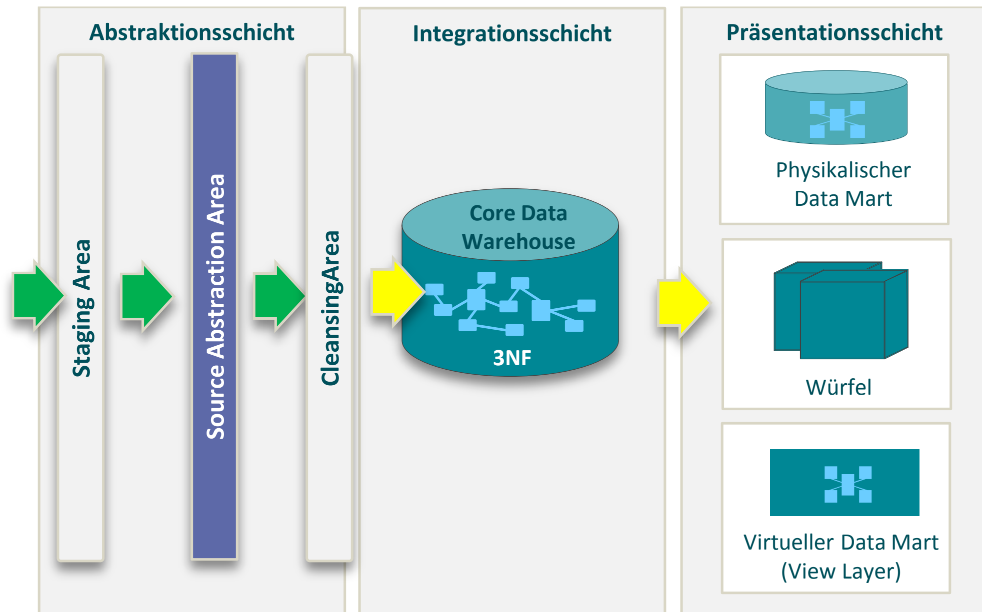

Abb.2: Einsatzmöglichkeiten für generischen Code

Im oben genannten Beispiel zur IBAN wäre lediglich die View, in der die Ladelogik der Cleansing Schicht enthalten ist, anzupassen. Durch die Erweiterung der Tabellen und der View um die neue Spalte, passt sich der vom Generator erzeugte Code analog mit an und es muss nichts selbst implementiert werden.

Prozessteuerung

Ein Codegenerator ist das Werkzeug um Programmier- und Testaufwand niedrig zu halten. Interessant ist wie diese Generatoren eingesetzt werden. Um zu vermeiden viel PL/SQL-Code zu schreiben in dem die Ablaufsteuerung hinterlegt ist werden Meta-Daten-Tabellen eingesetzt. Der PL/SQL-Code reduziert sich durch den Einsatz dieser Meta-Tabellen größtenteils auf einen Loop durch diese Tabellen.

Meta-Datenmodell

Obwohl der Großteil des DWHs durch generischen Code gesteuert wird, kann das Meta-Datenmodell auf ein minimales Maß reduziert werden.

Im Wesentlichen werden zwei Tabellen mit meta-informationen benutzt.

Die Tabelle *Meta_stage_ctrl* beschreibt den ersten Schritt im Ladeprozess, die Staging und Source Abstraction Schicht. Dort wird abgebildet in welchem Prozess Daten aus welchen Tabellen aus welcher Schicht in welche Tabelle in welcher Schicht fließen.

Die Tabelle *Meta_cleanse_ctrl* beschreibt den zweiten Schritt im Ladeprozess, die Cleansing Schicht und das Core-DWH. Dort wird abgebildet welche Tabellen zusammen mit welchen Sequences welche Tabellen im Core bewirtschaften.

Durch Verwendung dieser beiden Tabellen ist es ausreichend lediglich eine Prozedur lade (p_process_name => ‚Bewegungsdaten‘) aufzurufen, um einen bestimmten Teil des DWH zu laden. Im Hintergrund werden dann Subprogramme aufgerufen die sich alle benötigten Daten aus den Meta-Tabellen zusammen suchen und darauf dynamischen SQL generieren.

Programmlogik

Bislang wurde beschrieben, wie Daten von einer Schicht in die andere gelangen und wie Tabellen zu Prozessen zugeordnet werden. Der entscheidende Teil im DWH ist jedoch der Logik Teil.
Diese teilt sich in zwei Bereiche auf, die Lade- und die Businesslogik.

Ladelogik

Die Ladelogik findet sich in der Cleansing Area wieder und erfüllt die Aufgaben, Tabellen aus dem Vorsystem logisch zu einer Entität im DWH zusammenzuführen. Dabei findet der Umbruch der Nomenklatur des Vorsystems hin zu der des DWH statt, d.h. sowohl Spalten als auch Tabellen bekommen Namen, die DWH intern konsistent und verständlich sind. An dieser Stelle werden die Tabellen um die technischen IDs des DWHs erweitert, d.h. jede Tabelle bekommt über eine Sequence ihre eigenen IDs. Ebenfalls werden die Fremdschlüssel auf DWH interne Technik umgeschlüsselt. Es ist dabei zu beachten, dass an dieser Stelle keinerlei Information entsteht, sondern lediglich zusammengeführt wird.
Aus buchungstechnischen oder ähnlichen Gründen können diese Daten im Vorsystem in mehreren Tabellen zu finden sein. In diesem Fall muss in der Cleansing Area die Logik hinterlegt sein, wie diese Tabellen zusammen zu führen sind.

```
CREATE OR REPLACE VIEW kunde
AS
SELECT x.id, x.name AS vorname, x.name2 AS nachname, x.b_id as
tec_bundesland_id, nvl(b.dwh_id,-1) as dwh_bundesland_id
FROM kunden_mandant1 x left outer join core.bundesland b on x.b_id =
b.tec_id
UNION ALL
SELECT y.x100 AS id, z.x101 AS vorname, z.x102 AS nachname, y.y4 as
tec_bundesland_id, nvl(b2.dwh_id,-1) as dwh_bundesland_id
FROM kunden_mandant2_key y
LEFT OUTER JOIN kundne_mandant2_values z ON y.x100 = z.x100
LEFT OUTER JOIN core.bundesland b2 on y.y4 = b2.tec_id
;
```

Solange die Vorsysteme ihre Architektur nicht verändern bleibt diese Logik konstant. Diese Logik ist ausschließlich von den Vorsystemen abhängig und nicht vom Business.

Businesslogik

Die Business Logik findet sich im Presentation Layer wieder, d.h. oberhalb des Core DWHs. Im Gegensatz zur Ladelogik, die die Entitäten im DWH vorbereitet, enthält die Business Logik das Wissen der Fachbereiche und erzeugt neue abgeleitete Informationen. Es ist ebenfalls zu erwarten, dass die Businesslogik viel häufiger an neue Anforderungen angepasst werden muss. Theoretisch ist für die Businesslogik keinerlei Kenntnis über die Technik der Vorsysteme notwendig. Im beschriebenen Projekt war es möglich die Business Schicht komplett ohne Materialisierungen ausschließlich durch Views umzusetzen.

Beispielsweise wäre das Gehalt eines Mitarbeiters in der Vergangenheit immer als Grundgehalt + variabler Anteil errechnet worden.

Gehalt = Grundgehalt + variabler Anteil

Durch eine spätere fachliche Anforderung ergibt sich eine neue Berechnungslogik, die den geldwerten Vorteil noch mit einbezieht.

Gehalt = Grundgehalt + variabler Anteil + geldwerter Vorteil

Nach anschließender, betrieblicher Umstrukturierung müssen die Mitarbeiter nun ihren Parkplatz selbst bezahlen, weshalb das Gehalt sich nun wie folgt berechnet:

Gehalt = Grundgehalt + variabler Anteil + geldwerter Vorteil - Parkgebühr

Wäre das Gehalt eine Spalte im Core-DWH, so müssten für jede Änderung die echten Daten im Core DWH durch ein Update verändert werden. Wenn man diese Logik jedoch in der Business Schicht als View implementiert, muss nur diese eine View angepasst werden und Änderung lassen sich jederzeit zurücknehmen.

Fazit

Werden diese Prinzipien angewandt so entsteht ein DWH das aus mehreren Schichten besteht. Ein großer Teil der Programme darin werden durch Code Generatoren erzeugt. Es gibt genau Zwei Stellen an denen ein Programmierer Logik Implementieren muss. Zum einen die Ladelogik in der Cleansing Area, zum anderen die Business Logik im Presentation Layer. Durch dieses Vorgehen gibt es einen einfach Nachzuvollziehenden Datenfluss und nur wenige Stellen an denen programmiert werden muss.

Was bedeutet das in der Praxis?

Vorteile für den Entwickler/DBA
Was sind die realen Vorteile eines generischen DWHs? Die Struktur des DWHs ist einfach zu überschauen (siehe Referenzdatenmodell). Ebenso ist der Datenfluss im DWH sofort ersichtlich.

Auf Fragen woher welche Information kommt und warum sie nicht so ist wie erwartet, kann ohne großen Analyseaufwand adhoc geantwortet werden. Dazu müssen lediglich die Schichten zurück bis ins Vorsystem verfolgt werden. Dies fällt durch den strukturieren Datenfluss der keine Seiteneffekte zulässt sehr leicht. Um die Übersicht noch einfacher zu gestalten wurde in diesem Projekt BI Wiki eingesetzt. Dies ist ein Produkt von b.telligent, das basierend auf einem Wiki, die Prozesse und Zusammenhänge der Datenbank generisch liest und grafisch aufarbeitet. Damit kann dieses Wissen auch Personen ohne Zugriffsrechten zugänglich gemacht werden.

Änderungen im DWH sind sehr einfach und erfordern keine Implementierung in PL/SQL und nur wenige Änderungen in SQL. Die Zeit, die zum Testen benötigt wird, reduziert sich deutlich, da nur noch sehr kleine Testszenarien überprüft werden müssen. Nur die programmieren Teile, also die View mit Ladelogik und Businesslogik müssen intensiv getestet werden.

Aufwandsschätzungen fallen in einer weniger komplexen Umgebung ebenso deutlich leichter.

Vorteile für das Management
Die treibenden Kosten für ein DWH sind die Kosten für die Instandhaltung und die ausufernden Kosten für Erweiterungen.

Am Beispiel der Erweiterung des DWHs um neue Spalten ergibt sich meiner Erfahrung nach ein Vorteil von >10.000€ in einer Vergleichsimplementierung zu 1.000€ in einer generischen Implementierung. d.h. ein Faktor > 10.

Geht man weiterhin davon aus, dass etwa alle zwei Monate neue Spalten in ein DWH hinzugefügt werden müssen, so kommt man von >60.000€ auf 6.000€ also einer jährlichen Ersparnis von mindestens 54.000€ allein durch den Task „Neue Spalten hinzufügen“.

Neben den geringeren Kosten ist möglicherweise ein ebenso wichtiger Faktor der, dass die Time-To-Market deutlich reduziert werden kann und BI-Anforderungen schneller bedient werden können. Solange alle Informationen im Core vorhanden sind müssen lediglich die View die die Business Logik enthalten erweitert werden. Es ist also keine technischen Kenntnisse über die Vorsysteme notwendig.

Durch die strikte Trennung von Ladelogik und Businesslogik kann in den beiden Schichten unabhängig voneinander entwickelt bzw. getestet werden und die Auswirkungen von Änderungen bleiben leicht überschaubar.

Die Reduktion der Komplexität und die Bündelung der Logik auf nur zwei Stellen erlaubt es, schneller Wissen zu transferieren. Urlaube und Krankheiten einzelner Mitarbeiter führen z.B. nicht mehr so leicht zu kritischem Kenntnisverlust.

Die freigewordenen Ressourcen können sich damit voll auf die Umsetzung der Businesslogik konzentrieren. Damit kann ein DWH seiner eigentlichen Aufgabe, Business Intelligence voranzutreiben deutlich besser gerecht werden. Daten schnell, kostengünstig und verlässlich zur Verfügung zu stellen und neue Fragestellungen kurzfristig beantwortet zu können, muss das Ziel sein.

Der Mehrwert liegt dann in einer verbesserten Unterstützung der Fachbereiche, was dazu führt, dass das DWH somit direkt zum Unternehmenserfolg beiträgt.

Über b.telligent

b.telligent mit Sitz in München - sowie Niederlassungen in Düsseldorf, Hamburg, Stuttgart und einer Schwestergesellschaft in Zürich - ist eine Unternehmensberatung, die auf Einführung und Weiterentwicklung von Business Intelligence, Customer Relationship Management und E-Commerce in Unternehmen in Massenmärkten spezialisiert ist. Der Fokus liegt dabei auf der kontinuierlichen Optimierung von Geschäftsprozessen, Kunden- und Lieferantenbeziehungen durch den Erkenntnisgewinn aus der Verdichtung und Analyse von systemübergreifenden Geschäftsdaten. So lassen sich Margen erhöhen, Kosten senken und Risiken besser kontrollieren. Kunden von b.telligent sind Branchenführer aus den Bereichen Telekommunikation, Finanzdienstleistung, Handel und Industrie.

Kontaktadresse

Tobias Braunschober
b.telligent GmbH & Co. KG
Georg-Brauchle-Ring 52-54
D-80992 München

Telefon:	+49 (0) 176-100 0983
E-Mail	tobias.braunschober@btelligent.com
Internet:	www.btelligent.com