Bessere Datawarehouses mit Pipelined Table Functions

Gero Knapstein
OPITZ CONSULTING GmbH
Gummersbach

Schlüsselworte:

Pipelined Table Function, Datawarehouse, Historisierung, Parallelisierung, Datenstrom

Einleitung

Wenn es um Massendatenbewegungen im ETL-Bereich geht, sind reine SQL-DML-Statements immer das Beste.

Wirklich immer?

Nein!

Ein kleiner Anwendungsbereich wehrt sich hartnäckig gegen performante Abwicklung, Wartbarkeit und nachvollziehbare Implementierung.

Es sind dies die Anwendungsbereiche, die nur mit Mühe mit 4GL-Konstrukten der Oracle-SQL-Sprache, aber mit Leichtigkeit mit 3GL-Sprachen wie z.B. PL/SQL lösbar sind. Die folgenden vier Beispiele spiegeln möglicherweise die Anforderungen auch in Ihren Projekten wieder.

Das Near-Realtime Datawarehouse

Anforderung

In Nachtverarbeitungsläufen ist EM und Datamart zu bewirtschaften. Als Hauptdatenquelle dienen drei denormalisierte Tabellen eines ODS-Reportsystems. Eines der drei Tabellen – die Bestandstabelle – wird über OLTP-Verfahren tagsüber modifiziert. Die Modifikationen sollen möglichst zeitnah (< 10 Min.) und in einer Transaktion in das Datawarehouse eingebracht und in Ad-Hoc-Report- bzw. -Abfrage-Tools zur Anzeige gebracht werden.

Motivation

Die Analyse der Report-Anforderungen führte zu einer komplexen Entscheidungstabelle, nach der je nach Fall individuell gestaltete Faktensätze zusammenzustellen waren, um allen Filter- und Aggregationsanforderungen gerecht zu werden. Der Kunde wünschte überdies eine in PLSQL implementierte Lösung, unter anderem wegen der in geschlossenen Transaktionen durchzuführenden Near-Realtime-Datenbewegungen.

Realisierter Lösungsansatz

Die wenigen Quelltabellen wurden in SQL objekt-relational miteinander gejoint und als eine einzige Query aus Rows (Mastersatz) mit Nested Table Spalten (Detailsätze) dem ETL-Prozess (PLSQL-Package) zugeführt. Die bulkweise selektierten Daten wurden gemäß ihrer normalisierten Bestandteile zerlegt und in entsprechende Caches abgelegt. Für die Überführung der zerlegten Daten wurden diese

per „pipelined table function“ in Merge-Statements eingebracht und so den EM-, Dimensions- und Faktentabellen zugeführt. Es wurde ein Auswahl von Cache-Techniken entwickelt, um bestehende EM- bzw. Datamart-Daten mit den ankommenden Daten abgleichen zu können [s.a. DOAG-Vortrag „NearRealtime-Datawarehouse“ 2009]. Dieser „inside-PLSQL-Abgleich“ diente der Reduzierung der über Table-Functions auszugebenden Satzmengen. Zum einen wurde damit ein performanter Datentransport großer Datenmengen über Nacht, als auch eine hochperformante Near-Realtime-Behandlung erreicht. Als Latenzzeit zwischen OLTP-Erfassung im Vorsystem bis zum Abschluß aller DMLs gegen EM und Datamart wurden Zeiten zwischen 2 und 10 Sekunden erreicht. Zur Entkopplung der konkurrierenden OLTP-Vorgänge und dem exklusiven ETL-Prozess wurde eine Advanced Queue eingesetzt.

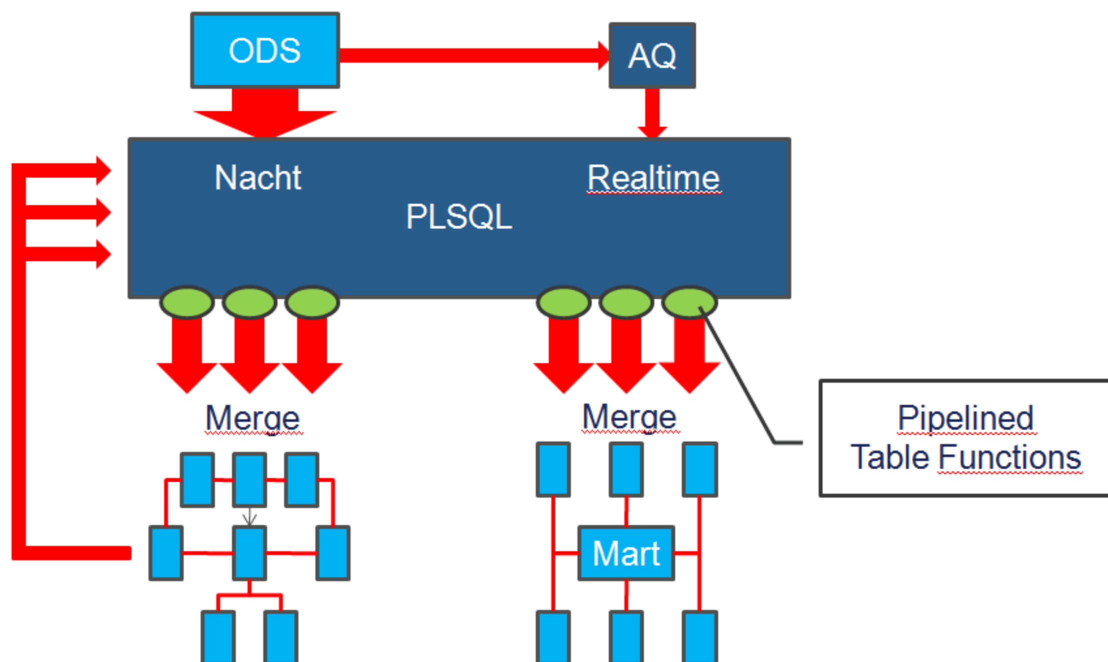

Abbildung 1: Monolithische Transformation in PLSQL; Datenbereitstellung für Merge mit Table Function

ETL-Bewirtschaftung mit OWB-Mappings ohne Transformation

Anforderung

Mit vollständiger Bewirtschaftung über Nacht und partieller Bewirtschaftung in Tagesläufen ist ein Datawarehouse nach klassischem Muster aufzubauen. Auftraggeber ist das Unternehmens-Controlling. Aus Kostengründen und als Maßnahme zur Qualitätsverbesserung des ERP-Vorsystems sollen Datenbereinigungen weitestgehend im Vorsystem selbst vorgenommen werden. Die Historisierung soll unverändert übernommen werden. Realisiert werden sollte der ETL-Prozess mit OWB 11.2 Standard-Edition.

Motivation

Es zeigte sich schnell, dass die durch Datenanalyse gestellten Bereinigungsanforderungen an das Vorsystem nur unzureichend von der IT-Abteilung erfüllt werden konnten. Für die über OWB-

Mappings in Fehlertabellen ausgesonderten Daten wurde eine automatische Fehleranalyse entwickelt. Ein großer Teil der Daten musste jedoch nachträglich einer Transformation zugeführt werden.

Realisierter Lösungsansatz

Als kostengünstige und gleichzeitig flexible Variante wurde ein Metadaten-basierter Generator entwickelt. Dieser stellt ein View-System mit (bei Bedarf) eingebundenen PLSQL-Funktionen zur Filterung bzw. Transformation der Stage-Daten bereit. Die anfänglich entwickelten analytischen Funktionen [s.a. DOAG-Vortrag „analytische Funktionen für Datawarehouses selbst gemacht“, 2010] wurden mit zunehmender Anforderungskomplexität und zunehmender Häufigkeit von Kontext-Switches durch Pipelined Table Functions ersetzt.

Eine Besonderheit dieser Table Functions ist, dass statt der Table-Function-„Cluster“-Klausel eine Aggregation mit Aufbereitung der Detailinformationen in sortierte Collections erfolgt, bevor die Datenquelle dem Table-Function-Ref-Cursor-Parameter [s.a. späteres Kapitel] zugeführt wird.

```
CREATE VIEW SE_ARTIKEL_BASIS_V
AS
   SELECT *
     FROM TABLE (SE_ARTIKEL_BASIS_P.SE_ARTIKEL_BASIS
       (CURSOR (with my_source AS (SELECT ID,... FROM SE_ARTIKEL_BASIS_B WHERE <uk-cols not null>
                            UNION ALL SELECT ID,... FROM FE_ARTIKEL_BASIS   WHERE <uk-cols not null>
                                   )
SELECT MAX(ART_NR) AS ART_NR ,MAX(MANDANT) AS MANDANT ,MAX(WG) AS WG
     , CAST
         ( MULTISET
             ( SELECT SE_ARTIKEL_BASIS_T(ID,...)
                 FROM my_source t2
                WHERE t1.ART_NR=t2.ART_NR AND t1.MANDANT=t2.MANDANT AND t1.WG=t2.WG
                ORDER BY t2.gueltig_von, t2.gueltig_bis
             ) AS SE_ARTIKEL_BASIS_N
         ) AS my_collection
  FROM my_source t1
 GROUP BY ART_NR,MANDANT,WG
), p_set_trunc_history_on => 1))
```

Abbildung 2: Objektrelationale Aufbereitung der Daten vor Übergabe an Pipelined Table Function

Pipelined Table Function erzeugt Zeitreihen für OWB-Mappings

Anforderung

Die für das Datawarehouse notwendigen ETL-Prozesse sollen (soweit nicht generisch gelöst) aus Dokumentationsgründen vollständig als OWB-Mappings implementiert werden. Prozedurale Prozesse oder Skripte, die nicht in OWB-Mappings eingebunden sind, sollen vermieden werden.

Motivation

Mit Table-Function-Operatoren lassen sich Daten erzeugende PLSQL-Prozesse einfach und kostengünstig in OWB-Mappings einbinden.

Realisierter Lösungsansatz

Alle zu erstellenden Datenmengen ohne Quell-Datenbasis (z.B. Zeitreihen) wurden mit parametrisierbaren Pipelined Table Functions implementiert und mit Table Function Operatoren in OWB-Mappings eingebunden.

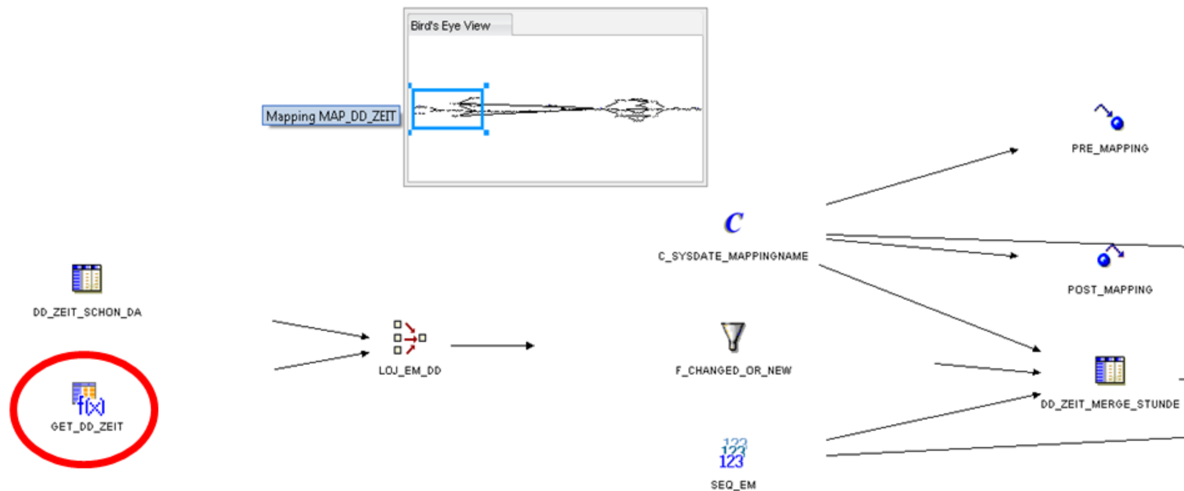

Abbildung 3: Integration einer Pipelined Table Function in ein OWB-Mapping

Table-Functions als Package-API zu Cursor- und Array-Daten

Anforderung
Alle Packages sollen einen regelmäßigen Unit-Test unterzogen werden

Motivation
Neben der API-Überprüfung durch Unit-Tests ergeben sich häufig Fragen über den Inhalt von Package-Variablen und Package-Collections. Erstere sind recht einfach mit verschiedenen Methoden zu debuggen. Aber erst die Table Functions eröffnen eine einfach handhabbare Präsentation der Collectioninhalte.

Realisierter Lösungsansatz
Record- und Collection-Types werden grundsätzlich in der Spezifikation deklariert. Es wird zu jeder im Package Body deklarierten Collection-Variable eine einfach gestaltete Table Function implementiert. Die Table Function APIs werden in Unit-Tests (UTplsql) genutzt.

Table Functions im PLSQL- und SQL-Kontext
Bis zur aktuellen Version Oracle 11.2 werden SQL-Statements in SQL-Engines und PLSQL-Code in PLSQL-Engines ausgeführt. Die Kommunikation zwischen diesen beiden Welten ist möglich, aber aufwändig („Kontext-Switch") und bremst SQL-Statements mit Aufrufen von user-defined PLSQL-Objekten deutlich aus.

Oracle hat eine Reihe von Kommunikationsformen zwischen SQL- und PLSQL-Kontext entwickelt. Neben der „Single Row Function" kennt Oracle auch die Funktionsart „Table Function" und „pipelined Table Function". Diese liefern pro Kontext-Switch jeweils eine vollständige Ergebniszeile (Pipelined Table Function) oder sogar ein fertiges Ergebniszeilenset (Table Function).

„Single Row Functions" klingt nach Bereitstellung fertiger Ergebniszeilen. Tatsächlich werden aber nur Einzelwerte als „Bestandteil" von Ergebniszeilen geliefert. Jede Bereitstellung bedeutet ein Kontext-Switch. Werden mehrere Funktionswerte pro Ergebniszeile benötigt, kostet dies ein „Mehrfaches" an Kontext-Switches pro Zeile. Ist die Kardinalität der Rückgabewerte klein und die

Funktion als „deterministic“ ausgewiesen, reduziert sich die Zahl der Kontext-Switche immerhin auf Kardinalitätsgröße.

Der kleine Unterschied „pipelined“

Für Anwendungen, deren vielspaltige Ergebnismengen überwiegend prozedural ermittelt werden sollen, empfehlen sich Lösungen mit Table Functions. Wie eine Table Function erstellt und benutzt wird, zeigen die nachfolgenden Erläuterungen und Programmbeispiele. Zuerst sollte aber noch der Unterschied zwischen „Table Function“ und „Pipelined Table Function“ geklärt werden.

Eine „Table Function“ returniert eine fertig aufbereitete Collection (Collection Type Instance) mit einer einzigen Aktion. Eine „pipelined Table Function“ liefert die Collection-Objekte (Object Type Instanzen) Zeile für Zeile. Und dies so, dass sie nach FIFO-Manier unmittelbar weiterverarbeitet werden können.

Wenn es um die Belieferung eines SQL-Kontextes geht, lassen sich die Wirkungsweisen wie folgt unterscheiden:

Bei großen Collections dauert es innerhalb der „Table Function“ eine entsprechend lange Zeit, bis die Collection aufbereitet ist. Mit einem einzigen abschließenden Kontext-Switch wird die vollständige Collection an den SQL-Kontext übergeben. Performanter geht’s nicht. Aber der SQL-Kontext-Thread muss vorher auf die Collection-Fertigstellung warten. Außerdem braucht es vor der Übergabe einen entsprechend großen Arbeitsspeicherbereich in der PGA.

Eine Pipelined Table Function kann jede einmal fertiggestellte Collection-Zeile unmittelbar zurückgeben. Es wird erst gar keine funktionsinterne Collection aufgebaut. Für jede Zeile wird allerdings separat ein Kontext-Switch benötigt, aber der SQL-Kontext-Thread hat faktisch keine Wartezeit auf verarbeitbare Informationen. Soweit sie unmittelbar weiterverarbeitet werden, brauchen Zwischenergebnisse nicht gecached zu werden.

Wie erstellt man Table Functions und Pipelined Table Functions?

Table Functions und pipelined Table Functions liefern strukturierte Table Collections zurück. Als Grundlage wird ein „Stored Object Type“ und ein darauf basierender „Stored Collection Type“ benötigt. Beide Typen können entweder mittels CREATE TYPE Statement oder als PLSQL-Typen in Package Spezifikations implementiert werden. Letztere werden implizit als Stored Types mit vom System vergebenen Namen umgesetzt.

Das (hier vereinfachte) Package PCK_ZEIT (letzte Abbildung) liefert Ergebnismengen als Grundlage für Zeit-Dimensionen. Jede Ergebnismenge besteht aus Zeitpunkt-Sätzen jeweils zwischen einem vorbestimmten Start- und Endzeitpunkt.

Das Package enthält jeweils eine „en-Bloc“ - und eine „pipelined“ Table Function Lösung. Die Funktion GET_ZEITPUNKT liefert für beide Table Functions die fertigen Collectionzeilen. Die Prozedur SET_BEHAVIOUR setzt Zeitunter- und –obergrenze, sowie Zeitinterval zwischen den Zeitpunkten.

Die pipelined Table Function GET_ZEIT_PIPELINED unterscheidet sich von der Function GET_ZEIT_EN_BLOC durch folgende Besonderheiten:

- Das Schlüsselwort „PIPELINED“ als Abschluss der Return-Klausel in der Funktionssignatur.
- Die Anweisung „PIPE ROW (<record- oder object-type-variable>);“
- Der Abschluss der Funktion mit „RETURN;“ ohne Rückgabewert.

Die nach dem Package-Code gelisteten Select-Statements zeigen, wie Table Functions in den SQL-Kontext eingebunden werden. Die Funktion TABLE(<[pipelined]table function>) formt die von der Table Function gelieferte Collection in eine Ergebnismenge um.

Pipelined Table Function im Datenstrom

Die Funktion GET_ZEIT_PIPELINED könnte in einem Datawarehouse als Datenquelle dienen. in einer ETL-Strecke sind aber auch Table Functions interessant, die man als Filter oder Pipe-Element in einem Datenstrom einsetzen kann. Die neuen Funktionen ADD_TAG_IM_JAHR und ADD_SAISON sind solche Funktionen. Ihre Signaturen gleichen der Funktion GET_ZEIT_PIPELINED, haben aber zusätzlich noch typisierte Ref-Cursor Parameter. Der jeweilige Anweisungsblock ist eine typische Schleifenverarbeitung auf einem geöffneten Ref-Cursor – in ADD_TAG_IM_JAHR mit Einzelsatzverarbeitung, in ADD_SAISON mit Bulk-Fetches. Der gefetchte Cursor-Record wird jeweils um eine weitere Spaltenbefüllung „angereichert“ und sofort per PIPE ROW –Anweisung weitergegeben.

Wie werden nun die Funktionen zu einem Datenstrom miteinander verkettet? Das Prinzip entspricht dem Schema der folgenden Select-Statements und Abbildung. Die Funktion CURSOR wandelt eine Ergebnismenge in einen geöffneten Ref-Cursor um.

```
Select * from TABLE( table_function( CURSOR( select * from … )));

select * from
TABLE (pck_zeit.add_saison (
  ( CURSOR (select * from table (pck_zeit.add_tag_im_jahr
      ( CURSOR (select * from table (pck_zeit.get_zeit_pipelined
))))))));
```

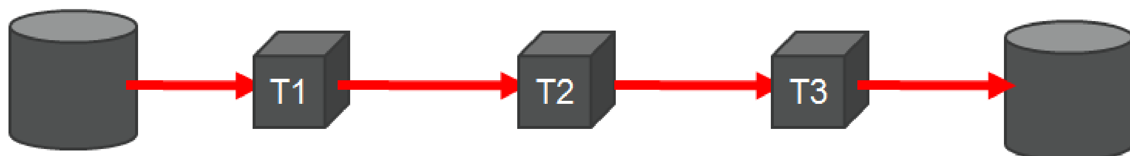

Abbildung 4: Datenstrom mit drei pipelined Table Functions

Parallelisierbare Pipelined Table Functions

Die Signaturklausel PIPELINED kann erweitert werden mit PARALLEL_ENABLE oder noch präziser mit „PARALLEL_ENABLE (PARTITION <refcursor> BY ANY)“ , „… BY RANGE <columnlist>)“ oder „… BY HASH <columnlist>)“. Ist die vom Ref-Cursor eingelesene Datenquelle parallelisierbar, wird auch die Ergebnismenge der Funktion „TABLE (<pipelined-Table-Function>)“ parallel ermittelt. Es lassen sich mehrere Ref-Cursor-Parameter abgeben, allerdings ist nur einer partitionierbar.

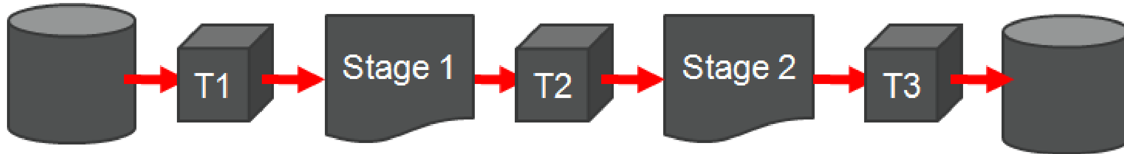

Abbildung 5: Datenstrom mit mehreren Stagebereichen

Die vorangegangene Abbildung zeigt einen in Teilprozesse zerlegten Geschäftsprozess mit mehreren Stage-Bereichen. Der Prozess T2 kann die Daten aus Stage1 erst aufnehmen, wenn T1 die Datenablage in Stage1 abgeschlossen hat. Entsprechend entstehen an den Stage-Bereichen Wartezeiten, die mit dem Pipelined Function basierten Datenstrom vermieden werden können. Dem Performanceverlust durch Kontext-Switches kann man, wie in der nächsten Abbildung, mit einer entsprechenden Parallelisierung begegnen.

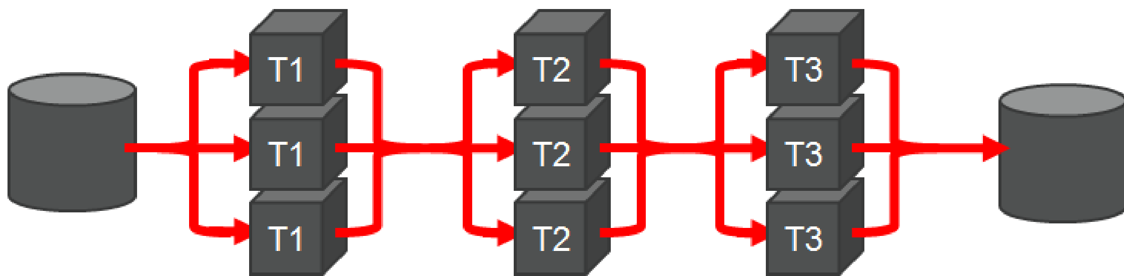

Abbildung 6: parallelisierter Datenstrom mit pipelined Table Functions

Anwendungsfelder parallelisierter Pipelined Datenströme
Spannende Möglichkeiten eröffnen sich in Anwendungen, die Geschäftslogiken in PLSQL-Funktionen aufbereitet haben. Mit dynamisch aufgestellten SQL-Statements, in denen Funktionen als Pipe-Elemente eingeflochten werden, lassen sich mit kleinem Aufwand flexible Workflow-Netzpläne umsetzen. Advanced Queueing ist hier flexibler, aber auch deutlich aufwändiger zu implementieren.

Frontends mit „first-Rows"-Abfragen können von dem schnellen Datendurchsatz durch eine aufwändige Logikabfolge profitieren. Insbesondere in Reportingsystemen mit getrennten Teilverarbeitungen können verblüffende Performancegewinne erzielt werden, wenn Stage-Bereiche und deren Latenzzeiten durch kontinuierlich fließende Datenströme in Pipeline-Systeme ersetzt werden.

Ausblick
Zur ernsthaften Konkurrenz reiner SQL-Massendatenverarbeitung werden Pipelined Table Functions dann, wenn Oracle in einer zukünftigen Version den „PIPE ROW ()" –Befehl um Bulk-Mechanismen erweitert. Die Zahl der Kontext-Switches würde dann um den Faktor der Bulkgrößen sinken.

```
create or replace package pck_zeit is
  --
  type t_zeit_rec is record
  ( minute number(2)
  , stunde number(2)
  , datum date
```

```
  , uhrzeit date
  , saison varchar2(4)
  , tag_im_jahr number
  );
  type t_zeit_col is table of t_zeit_rec;
  type t_zeit_refcurs is ref cursor return t_zeit_rec;
  --
  procedure set_behaviour
  (p_start date, p_ende date, p_interval number);
  function get_zeit_en_bloc return t_zeit_col;
  function get_zeit_pipelined
  return t_zeit_col pipelined parallel_enable;
  function add_tag_im_jahr
  (curs in t_zeit_refcurs) return t_zeit_col pipelined;
  function add_saison
  (curs in t_zeit_refcurs) return t_zeit_col pipelined;
end;
```

Abbildung7: Package Spezifikation PCK_ZEIT mit vier Table-Functions

```
create or replace package body pck_zeit is
  ----------------------------------------------------------------
  type t_maske_rec is record
  ( datum  varchar2(8)  := 'yyyymmdd'
  , stunde varchar2(12) := 'hh24'
  , minute varchar2(14) := 'mi'
  , tag_im_jahr varchar2(3) := 'ddd'
  );
  k_maske t_maske_rec;
  v_zeit_start date := to_date('20110101', k_maske.datum);
  v_zeit_ende date  := to_date('20111231', k_maske.datum);
  v_zeitpunkt date  := v_zeit_start;
  v_interval number := 1;
  ----------------------------------------------------------------
  function get_zeitpunkt return t_zeit_rec is
    rec t_zeit_rec;
  begin
    rec.minute := to_number(to_char(v_zeitpunkt, k_maske.minute));
    rec.stunde  := to_number(to_char(v_zeitpunkt, k_maske.stunde));
    rec.datum   := to_date(to_char(v_zeitpunkt, k_maske.datum )
                          ,k_maske.datum );
    rec.uhrzeit := v_zeitpunkt;
    --
    v_zeitpunkt := v_zeitpunkt + v_interval;
    --
    return rec;
  end;
  ----------------------------------------------------------------
  procedure set_behaviour
  (p_start date, p_ende date, p_interval number) is
```

```
begin
  v_zeit_start := p_start;
  v_zeit_ende  := p_ende ;
  v_interval   := p_interval  ;
end;
----------------------------------------------------------------
function get_zeit_en_bloc return t_zeit_col is
  v_zeit_col t_zeit_col := t_zeit_col();
  i number := 0;
begin
  v_zeitpunkt  := v_zeit_start;
  loop
    exit when v_zeitpunkt > v_zeit_ende;
    v_zeit_col.extend; i := i + 1;
    v_zeit_col(i) := get_zeitpunkt;
  end loop;
  return v_zeit_col;
end;
----------------------------------------------------------------
function get_zeit_pipelined return t_zeit_col
pipelined parallel_enable is
begin
  v_zeitpunkt  := v_zeit_start;
  loop
    exit when v_zeitpunkt > v_zeit_ende;
    pipe row (get_zeitpunkt);
  end loop;
  return;
end;
----------------------------------------------------------------
function add_tag_im_jahr
( curs in t_zeit_refcurs )
return t_zeit_col pipelined
parallel_enable (partition curs by any)
is
  rec t_zeit_rec;
begin
  loop
    fetch curs into rec;
    exit when curs%notfound;
    rec.tag_im_jahr := to_number(to_char(v_zeitpunkt
                              ,k_maske.tag_im_jahr));
    pipe row (rec);
  end loop;
  return;
end;
----------------------------------------------------------------
function add_saison
( curs in t_zeit_refcurs )
return t_zeit_col pipelined
parallel_enable (partition curs by any)
```

```
  is
    k_sommer_anfang number
      := to_number(to_char(to_date('0103','ddmm'),'ddd'));
    k_sommer_ende   number
      := to_number(to_char(to_date('3108','ddmm'),'ddd'));
    col t_zeit_col;
  begin
    loop
      fetch curs bulk collect into col limit 100;
      exit when col.count = 0;
      for i in nvl(col.first,1) .. nvl(col.last,0) loop
        if col(i).tag_im_jahr
        between k_sommer_anfang and k_sommer_ende
        then
          col(i).saison := 'SS';
        else
          col(i).saison := 'WS';
        end if;
        pipe row (col(i));
      end loop;
    end loop;
    return;
  end;
  ----------------------------------------------------------------
end;
```

Abbildung 8: Package Body PCK_ZEIT mit vier Table-Functions

```
Select * from table(pck_zeit.get_zeit_en_bloc);
Select * from table(pck_zeit.get_zeit_pipelined);
```

Abbildung 9: Einbinden von Table-Functions in SQL (pipelined und nicht pipelined)

Kontaktadresse:

Gero Knapstein
OPITZ CONSULTING GmbH Gummersbach
Kirchstraße 6
D-51647 Gummersbach

Telefon: +49 (0) 2261-6001-0
E-Mail: Gero.Knapstein@opitz-consulting.com
Internet: www.opitz-consulting.com