Analytische Funktionen selbst gemacht!
Data Cartridges für Data Warehouses

Gero Knapstein
OPITZ CONSULTING GmbH
Gummersbach

Schlüsselworte:

ODCI, Oracle Data Cartridge Interface, ODCIAggregate routines, user-defined aggregate, analytic function, aggregate function, object type, structured input, performance, example, OWB

Einleitung

Nahezu in jeder Data Warehouse Implementierung vermisst man eine passende analytische Funktion für die performante bzw. wartungsfreundliche Gestaltung des ETL. Dieser Vortrag soll Ihnen zeigen, wie einfach und sinnvoll es ist, sich Ihre analytische Funktion mit Hilfe der Data Cartridge Technologie selber zu schreiben. Es werden Ihnen Möglichkeiten für komplexe Verarbeitung und Debugging aufgezeigt. Die für Datawarehouse-Projekte nützlichen Funktionen DISTLISTAGG und HISTCHECK werden vorgestellt. Abschließend wird der Einsatz von selbstgeschriebenen analytischen Funktionen im OWB beschrieben.

Analytische Funktion im Vergleich

Analytische Funktionen grenzen sich in SQL-Befehlen gegenüber aggregierenden Funktionen und „Single Row“ Funktionen ab. Eine Single Row Funktion übernimmt die Werte aus genau einer Datenzeile und liefert den Rückgabewert an diese Datenzeile zurück. Eine aggregierende Funktion nutzt ein beliebig umfangreiches Datenzeilen-Set als Eingabe. Die Rückgabe erfolgt hingegen auf genau einer Datenzeile, was eine Aggregation der Ausgangsdatenzeilen mit expliziter oder impliziter Gruppierung notwendig macht.

Die analytische Funktion übernimmt Eingangswerte und liefert Rückgabewerte für das gleiche Datenzeilen-Set. Eine Aggregation ist damit nicht mehr notwendig. Darüber hinaus kann die Eingabe auf eine Teilmenge des Datenzeilen-Sets reduziert werden. Diese Teilmenge des Datenzeilen-Sets kann für jede Ergebniszeile eine andere Teilmenge sein.

SQL-Syntax der analytischen Teilschritte

Um dies zu erreichen, durchläuft die SQL-Engine mehrere Teilschritte, die sich syntaktisch im SQL-Befehl niederschlagen. Der Funktionsaufruf startet vergleichbar mit dem der aggregierenden Funktion, leitet dann aber über in ein Schlüsselwort für analytische Funktionen (i.d.R. „OVER“) und anschließend in die Beschreibung der folgenden drei Teilschritte, die jeweils lokal für die Funktionsanwendung durchgeführt werden:

- Laufzeit-Partitionierung des Ausgangsdatensets (→Query-Partition),
- Sortierung der Query-Partition (→sortierte Query-Partition)
- „Windowing“ (Teilmengenextraktion) auf der sortierten Query-Partition

Im folgenden Beispiel (Listing [1]) wird die Summen-Funktion „SUM()“ analytisch genutzt, um den „kumulierten Monatsabverkauf“ pro Artikel vom Monatsbeginn bis zum Vortag und den „Abverkauf der letzten 10 Kalendertage“ darzustellen.
In der „Query-Partition-Klausel“ des „kumulierten Monatsabverkauf“ wird entsprechend über Artikel und Monatsinformation gruppiert. Die Query-Partition von „Abverkauf der letzten 10 Kalendertage“ umfasst hingegen alle Sätze eines Artikels.
Die „Order-By-Klausel“ ist in beiden Anwendungsfällen gleich: Die Query-Partition wird zeitlich aufsteigend sortiert.
In der „Windowing-Klausel“ wird auf der sortierten Query-Partition die benötigte Teilmenge definiert: Für den „kumulierten Monatsabverkauf“ gehören zur Windowing-Teilmenge alle Zeilen zwischen Monatsanfang (dies entspricht der ersten Zeile der Query-Partition) und der aktuellen Zeile (physikalisches Windowing). Für den „Abverkauf der letzten 10 Kalendertage“ wird Teilmengenstart und –ende als Zeitdifferenz zum Datum der Ergebniszeile berechnet (logisches Windowing). Basierend auf die so definierten Query-Partition-Teilmengen leistet die Summenfunktion das zu Erwartende: Sie summiert die Abverkaufsmengen.

```
SELECT
  menge, datum, artikel_nr
, SUM (menge)
  OVER
  ( PARTITION BY to_char(datum, ’YYYYMM’), artikel_nr
    ORDER BY datum
    ROWS BETWEEN UNBOUNDED PRECEDING
             AND CURRENT ROW
  ) as ”Kumulierter Monatsabverkauf”,
, SUM (menge)
  OVER
  ( PARTITION BY artikel_nr
    ORDER BY datum
    RANGE BETWEEN NumToDSInterval(11, ’days’) PRECEDING
              AND NumToDSInterval( 1, ’days’) PRECEDING
  ) as ”Abverkauf der letzten 10 Tage”
FROM abverkauf;
```

Listing 1: analytische Berechnung von kumulierenden bzw. vergangenen Abverkaufsmengen

Für die meisten Oracle-eigenen analytischen Funktionen und für alle als Data Cartridge implementierten Funktionen gilt die Syntax:

```
SELECT …
<Funktionsname> (spaltenliste)
OVER
( <query-partition-Klausel>
  [<order-by-Klausel> [<windowing-Klausel>]]
) … FROM …
```

Weitere Details dazu finden sie unter Link [1].

Grundlagen für den Eigenbau: Die Oracle Data Cartridge Interfaces und Data Cartridges

Mit dem Oracle-Release 8 wurden nicht nur „Object Types", sondern auch etliche Erweiterungen etabliert, die den Einsatz von Object Types einfordern. Eine wesentliche Erweiterung war die Schaffung des „Oracle Extensibility Architecture Framework" als Grundlage für eine systematische Erweiterbarkeit der Datenbank-Fähigkeiten durch Drittanbieter. Als Schnittstellen wurden die „Oracle Data Cartridge Interfaces" (ODCI) für eine Reihe von Funktionalitäten geschaffen. Drittanbieter implementieren mit Hilfe von Object Types passend zu den Data Cartridge Interfaces sogenannte „Data Cartridges". Diese erweitern nach Deployment im Datenbankserver als „pluggable Extensions" die Datenbankserver-Funktionalität.

Mit Oracle 9i gab es eine Reihe von Erweiterungen und Schnittstellenänderungen an den ODCI-Interfaces. Seitdem sind diese Schnittstellen stabil und werden von Oracle mit der nötigen Umsicht erweitert. Data Cartridge -Entwickler und –Nutzer befinden sich also in der komfortablen Situation, ohne Nachbesserungen von den Oracle Entwicklungen durch Server Upgrades zu profitieren. Neben der Aufwärtskompatibilität bietet das Object Type Body Wrapping die notwendigen Voraussetzungen, Implementierungen auch für Fremdnutzer zu schaffen.

Zur Zeit werden von Oracle folgende ODCI-Interfaces bereitgestellt, von denen wir uns mit dem Letztgenannten befassen werden:

- ODCIIndex
  Dieses Interface ist die Grundlage für Domain-Index-Implementierungen. Es ermöglicht den Bau bedarfsspezifischer Index-Lösungen abseits der BTree- und Bitmap-Indizierungsverfahren. Als wohl bekanntestes Beispiel ist die von Oracle implementierte und häufig eingesetzte „Text-Option" zu nennen.
- ODCIStats
  Mit diesem Interface lässt sich implementieren, wie für Domain-Indizes Statistiken erstellt und vom Cost Based Optimizer genutzt werden.
- ODCITable
  Hinter diesem Interface verbirgt sich die Möglichkeit, Table-Functions zu entwickeln. Eine Besonderheit dieses Interfaces ist, dass es hochgradig nativ in PLSQL integriert ist. Viele Table-Functions schreibende PLSQL-Entwickler wissen nicht, dass sie genau genommen „Data Cartridges" entwickeln. Eine direktere Ansprache des ODCITable-Interfaces ist notwendig, wenn es um weiterführende Performance-Maßnahmen oder um die Einbindung externer Prozeduren geht.
- ODCIAggregate
  Dies ist das uns interessierende Interface. ODCIAggregate ermöglicht die Implementierung bedarfsspezifischer „Aggregate Functions" bzw. „Analytic Functions" (bei Beidem handelt es sich um die gleiche Implementierung. Die Implementierung ist je nach syntaktischer SQL-Nutzung aggregierend oder analytisch wirksam).

Die Data Cartridge Implementierung

Um eine analytische Data Cartridge (also eine Benutzer-definierte analytische Funktion) zu erstellen, ist bemerkenswert wenig zu tun. Es sind zwei „stored" Objekte anzulegen: Ein zum Interface ODCIAggregate passender „stored object type" (im Weiteren auch „Aggregate-Type" genannt) und eine „stored function" zu Deklarationszwecken.

Im nachfolgenden Listing [2] bauen wir zu Demonstrationszwecken eine möglichst einfache Oracle-Funktion nach: die Summenfunktion „SUM“. Den Nachbau bekommt den Namen „MY_SUM“, der zugehörige Aggregate-Type den Namen „MY_SUM_TYPE“.

Interfaces implementierende Object-Types müssen die vom Interfaces geforderten Funktionen signaturgetreu enthalten. Die zum Interface ODCIAggregate gehörenden Funktionen sind:

- ODCIAggregateINITIALIZE (mandatory)
- ODCIAggregateITERATE (mandatory)
- ODCIAggregateMERGE (mandatory)
- ODCIAggregateTERMINATE (mandatory)
- ODCIAggregateDELETE (optional)
- ODCIAggregateWRAPCONTEXT (optional)

In unserem Nachbau vernachlässigen wir zuerst einmal die Funktion ODCIAggregateDELETE. Die Funktion ODCIAggregateWRAPCONTEXT wird ausschließlich für die Nutzung externer Prozeduren benötigt und deshalb in diesem Vortrag nicht behandelt. Die nachfolgende Aggregate-Type Spezifikation enthält genau die „mandatory“-Funktionen. Das einzige Attribut „Summe“ dient zur Aggregation der in ODCIAggregateITERATE mittels Parameter „my_Value“ übergebenen Werte. Weitere Details dazu finden sie unter Link [2].

```
CREATE OR REPLACE TYPE my_sum_type IS OBJECT
( Summe NUMBER                                -- Aggregationsspeicher
, STATIC FUNCTION ODCIAggregateINITIALIZE
    ( object_instance IN OUT my_sum_type) RETURN NUMBER
, MEMBER FUNCTION ODCIAggregateITERATE
    ( self in out my_sum_type, my_Value IN NUMBER) RETURN NUMBER
, MEMBER FUNCTION ODCIAggregateMERGE
    ( self in out my_sum_type, other in my_sum_type) RETURN NUMBER
, MEMBER FUNCTION ODCIAggregateTERMINATE
    ( self in out my_sum_type, value OUT NUMBER
    , flags IN VARCHAR2           -- nur relevant für external Proz.
    ) RETURN NUMBER
);
```

Listing 2: Aggregate-Type Spezifikation für Data Cartridge “MY_SUM”

Auch der Aggregate-Type-Body stellt keine allzu großen Anforderungen. Jede Funktion kommt in unserem Nachbau mit einer einzigen Codezeile für die relevante Verarbeitung aus. Die Rückgabe `RETURN ODCIConst.SUCCESS` entspricht dem in C-Dialekten üblichen Rückgabe des Erfolgsstatus.

In der INITIALIZE-Funktion (eigentlich „odciaggregateinitalize“, aber wir kürzen im Weiteren ab) wird die Objektinstanz mit „Summe := 0“ instanziiert. Diese Funktion ist die einzige „Static Function“ und muss daher die Instanz „by Reference“ nach außen geben. Alle anderen Funktionen sind „member Functions“ und operieren auf dem – hier explizit angegebenen – Parameter „self“. Die Funktion ITERATE verrichtet die eigentliche Arbeit des Aufsummierens. Die Funktion TERMINATE liefert den aufsummierten Wert an die Zielzeile zurück. Falls Parallelisierung stattfindet, sorgt die Funktion MERGE für eine Zusammenführung der Teilsummen. Die parallele Objektinstanz wird dabei „by Reference“ über den Parameter „other“ herangeführt. Der Parameter „flag“ in der Funktion TERMINATE hat nur bei Nutzung externer Prozeduren Relevanz.

```
CREATE OR REPLACE TYPE BODY my_sum_type IS
  ---------------------------------------------
  STATIC FUNCTION ODCIAggregateINITIALIZE
    (object_instance IN OUT my_sum_type) RETURN NUMBER IS
  BEGIN
    object_instance := my_sum_type(Summe => 0);      -- Verarbeitung
    RETURN ODCIConst.SUCCESS;
  END;
  ---------------------------------------------
  MEMBER FUNCTION ODCIAggregateITERATE
    (self in out my_sum_type, my_Value IN NUMBER) RETURN NUMBER IS
  BEGIN
    Summe := Summe + my_Value;                        -- Verarbeitung
    RETURN ODCIConst.SUCCESS;
  END;
  ---------------------------------------------
 …
```

Listing 3 (Teil 1): Aggregate-Type Body für Data Cartridge "MY_SUM"

```
  …
  ---------------------------------------------
  MEMBER FUNCTION ODCIAggregateMERGE
    (self in out my_sum_type, other in my_sum_type) RETURN NUMBER IS
  BEGIN
    Summe := Summe + other.Summe;                     -- Verarbeitung
    RETURN ODCIConst.SUCCESS;
  END;
  ---------------------------------------------
  MEMBER FUNCTION ODCIAggregateTERMINATE
    (self in out my_sum_type, Out_Value OUT NUMBER) RETURN NUMBER IS
  BEGIN
    Out_Value := Summe;                               -- Verarbeitung
    RETURN ODCIConst.SUCCESS;
  END;
  ---------------------------------------------
END;
```

Listing 3 (Teil 2): Aggregate-Type Body für Data Cartridge "MY_SUM"

Jetzt fehlt nur noch die Funktion selber. Diese legen wir mit einem Create-Befehl wie in Listing [4] an. Das Bemerkenswerte an diesem Befehl ist, dass der sonst übliche Function-Body fehlt und stattdessen eine eigene Klausel – die Aggregate-Klausel – verwendet wird. Der Create-Befehl persistiert sozusagen die Zuordnung des Bezeichners „MY_SUM“ zum Aggregate-Type „MY_SUM_TYPE“. Die Klausel „parallel_enable“ erlaubt dem Server, Teilergebnisse in getrennten Aggregationskontexten zu ermitteln, die abschließend mit der Aggregate-Type Funktion ODCIAggregateMERGE zusammengeführt werden. Wenn eine Geschäftslogik parallele Verarbeitung ausschließt, dann ist die „parallel_enable“-Klausel wegzulassen. Die Aggregate-Type Funktion ODCIAggregateMERGE muss implementiert werden, aber der Verarbeitungsteil im Body kann dann leer bleiben.

```
CREATE OR REPLACE FUNCTION my_sum (val IN NUMBER) RETURN NUMBER
PARALLEL_ENABLE
AGGREGATE USING my_sum_type;
```

Listing 4: User-defined Aggregate Function "MY_SUM"

Parametrisierung und Debugging

Auffällig ist die Tatsache, dass die Interface-Funktion OCDIAggregateITERATE nur einen Parameter vorsieht. Aber der Parameter kann es in sich haben! Hier können Sie nicht nur die skalaren Parameter NUMBER, DATE und VARCHAR2 verwenden, sondern auch strukturierte oder amorphe Parameter. D.h., sie können als Parametertyp Object-Types, Collections und LOBs verwenden. Wenn sie also eine analytische Funktion mit vier Eingangsparametern gestalten wollen, dann schreiben Sie sich doch einfach einen Aggregatetype-Inputtype mit vier Attributen, der beim Aufruf der analytischen Funktion verwendet wird. Ein – sicherlich verkraftbarer – Wehrmutstropfen ist, dass es bei mehreren Eingangsparametern eben nicht ohne Input-Object-Type geht. Ein Beispiel für die Anwendung eines Input-Object-Types finden sie in den Listings [5] und [8].

Was mit den Input-Parametern geht, geht natürlich auch mit den Rückgabewerten. Auch hier ist die gleiche Vielfalt der Typisierungen machbar. Meine analytischen Funktionen implementiere ich gewöhnlich so, dass ich erst mal mit einem Aggregatetype-Outputtype arbeite. Über diesen liefere ich den Rückgabewert, der dann eines der Outputtype-Attribute ist. Gleichzeitig lege ich immer auch ein zweites Outputtype-Attribut vom Typ Varchar2(100) oder vom Typ Collection of Varchar2(100) mit an. Über dieses zweite Outputtype-Attribut kann ich dann Debugging-Texte oder ganze Text-Listen ausgeben. Im Listing [5] wird „MY_SQL" mit Object-Type-Rückgabeparameter deklariert. Ein entsprechendes Select-Ergebnis – so wie es in SQL*Plus auftauchen könnte – finden Sie im Listing [6]. Ein solches Ergebnis entsteht, wenn der implementierte Aggregatetype einen neu auftauchenden Funktionstitel in eine neue Collection-Zeile schreibt oder bei Funktionstitel-Wiederholungen den „Counter" der letzten Collection-Zeile hochgezählt.

```
CREATE OR REPLACE TYPE my_sum_info_type IS OBJECT
( text VARCHAR2(100), counter NUMBER );

CREATE OR REPLACE TYPE my_sum_info_col IS TABLE OF my_sum_info_type;

CREATE OR REPLACE TYPE my_sum_output_type IS OBJECT
( value NUMBER, info my_sum_info_col );

CREATE OR REPLACE TYPE my_sum_type IS OBJECT
…
MEMBER FUNCTION ODCIAggregateTERMINATE
( SELF IN OUT my_sum_type
, returnvalue OUT my_sum_output_type                    -- !!!
, FLAGS IN VARCHAR2 ) RETURN NUMBER
… );

CREATE OR REPLACE FUNCTION my_sum (input IN NUMBER)
  RETURN my_sum_output_type                             -- !!!
  PARALLEL_ENABLE AGGREGATE USING my_sum_type;
```

Listing 5: User-defined Aggregate Function "MY_SUM" mit strukturiertem Rückgabeparameter

```
select my_sum(my_sum_input_type(value)) from demo_tab;

MY_SUM(VALUE, INFO(TEXT, COUNTER))
MY_SUM_OUTPUT_TYPE(1250025000,
MY_SUM_INFO_COL(MY_SUM_INFO_TYPE('initialize', 1),
MY_SUM_INFO_TYPE('iterate', 50000),
MY_SUM_INFO_TYPE('merge', 3),
MY_SUM_INFO_TYPE('terminate', 1)))

1 row selected.
```

Listing 6: Debugging von "MY_SUM" mit strukturiertem Rückgabeparameter

Performance

Die Performance einer Data-Cartridge MY_SUM ist schlechter als der Oracle-eigenen Summenfunktion. Die Laufzeit einer Summierung einer numerischen Spalte einer Index-freien Tabelle mit 1.000.000 Datensätzen mit SUM( ) ist ca. 10-mal so schnell wie die Summierung mit der Data-Cartridge MY_SUM( ). Dies liegt im Wesentlichen an der Tatsache, dass mit jedem Funktionsaufruf der Data-Cartridge-Interface-Funktionen ein Kontextswitch zwischen SQL-Engine und PLSQL-Engine durchgeführt wird. Demgegenüber sind zentrale aggregierende Funktionen wie SUM u.ä. hochgradig in der SQL-Engine integriert.

Wird eine Data-Cartridge analytisch mit voranschreitender Windowing-Startzeile genutzt (also nicht: … ROWS between UNBOUNDED PRECEDING and …) dann bringt die Implementierung der optionalen Funktion ODCIAggregateDELETE deutliche Vorteile. Oracle ist dann in der Lage, Aggregationskontexte zwecks Wiederverwendung zu kopieren und in der Kopie nicht mehr benötigte Eingangsparameter mittels DELETE-Funktion wieder heraus zurechnen. Ohne DELETE-Funktion müsste der Aggregationskontext der folgenden Zeile komplett neu berechnet werden.

Sollen in einem SQL-Statement zahlreiche Ergebnisspalten mit Data-Cartridges berechnet werden, kann es sinnvoll sein, statt dem vielfachen Einsatz skalarer returnierender Data-Cartridges eine Data-Cartridge mit einem Rückgabe-Object-Type zu schreiben, mittels dem die vielfachen Ergebnisse über einen Kontext-Switch zurückgegeben werden.

Nützliche Data-Cartridges für Data-Warehouses

Zwei der Aggregate-Data-Cartridges, die in meinen Data-Warehouse-Projekten fast immer zum Einsatz kommen, sind die beiden Funktionen DISTLISTAGG( ) und HISTCHECK( ). Diese beiden Funktionen werden folgend kurz beschrieben und Code-Teile vorgestellt.

Data-Cartridge DISTLISTAGG( )

Oracle stellt seit dem Release 11R2 die Funktion LISTAGG bereit, die Spaltenwerte zu Komma-separierten Listen konkatenieren soll. Dies ist ein immer wieder nachgefragtes generisches Feature, welches dankbarerweise von Oracle nachgeliefert wurde. Leider hat die Funktion einige Nachteile. Einer der Nachteile verbietet geradezu den Einsatz von LISTAGG.

- Die Funktion kann nur mit einer proprietären Syntax genutzt werden (verschmerzbar).
- Die Funktion liefert immer vollständige, jedoch keine distinkte Wertelisten. Distinkte Listen sind aber im DWH-Umfeld der häufigste benötigte Fall, wenn es um Listenerstellung geht.
- Die Funktion läuft bei großen Query-Partitions auf einen Variablen-Überlauf !!!! Scheinbar wurde keine Substring-Mimik etabliert, mit der ein Stringvariablen-Überlauf verhindert werden könnte. Meinerachtens muss diese Funktion fehlerbereinigt werden, bevor sie mit gutem Gewissen einsetzbar ist.

Für die Konkatenation distinkter Listen eignet sich dagegen DISTLISTAGG( ) deren wesentlicher Code-Teil „ODCIAggregateITERATE" im Listing [7] vorgestellt wird. Der Aggregate-Type arbeitet mit den String-Attributen „ErgebnisString" (varchar2(4000)), „Unmoeglichkeitswert" und „Trenner". Der Member Function ODCIAggregateITERATE werden per Object-Type „distlistagg_input_type" die zu konkatenierende Spalte „String" und optional Unmöglichkeitwert (Default sind zwei Doppelpunkte) und Trenner (Default ist ein Komma) übergeben. Die ODCIAggregateMERGE-Funktion hat keinen wirksamen Code. Die Funktion ODCIAggregateDELETE wird nicht implementiert. Die Stored Function DISTLISTAGG wird nicht mit „parallel_enable" ausgestattet.

```
CREATE OR REPLACE TYPE BODY distlistagg_type IS
  …
  -----------------------------------------------------------------
  MEMBER FUNCTION ODCIAggregateITERATE
  ( SELF IN OUT distlistagg_type, input IN distlistagg_input_type
  ) RETURN NUMBER IS
  BEGIN
    --      -- initiale Übernahme von Unmöglichkeitswert und Trenner
    --      -- beim Ersten Aufruf von ITERATE.
    if SELF.trenner is null then
      SELF.unmoeglichkeitswert := input.unmoeglichkeitswert;
      SELF.trenner             := input.trenner;
      SELF.ergebnisstring      := input.unmoeglichkeitswert;
    end if;
    --
    --      -- Konkateniere, wenn Wert noch nicht in Liste.
    IF INSTR( SELF.ergebnisstring
            , input.unmoeglichkeitswert
           || input.string
           || input.unmoeglichkeitswert) = 0 THEN
      SELF.ergebnisstring := SUBSTR(SELF.ergebnisstring
        || input.value || input.unmoeglichkeitswert, 1, 4000);
    END IF;
    --
    RETURN ODCICONST.SUCCESS;
  END;
  -----------------------------------------------------------------
  …
```

Listing 7: Die Aggregate-Type Funktion ODCIAggregateITERATE des Data-Cartridge LISTAGG.

Data-Cartridge HISTCHECK( )

Diese Funktion dient der Überprüfung historisierter Datenquellen. Sie wird in DWH-Projekten gerne in OWB-Mappings genutzt, da sie sich ausgezeichnet in einem Expression-Operator etablieren – und nachgelagert filtern lässt. Die Funktion liefert positive Werte für fehlerfreie – und negative Werte für fehlerhafte Historisierungen. Die Fehlertyp-beschreibende Ausprägung des negativen Werts kann zur Fehleranalyse genutzt werden. Die Fehleranalyse erfolgt nicht nur satzintern (z.B. muss gelten: gueltig_von < gueltig_bis), sondern aufgrund des analytischen Vorgehens auch satzübergreifend (z.B.: Wenn Vorgaengersatz vorhanden und lückenfreie Historisierung gefordert, dann muss gelten: aktueller_satz.gueltig_bis = vorgaenger_satz.gueltig_von).

Tatsächlich gibt es kein einheitliches Vorgehen bei der Historisierung. In jeden Projekt stößt man auf eigene Historisierungsregeln, die eine bedarfsspezifische Implementierung einer „HISTCHECK“ Data-Cartridge erfordern. Dennoch macht hier die Präsentation der wichtigen Codeabschnitte beispielgebend Sinn.

Im SQL-Statement werden per Object-Type „histcheck_input_type“ die Spalten „gueltig_von“ und „gueltig_bis“ der zu überprüfenden historisierten Tabelle und das im Projekt definierte Unendlichkeitsdatum (z.B. „31.12.2999“) übergeben. In vielen Projekten macht es Sinn, den jüngsten

Historisierungssatz als „valide“ anzunehmen und die älteren Sätze gegen die Jüngeren abzugleichen. Endsprechend (s.a. Listing [9] ist die Query-Partition mit „gueltig_bis desc, gueltig_von desc“ zu sortieren.

```
CREATE OR REPLACE TYPE histcheck_input_type IS OBJECT
( gueltig_von DATE, gueltig_bis DATE, unendlich DATE );

CREATE    OR    REPLACE    TYPE    BODY    histcheck_type    IS    OBJECT
( prev_input    histcheck_input_type
, row_num       NUMBER
, row_is_valid  NUMBER
, STATIC FUNCTION ODCIAggregate…
…
);

CREATE OR REPLACE FUNCTION histcheck (input IN histcheck_input_type)
return NUMBER – kein ”parallel_enable” !
AGGREGATE USING histcheck_type;

CREATE     OR     REPLACE     TYPE     BODY     histcheck_type     IS
  -----------------------------------------------------------------
  STATIC FUNCTION ODCIAggregateINITIALIZE
  ( inst IN OUT histcheck_type ) RETURN NUMBER IS
  BEGIN
    inst := histcheck_type
             ( prev_input   => HISTCHECK_INPUT_TYPE
                                ( gueltig_von     => to_date(null)
                                , gueltig_bis     => to_date(null)
                                , unendlich       => to_date(null)
                                )
             , row_num      => 0
             , row_is_valid => 0
             );
    RETURN ODCIConst.SUCCESS;
  END;
  -----------------------------------------------------------------
  MEMBER FUNCTION ODCIAggregateITERATE
  ( SELF IN OUT histcheck_type, input IN histcheck_input_type
  ) RETURN NUMBER IS
    v_row_is_valid NUMBER := 1;
  BEGIN
    if (v_row_is_valid > 0
    and input.gueltig_von > input.gueltig_bis)
    then v_row_is_valid := -91;
    end if;
    --
    if (v_row_is_valid > 0 and row_num = 0
    and input.gueltig_bis > input.unendlich)
    then v_row_is_valid := -92;
    end if;
```

```
    --
    if (v_row_is_valid > 0 and row_num > 0
    and input.gueltig_bis > prev_input.gueltig_von)
    then v_row_is_valid := -93;
    end if;
    --
    if (v_row_is_valid > 0
    and input.gueltig_von = input.gueltig_bis)
    then v_row_is_valid := -94;
    end if;    --
    --
    if v_row_is_valid > 0 then
      self.prev_input := input;
      self.row_num := self.row_num + 1;
    end if;
    self.row_is_valid := v_row_is_valid;
    --
    RETURN ODCICONST.SUCCESS;
  END;
  ------------------------------------------------------------------
  MEMBER FUNCTION ODCIAggregateMERGE
  ( SELF IN OUT histcheck_type, other_context IN histcheck_type )
  RETURN NUMBER IS
  BEGIN -- Funktion wird nicht aufgerufen.
    RETURN ODCIConst.SUCCESS;
  END;
  ------------------------------------------------------------------
  MEMBER FUNCTION ODCIAggregateTERMINATE
  ( SELF IN OUT histcheck_type
  , returnvalue OUT NUMBER, flags IN VARCHAR2 ) RETURN NUMBER IS
  BEGIN
    returnvalue := row_is_valid;
    RETURN ODCIConst.SUCCESS;
  END;
  ------------------------------------------------------------------
END;
```

Listing 8: Wesentliche Auszüge aus der `Data-Cartridge HISTCHECK`.

```
SELECT histcheck
       ( histcheck_input_type
           ( gueltig_von
           , gueltig_bis
           , to_date('31.12.2999', 'dd.mm.yyyy') -- unendlich
       )) OVER
       ( PARTITION BY artikel_nr
         ORDER BY gueltig_bis desc, gueltig_von desc
       ) AS histcheck_result
FROM   stage_artikel … ;
```

Listing 9: SQL-Nutzung der Funktion `HISTCHECK`.

Nutzung von Data-Cartridges im OWB

Um Stored Objekte in OWB-Mappings nutzen zu können, müssen die Data-Cartridge Objekte - wie andere Stored Objekte auch - in den OWB importiert werden. Dies funktioniert reibungslos für die Object-Types, nicht aber für die Stored Function mit der Aggregate-Klausel. Scheinbar stört sich der OWB an dieser Klausel oder aber am fehlenden Function-Body.

Als Workaround bietet sich hier an, die Funktion händig mit „new Function" unter Transformations → Functions anzulegen, und die In- und Out-Parameter zu definieren. Da weder ein Body definiert werden soll, noch Einstellungsmöglichkeiten für die Aggregate-Klausel existieren, konfiguriert man die Funktion besser auf „nicht deployable".

Danach ist die Data-Cartridge für analytische Nutzung wie in Abbildung [1] in Expression-Operatoren validierbar und nachgelagert als Filter nutzbar. Wenn die Funktion nicht händig angelegt wurde, wird die Validierung im Expression-Editor des Operators fehlschlagen. Nichtsdestotrotz lässt sich das zugehörige Mapping deployen und ohne Laufzeitfehler ausführen.

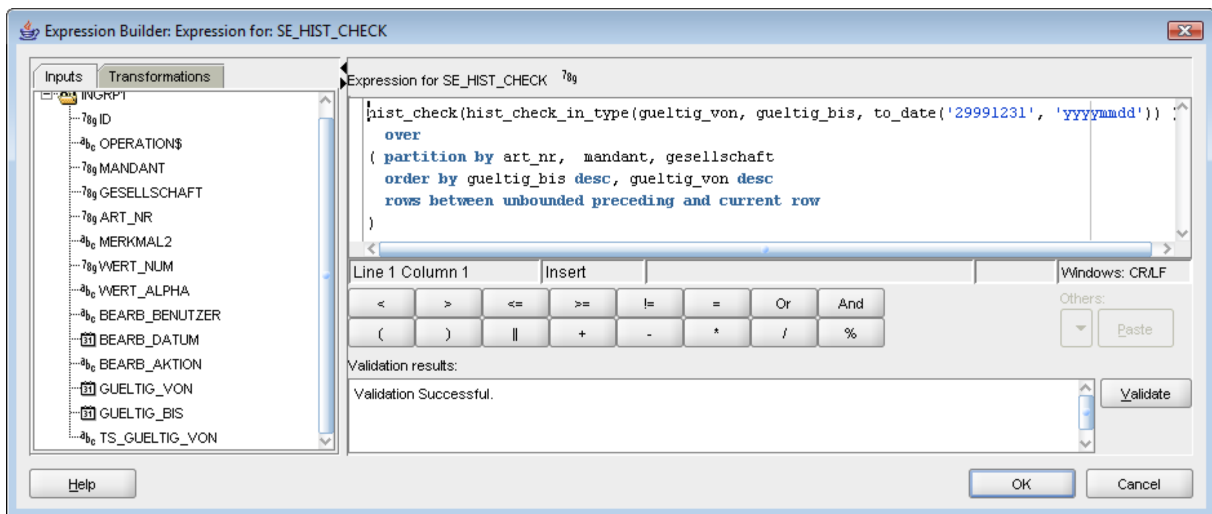

Abb. 1: analytische HISTCHECK-Nutzung im Expression-Operator eines OWB-Mappings.

Für eine aggregierende Nutzung kann die Data-Cartridge wie Abbildung [2] in Aggregate-Operatoren eingesetzt werden. Auch hier erkennt der OWB die Data-Cartridge nicht als nutzbare Aggregate-Funktion und bietet sie nicht in der Funktionsliste an (In der Abbildung [2] vermisst man beispielsweise die im Projekt vorhandene Funktion MY_SQL in der alphabetisch sortierten Funktionsliste). Als Workaround kann man die den Ausdruck wie in der Abbildung direkt im Expression-Editor des Operators erfassen.

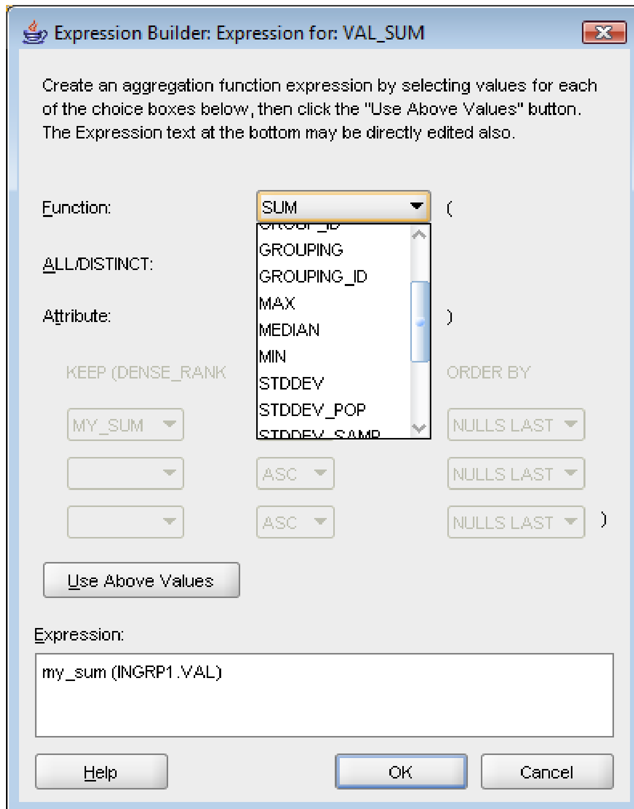

Abb. 2: aggregierende MY_SUM-Nutzung im Aggregation-Operator eines OWB-Mappings.

Resümee

Aggregate-Data-Cartridges mit ihren zeilenübergreifenden analytischen Möglichkeiten stellen ein mächtiges Werkzeug für SQL-Statements in den zunehmend komplexer werdenden Datawarehouse-Projekten dar. Mit der standardisierten Syntax für analytische Anwendung ist eine gute Wartbarkeit und hohe Akzeptanz im Entwicklungsteam zu erwarten. Die Möglichkeiten der Vereinfachung sonst komplexer Lösungen machen den Performance-Impact durch Kontextswitches in den meisten Fällen wett. Unter Berücksichtigung der heute noch notwendigen Workarounds lassen sich aggregierend und bzw. analytisch arbeitende Data-Cartridges gewinnbringend in ETL-Werkzeugen wie z.B. dem OWB einsetzen.

Nützliche Links

Link 1: *http://download.oracle.com/docs/cd/E11882_01/server.112/e17118/functions004.htm#i81407*
Link 2: *http://download.oracle.com/docs/cd/E11882_01/appdev.112/e10765/ext_agg_ref.htm#i76772*

Kontaktadresse:

Gero Knapstein
OPITZ CONSULTING Gummersbach GmbH
Kirchstraße 6

D-51647 Gummersbach

Telefon:	+49 (0) 2261 6001-0
Fax:	+49 (0) 2261 6001-4200
E-Mail	gero.knapstein@opitz-consulting.com
Internet:	www.opitz-consulting.com