SCD2 mal anders

Andrej Pashchenko
Trivadis GmbH
Düsseldorf

Schlüsselworte
Slowly Changing Dimensions Type 2, Historisierung.

Einleitung

Die Historisierung der Daten ist eine übliche, aber auch komplexe und rechenintensive Aufgabe in einem Data Warehouse System. Man hat die Herausforderung beim Laden von der historisierten Core-Schicht, Data Vault Modellen, Dimensionen, usw. Die herkömmliche Vorgehensweise fängt mit einem Outer Join an und beinhaltet eine Art Änderungserkennung, die mit Berücksichtigung der NULL-Werte durchgeführt werden soll und an sich wahrscheinlich den kompliziertesten Teil darstellt. Nun, wenn man an Standardfunktionalität der SQL-Sprache denkt, kommt man schnell auf zwei Bordmitteln, die Gleichheitsprüfung (und somit auch Änderungserkennung) auf die gewünschte Weise ausführen: GROUP BY Klausel oder PARTITION BY Klausel bei analytischen Funktionen. Kann man diese für die SCD2-Historisierung einsetzen? Macht es Sinn? Und wie wird der ETL-Prozess aussehen?

Einführung und die typische Vorgehensweise

Schauen wir uns folgendes Beispiel an. Eine Quelltabelle wird täglich in eine Zieltabelle ins Datawarehouse System geladen. Die Zieltabelle hat Datumsspalten für die Gültigkeitsangaben und erlaubt mehrere Versionen pro Business Key. Wenn wir für einen Business Key an einem Ladetag Änderungen in den gelieferten Daten feststellen, schließen wir die aktuelle Version ab (setzen das Gültigkeitsenddatum) und legen eine neue Version an. Keine Daten werden überschrieben und alle Änderungen sind über den gesamten Zeitraum nachvollziehbar.

Der meist verbreitete Ansatz führt zunächst einen Outer Join von der Quelltabelle mit den aktuellen Versionen aus der Zieltabelle über den Business Key durch. Wenn wir im Join-Ergebnis den Business Key nur auf einer Seite des Joins haben, bedeutet das, dass wir mit neuen oder gelöschten Datensätzen zu tun haben. Wenn der Key auf beiden Seiten vorhanden ist, müssen wir die tatsächlichen Änderungen weiter untersuchen. Hier müssen wir darauf achten, dass die NULL-Werte richtig berücksichtigt werden. Das macht die Prüfung komplizierter. Zum Einsatz kommen hier verschiedene Techniken: datentypabhängige NVL-Aufrufe, DECODE-s, die (inzwischen dokumentierte) Funktion LNNVL, nicht dokumentierte SYS_OP_MAP_NONNULL, etc. Auch über das Erzeugen, Abspeichern und Abgleichen von HASH-Werten über alle Spalten kommt man ans Ziel.

Neben der Komplexität der Änderungserkennung existiert ein weiteres Problem: man muss die alten Versionen der Daten „abschließen“ und die neuen Versionen einfügen. Die Ermittlung entsprechender Datenmengen bedeutet in der Regel einen zweifachen Zugriff und Join von der Quell- und Zieltabelle, ob man es in einem ETL-Prozess mit einem MERGE-Statement macht oder in zwei separaten Prozessen mit UPDATE und INSERT.

Alternativer Ansatz

Wie könnte eine alternative Vorgehensweise aussehen? Interessanterweise kann dabei ein SQL Bordmittel helfen, dass sicher nicht nur vom Autor sehr oft für „ad hoc“ Abfragen verwendet wird: die GROUP BY Klausel (oder alternativ die PARTITION BY Klausel einer analytischen Funktion). Wie funktioniert das?

Angenommen, wir haben zwei Datensätze, die auf unterschiedliche Werte in gleichen Spalten zu prüfen sind. Wir können über alle Spalten gruppieren und auf den Wert der Funktion `COUNT(*)` schauen. Wenn die Datensätze komplett gleich sind, fallen sie durch Gruppierung zusammen und der Wert von `COUNT(*) = 2`. Sind die Datensätze dagegen in mindestens einer Spalte unterschiedlich, „überstehen" beide Datensätze die Gruppierung und liefern jeweils `COUNT(*) = 1` zurück.

Auch das Verhalten der GROUP BY Klausel bezüglich NULL-Werte passt exakt zu unseren Anforderungen: zwei NULL-Werte gelten als „gleich", der Vergleich von NULL- und nicht-NULL-Werten funktioniert direkt und heißt im Ergebnis „ungleich".

Schauen wir an einem Beispiel konkret an, wie der ETL-Prozess aufgebaut ist. Wir versionieren dabei die Mitarbeiterdaten einer erfundenen Firma in der Tabelle T_TARGET (s. Abb. 1). Neue Daten kommen täglich über die Tabelle T_SOURCE rein (s. Abb. 2). In den nächsten Code-Listings bauen wir Schritt für Schritt das MERGE-Statement zur Übernahme neuer Daten auf. Um Platz zu sparen, werden in nachfolgenden Listings die bereits ausgearbeiteten Teil-Abfragen aus vorherigen Listings referenziert.

DWH_KEY	VALID_FROM	VALID_TO	CUR_VERSION	ETL_OP	BUS_KEY	FIRST_NAME	SECOND_NAMES	LAST_NAME	HIRE_DATE	FIRE_DATE	SALARY
1	01.12.2016		Y	INS	123	Roger		Federer	01.01.2010		900000
2	01.12.2016		Y	INS	456	Rafael		Nadal	01.05.2009		720000
3	01.12.2016		Y	INS	789	Serena		Williams	01.06.2008		650000

Abb. 1: Inhalt der Zieltabelle T_TARGET

LOAD_DATE	BUS_KEY	FIRST_NAME	SECOND_NAMES	LAST_NAME	HIRE_DATE	FIRE_DATE	SALARY
02.12.2016	123	Roger		Federer	01.01.2010		900000
02.12.2016	456	Rafael		Nadal	01.05.2009		720000
02.12.2016	789	Serena	Jameka	Williams	01.06.2008		650000
02.12.2016	345	Venus		Williams	01.11.2016		500000

Abb. 2: Inhalt der Quelltabelle T_SOURCE

Als Ausgangpunkt haben wir nach wie vor die Quelltabelle und die aktuellen Versionen in der Zieltabelle. Diesmal joinen wir diese nicht, sondern führen sie mit einem UNION ALL Operator zusammen – Listing 1. Um später in dem Zwischenergebnis die Datensätze aus der Quell- und der Zieltabelle unterscheiden zu können, führen wir ein entsprechendes Kennzeichen (Spalte `SOURCE_TARGET`) ein.

```
-- Hier aktuelle Versionen aus der Zieltabelle
SELECT  'TARGET' source_target  -- um die Datensätze später unterscheiden zu können
,       business_key
,       dwh_valid_from
,       first_name
,       second_names
,       last_name
,       hire_date
,       fire_date
,       salary
FROM    t_target
WHERE   current_version = 'Y'
-- Hier die neuen Daten
UNION ALL
SELECT  'SOURCE' source_target
,       business_key
,       load_date dwh_valid_from  -- Wir nutzen load_date as neues Gültigkeits-Startdatum
,       first_name
,       second_names
,       last_name
,       hire_date
,       fire_date
```

```
,       salary
FROM    t_source;
```

Listing 1 UNION ALL Quell- und Zieltabellen

Über diesen Zwischendatenbestand führen wir dann die Gruppierung aus und zählen die Datensätze. Dies kann man an dieser Stelle mit Einsatz der analytischen Funktionen lösen (alle SCD2-Spalten in PARTITION BY) oder auch mit der klassischen GROUP BY Klausel wie im Listing 2.

```
WITH union_source_target AS
(
...
-- Hier kommt die Abfrage aus dem Listing 1 (UNION ALL)
...
)
SELECT
--
--  Wenn cnt = 2 – die zwei Versionen waren gleich, ansonsten INS/UPD/DEL
--
        COUNT(*) cnt
 ,      MIN(un.source_target) source_target
 ,      MIN(un.valid_from ) valid_from
 ,      CASE
            WHEN MIN(un.source_target) = 'TARGET' THEN DATE '&load_date' -1
                      ELSE MAX(un.valid_to)
           END valid_to
 ,      MIN(un.cur_version) cur_version
 ,      bus_key
 ,      first_name
 ,      second_names
 ,      last_name
 ,      hire_date
 ,      fire_date
 ,      salary
 FROM   union_source_target un
 GROUP BY bus_key, first_name, second_names, last_name, hire_date, fire_date, salary
```

Listing 2. Gruppierung über SCD2-Spalten

Anschließend filtern wir die Datensätze ohne Änderungen (`COUNT(*)=2`) heraus. Das daraus resultierende Zwischenergebnis kann schon als Grundlage für das abschließende MERGE in die Zieltabelle verwendet werden, siehe Listing 3.

```
MERGE INTO  t_target t
USING (
...
-- Hier kommen die Abfragen aus dem Listing 2
...
SELECT *
FROM   action_needed
WHERE  cnt != 2 ) q
ON ( t.bus_key  = q.bus_key AND t.valid_from = q.valid_from  )
WHEN NOT MATCHED THEN INSERT (    dwh_key
                                , valid_from
                                , valid_to
                                , cur_version
                                , etl_op
                                , bus_key
                                , first_name
                                , second_names
                                , last_name
                                , hire_date
                                , fire_date
                                , salary )
```

```
    VALUES (  scd2_seq.nextval
            , q.valid_from
            , q.valid_to
            , 'Y'
            , 'INS'
            , q.bus_key
            , q.first_name
            , q.second_names
            , q.last_name
            , q.hire_date
            , q.fire_date
            , q.salary )
WHEN MATCHED THEN UPDATE SET
             t.valid_to = q.valid_to
           , t.cur_version = 'N'
           , t.etl_op = 'UPD';
```

Listing 3. Datenübernahme mit einem MERGE-Statement

Nach dem Ausführen vom MERGE-Befehl wurden erwartungsgemäß die neuen Versionen für Business Key 789 und 345 angelegt, sowie die alte Version für Business Key 789 abgeschlossen:

DWH_KEY	VALID_FROM	VALID_TO	CUR_VERSION	ETL_OP	BUS_KEY	FIRST_NAME	SECOND_NAMES	LAST_NAME	HIRE_DATE	FIRE_DATE	SALARY
1	01.12.2016		Y	INS	123	Roger		Federer	01.01.2010		900000
6	02.12.2016		Y	INS	345	Venus		Williams	01.11.2016		500000
2	01.12.2016		Y	INS	456	Rafael		Nadal	01.05.2009		720000
3	01.12.2016	01.12.2016	N	UPD	789	Serena		Williams	01.06.2008		650000

Abb. 3: Inhalt der Zieltabelle T_TARGET nach der Datenübernahme per MERGE

Wenn das Quellsystem eine physikalische Löschung der Datensätze erlaubt, muss man für eine saubere Abbildung im Data Warehouse noch einige zusätzliche Vorkehrungen in dem SQL-Statement machen. Diese erhöhen die Komplexität nur unwesentlich und sind in den Demo-Skripten zu der Präsentation enthalten.

Use Cases und Performance-Aspekte

Wie sieht es mit der Performance der vorgeschlagenen Lösung in einigen häufigen Anwendungsfällen aus?

Wir betrachten das Laden einer versionierten Tabelle im Datawarehouse Core-Bereich aus einer Staging-Tabelle. Schauen wir uns erst mal mögliche Konstellationen an, welche Datenmengen wir zu verarbeiten haben und wo die Performance verloren gehen kann. Grundsätzlich gibt es zwei Implementierungsvarianten für diesen ETL-Prozess: Voll- und Delta-Laden.

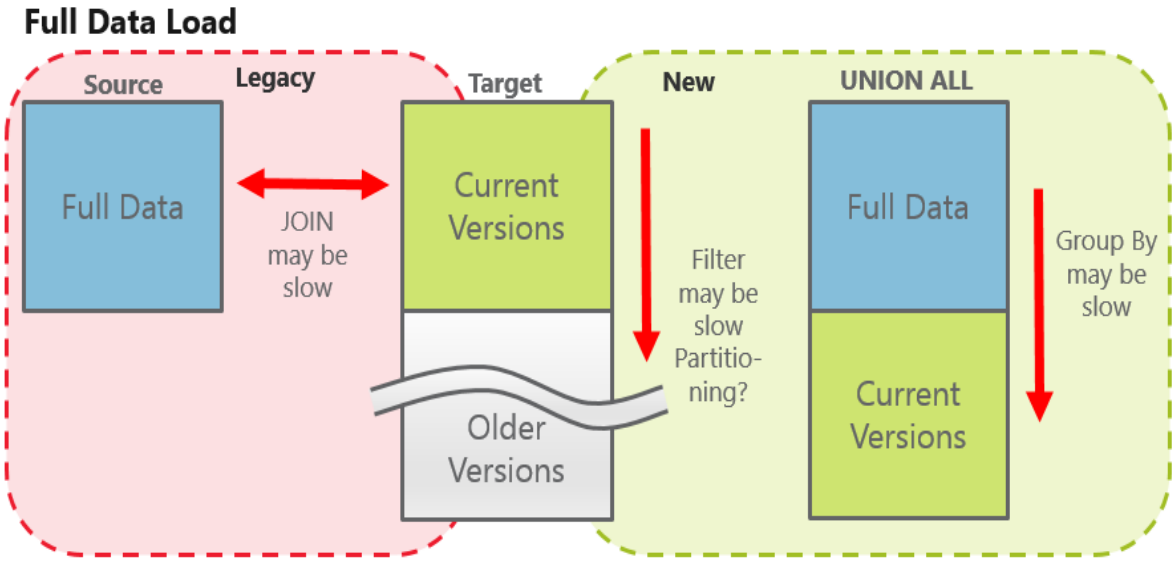

Abb. 4: Vollladen

Beim Vollladen haben wir den vollen Datenabzug aus dem Quellsystem im Staging-Bereich. Der Join von diesem Vollabzug zu allen gültigen Datensätzen in der versionierten Tabelle kann unter Umständen langsam sein. Und dieser Join muss dazu auch noch zwei Mal ausgeführt werden, es sei denn wie würden die Zwischenergebnisse des Joins im ETL-Prozess materialisieren.

Auf der anderen Seite müssen wir mit dem „neuen“ Ansatz über die Datenmenge gruppieren, die zwei Mal so groß ist wie der Vollabzug der Daten aus dem Quellsystem. Und das kann auch langsam sein!

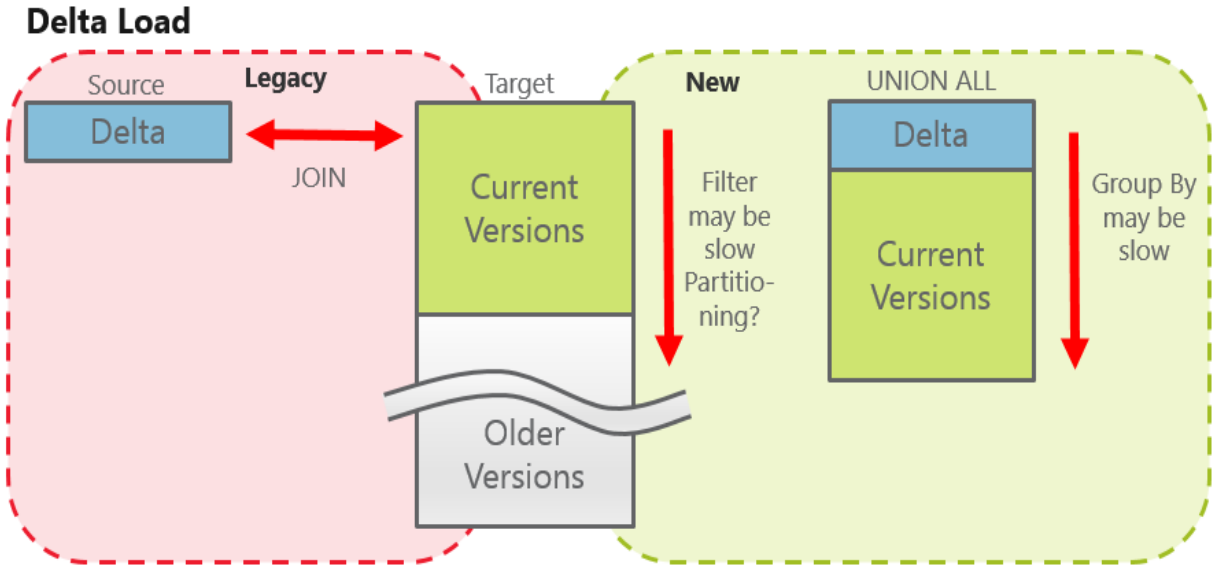

Abb. 5: Delta-Laden

Beim Delta-Laden haben wir im Staging-Bereich in der Regel nur einen kleinen Teil der Daten – die neuen und die aktualisierten Datensätze. Somit kann der Join zu gültigen Datensätzen im Core

ziemlich effizient sein, wobei wir mit dem „neuen“ Ansatz immer noch über beträchtliche Datenmenge gruppieren müssen: alle aktuellen Datensätze plus „Delta“. Wie können wir das verbessern?

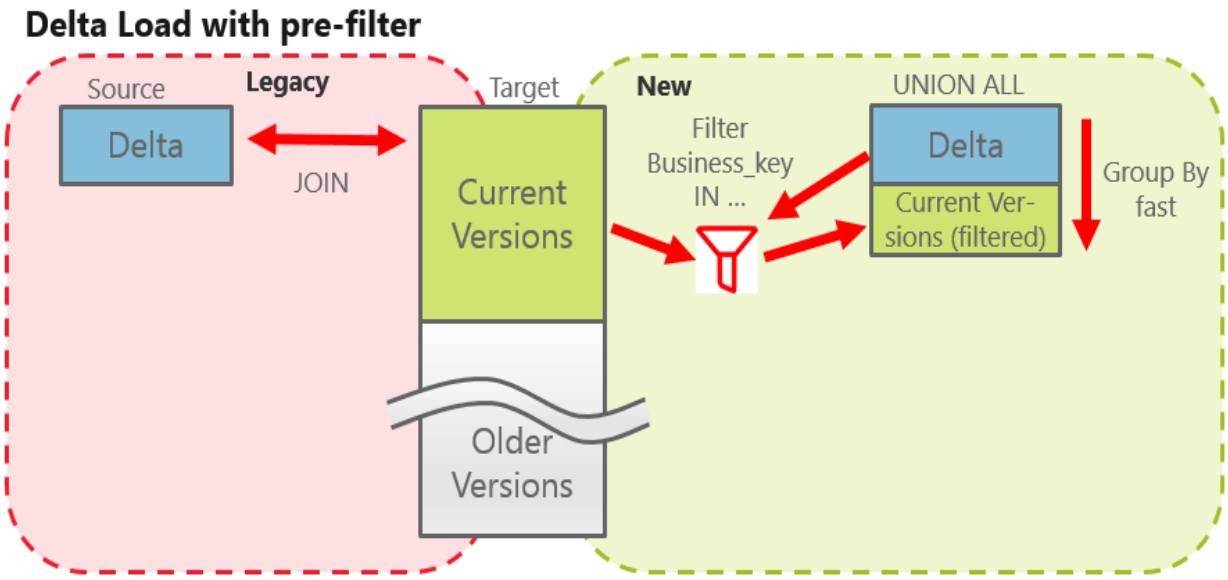

Abb. 6: Delta-Laden mit einem Vorfilter

Offensichtlich müssen wir nur die Datensätze vergleichen (auf die Änderungen untersuchen), die es bereits im Core-Bereich gab. Wir können eine Art Vorfilter einbauen, zum Beispiel als IN- oder EXISTS-Unterabfrage über den Business Key. Somit muss am Ende nur über eine Datenmenge gruppiert werden, die ungefähr zwei Mal so groß ist, wie der Delta-Extrakt.

Einige Rahmenbedingungen und Zahlen zum Testfall:

- ziemlich breite Tabelle mit 120 Spalten
- Vergleich des herkömmlichen Ansatzes mit GROUP BY bzw. analytischen Funktion
- Vollextrakt in der Staging-Tabelle als Quelle vs. Deltaextrakt (mit und ohne Vorfilter)
- ca. 6 Mio Datensätze in der Zieltabelle
- ca. 3 Mio Datensätze im Vollextrakt
- ca. 3000 Datensätze im Deltaextrakt

Vorgehensweise	Delta-Laden, min	Vollladen, min
Outer Join (herkömmliche Ansatz)	0:09	0:41
GROUP BY	1:10	1:04
GROUP BY mit Vorfilter	0:04	N/A
Analytische Function	2:12	4:52
Analytische Function mit Vorfilter	0:12	N/A

Beim Laden von Deltaextrakt sah der herkömmliche Ansatz aus der Performance-Sicht ziemlich gut aus. Vor allem seine Komplexität, mit dem aufwendigen Vergleich aller Spalten oder Generieren, Speichern und Vergleichen eines Hash-Diffs für alle Datensätze, war der Grund für die Suche nach einem alternativen Verfahren. Auf der anderen Seite, der „reine“ GROUP BY Ansatz ist einfacher

aber deutlich langsamer. Der Gewinner ist hier der GROUP BY mit Vorfilter. Aber der Preis ist wieder eine etwas höhere Komplexität.

Beim Vollladen kann der „neue“ Ansatz mit dem Join nicht mithalten – er ist um ca. 50% langsamer.

Um den Unterschied zwischen GROUP BY und analytischen Funktion zu verstehen, müssen wir uns die Ausführungspläne anschauen. Das Gruppieren wurde bis Oracle 10g Release 2 mithilfe der Operation SORT GROUP BY durchgeführt. Und diese ist vergleichbar mit WINDOW SORT, die wir in dem Plan für die analytische Funktion COUNT() sehen. Mit 10g Release 2 wurde **hash aggregation** eingeführt. Diese können wir durch die Schritte HASH GROUP BY im Ausführungsplan erkennen (Listing 4, Zeile 21). Aber die analytischen Funktionen profitieren nicht von dem neuen Feature, die Zeit geht an den Zeilen 14 und 15 (Listing 5) – WINDOW SORT – verloren.

```
-------------------------------------------------------------------------
| Id  | Operation                                | Name                 |
-------------------------------------------------------------------------
|   0 | MERGE STATEMENT                          |                      |
|   1 |  MERGE                                   | CO_S_ORDER_TEST      |
|   2 |   VIEW                                   |                      |
|   3 |    SEQUENCE                              | SEQ_CO_S_ORDER       |
|   4 |     PX COORDINATOR                       |                      |
|   5 |      PX SEND QC (RANDOM)                 | :TQ10005             |
|*  6 |       HASH JOIN OUTER BUFFERED           |                      |
|   7 |        PX RECEIVE                        |                      |
|   8 |         PX SEND HASH                     | :TQ10003             |
|*  9 |          VIEW                            |                      |
|  10 |           WINDOW SORT                    |                      |
|  11 |            PX RECEIVE                    |                      |
|  12 |             PX SEND HASH                 | :TQ10002             |
|* 13 |              FILTER                      |                      |
|  14 |               HASH GROUP BY              |                      |
|  15 |                PX RECEIVE                |                      |
|  16 |                 PX SEND HASH             | :TQ10001             |
|  17 |                  HASH GROUP BY           |                      |
|  18 |                   VIEW                   |                      |
|  19 |                    UNION-ALL             |                      |
|  20 |                     PX BLOCK ITERATOR    |                      |
|* 21 |                      TABLE ACCESS FULL   | STG_S_ORDER_DELTA    |
|* 22 |                     HASH JOIN RIGHT SEMI |                      |
|  23 |                      PX RECEIVE          |                      |
|  24 |                       PX SEND BROADCAST  | :TQ10000             |
|  25 |                        PX BLOCK ITERATOR |                      |
|* 26 |                         TABLE ACCESS FULL| STG_S_ORDER_DELTA    |
|  27 |                      PX BLOCK ITERATOR   |                      |
|* 28 |                       TABLE ACCESS FULL  | CO_S_ORDER_TEST      |
|  29 |        PX RECEIVE                        |                      |
|  30 |         PX SEND HASH                     | :TQ10004             |
|  31 |          PX BLOCK ITERATOR               |                      |
|* 32 |           TABLE ACCESS FULL              | CO_S_ORDER_TEST      |
-------------------------------------------------------------------------
```

Listing 4 Ausführungsplan für GROUP BY mit Vorfilter

```
-------------------------------------------------------------------
| Id  | Operation                          | Name              |
-------------------------------------------------------------------
|   0 | MERGE STATEMENT                    |                   |
|   1 |  MERGE                             | CO_S_ORDER_TEST   |
|   2 |   VIEW                             |                   |
|   3 |    SEQUENCE                        | SEQ_CO_S_ORDER    |
|   4 |     PX COORDINATOR                 |                   |
|   5 |      PX SEND QC (RANDOM)           | :TQ10003          |
|*  6 |       HASH JOIN RIGHT OUTER BUFFERED|                  |
|   7 |        PX RECEIVE                  |                   |
|   8 |         PX SEND HASH               | :TQ10001          |
|   9 |          PX BLOCK ITERATOR         |                   |
|* 10 |           TABLE ACCESS FULL        | CO_S_ORDER_TEST   |
|  11 |        PX RECEIVE                  |                   |
|  12 |         PX SEND HASH               | :TQ10002          |
|* 13 |          VIEW                      |                   |
|  14 |           WINDOW SORT              |                   |
|  15 |            WINDOW SORT             |                   |
|  16 |             PX RECEIVE             |                   |
|  17 |              PX SEND HASH          | :TQ10000          |
|  18 |               VIEW                 |                   |
|  19 |                UNION-ALL           |                   |
|  20 |                 PX BLOCK ITERATOR  |                   |
|* 21 |                  TABLE ACCESS FULL | STG_S_ORDER_FULL  |
|  22 |                 PX BLOCK ITERATOR  |                   |
|* 23 |                  TABLE ACCESS FULL | CO_S_ORDER_TEST   |
-------------------------------------------------------------------
```

Listing 5 Ausführungsplan für analytische Funktion mit Vorfilter

Im oben betrachteten Testfall war eine Staging-Tabelle die Quelle für den ETL-Prozess. Was wäre, wenn wir eine Dimensionstabelle aus einer riesengroßen versionierten Core-Tabelle oder über eine View laden würden? Der Vorteil, dass wir die große Tabelle nicht zwei Mal lesen müssen oder die komplizierte View-Logik nicht zwei Mal ausführen müssen kann den Performanceverlust beim Gruppieren durchaus überwiegen, sogar ohne Vorfilter. Im untersuchten Testfall werden einige „große“ Tabellen (50 Gb, 40+ Mio Datensätze) in einer View gejoint. Daraus werden täglich etwa 500 Dimensionsdatensätze produziert. Die Ladezeit konnte um 45% reduziert werden (3 min 50 sec → 2 min).

Fazit

Der vorgeschlagene Ansatz ist eine einfache Alternative, die durchaus weiteres Testen verdient hat. Nicht in jedem Anwendungsfall. Nach meinem Verständnis sollte die Entwicklung eines Data-Warehouse-Systems hochautomatisiert ablaufen, mit Einsatz entsprechender Generator- /DWH-Automation Tools, die die ganze Komplexität der Deltaerkennung verstecken. Sollte das nicht der Fall sein und die ganzen ETL-Prozesse mit Versionierung der Daten im Wesentlichem manuell entwickelt werden, kann man durchaus diesen „neuen“ Ansatz in Betracht ziehen und von seiner Einfachheit in der Entwicklung und Pflege profitieren.

Kontaktadresse:
Andrej Pashchenko
Trivadis GmbH
Werdener Str. 4
D-40227 Düsseldorf

Telefon: +49 (0) 12-345 6789
Fax: +49 (0) 12-345 6788
E-Mail andrej.pashchenko@trivadis.com
Internet: www.trivadis.com