Parallel Enabled Pipelined Table Functions im Praxiseinsatz

Karin Patenge
Oracle Deutschland B.V. & Co. KG
Potsdam

Schlüsselworte
Table Functions, Parallel Processing, Pipelining, Performance, PL/SQL, Code Optimization, Database, Development, Geocoding, Spatial Data Processing.

Einleitung
Table Functions sind PL/SQL-Funktionen, die sich ähnlich verhalten, wie SQL SELECT-Statements. Sie liefern ganze Tabellen anstatt skalarer Werte zurück. Diese wurden mit der Oracle Datenbank Version 9i eingeführt und tauchen seither immer wieder als Tipp auf, wenn es um fortgeschrittene Techniken zum Performance Tuning der Oracle Datenbank geht.
Dieser Artikel gibt einen Überblick zum Aufbau und der Nutzung von *Table Functions* am Beispiel der Prozessierung von raumbezogenen Daten. Dieser fachliche Hintergrund wurde gewählt, weil dabei typischerweise sehr große Datenmengen in verschiedenen Schritten effizient zu prozessieren sind. Hier können über den Einsatz von Table Functions und deren parallele Verarbeitung sehr große Performanceunterschiede gemessen werden.

Einführung in Table Functions
Eingangs hatte ich beschrieben, das *Table Functions* im Ergebnis jeweils eine Tabelle ausgeben. Grundlage dafür ist der *Table Operator*, welcher den Inhalt einer *Collection* als relationales Datenset zurückgibt, dass dann einem SELECT-Aufruf übergeben werden kann. Die Struktur des Datensets wird typischerweise als ein Objekttyp mit `CREATE TYPE` beschrieben.
Innherhalb des *Table Operator* kann ebenso eine Funktion aufgerufen werden, welche eine solche *Collection* zurückgeben. Die Funktion kann dabei Standalone oder innerhalb eines Package definiert sein. Listing 1 zeigt die Syntax für einen solchen Aufruf sowie ein paar erste einfache Beispiele.

```
-- Allgemeine Syntax für Table Functions
select column_list from table (function_name(parameter_list))
/

select column_list
from table (collection1) t1
  , table_name t2
  , table (collection2) t3
/

-- Anlegen einen Objekttyps für eine einfache Collection
create or replace type list_of_cities_t
  as table of varchar2(50)
/

-- Anlegen einer Funktion, die eine Collection
--   von skalaren Werten (Strings) zurückgibt
create or replace function my_favorite_cities
```

```
return list_of_cities_t
is
  cities list_of_cities_t := list_of_cities_t (
    'Berlin'
    ,'Sydney'
    ,'Vancouver'
    ,'Seattle'
    ,'Stockholm');
begin
   return cities;
end;
/

-- Aufruf der Funktion im Table Operator als Table Function
select * from table(my_favorite_cities())
/

-- Ergebnis:
COLUMN_VALUE
Berlin
Sydney
Vanouver
Seattle
Hamburg
```

Listing 1: Syntax von Table Functions und Beispiel (siehe auch [1])

Dieser Ansatz funktioniert natürlich auch mit nicht-skalaren Werten, sondern Objekten, die aus mehr als einem Attribut bestehen. Dabei reicht es dann nicht mehr, nur einen Objekttyp anzulegen. Zusätzlich wird ein zweiter Objekttyp definiert, der eine Nested Table darstellt und als TABLE OF vom 1. Objekttyp angelegt wird.
Das Beispiel in Listing 2 gibt eine Tabelle mit 2 Instanzen des 2. Objekttyps zurück. Jede Instanz (Zeile) enthält als Attribute einen ISO-Country Code, einen Hauptstadt-Namen sowie den Mittelpunkt der Hauptstadt als SDO_GEOMETRY-Objekt.

```
-- Komplexer Objekttyp mit 3 Attributen
create or replace type capital_t as object (
  iso_country_code char(2)
  , name varchar2(100)
  , location sdo_geometry)
/
-- Anlegen einer Nested Table basierend auf dem Objekttyp
create or replace type capitals_nt
   is table of capital_t
/
-- Funktion, die eine Nested Table zurückgibt
--   mit 2 Instanzen vom zuvor angelegten Objekttyp
create or replace function list_countries
return capitals_nt
is
  l_return capitals_nt :=
```

```
    capitals_nt(
      capital_t(
       'DE'
       , 'Berlin'
       , mdsys.sdo_geometry(
         2001
        , 8307
        , mdsys.sdo_point_type(
          13.37714
          , 52.51598
          , null), null, null))
      , capital_t (
        'FR'
        , 'Paris'
        , mdsys.sdo_geometry(
          2001
          , 8307
          , mdsys.sdo_point_type(
            2.34125
            , 48.85687
            , null), null, null)));
begin
  return l_return;
end;
/
-- Aufruf der Table Function
select * from table(list_countries())
/

-- Ergebnis:
ISO_COUNTRY_CODE NAME   LOCATION
DE               Berlin [MDSYS.SDO_GEOMETRY]
FR               Paris  [MDSYS.SDO_GEOMETRY]
```

Listing 2: Beispiel Table Function mit nicht-skalaren Rückgabewerten

Für die Arbeit mit raumbezogenen Daten in der Oracle Datenbank (Oracle Locator bzw. Oracle Spatial and Graph) gibt es einige bereits implementierte *Table Functions*, so. z.B.

(a) für die Extraktion der Koordinaten aller Stützpunkte in einem Polygon, welches zur Beschreibung einer Fläche definiert wird (Funktionalität von Oracle Locator) oder
(b) die in 12.1 eingeführte Funktion `SDO_POINTINPOLYGON`, die ermittelt, ob ein 2-dimensionaler Punkt räumlich gesehen innerhalb eines Polygons (Fläche) liegt (Funktionalität von *Oracle Spatial and Graph*).

Listing 3 zeigt die SQL Syntax für die Verwendung dieser *Table Functions* auf dem World Sample Beispieldatenset des Oracle Partners HERE [1]. Für die PointInPolygon-*Table Function* wird das Polygon direkt als ein `SDO_GEOMETRY`-Objekt übergeben. Diese spannt ein Rechteck mit den Koordinaten Länge=8 / Breite = 48 für die linke untere Ecke sowie Länge=15 / Breite = 55 für die rechtere obere Ecke über Deutschland auf (siehe Abb. 1). Das verwendete Koordinatenbezugssystem ist 8307. Die Punkte liefert die Tabelle WOM_POI . POI steht hierbei für *Points of Interest*.

```
-- PointInPolygon-Table Function
```

```
select *
from table(
  sdo_pointinpolygon(
    cursor(
      select
        p.geometry.sdo_point.x lon   -- Extraktion x-Koordinatenwert
        , p.geometry.sdo_point.y lat -- Extraction y-Koordinatenwert
        , poi_id id
        , name
      from
        wom_poi p),
      sdo_geometry (
        2003
        , 8307
        , null
        , mdsys.sdo_elem_info_array(1, 1003, 3)
        , mdsys.sdo_ordinate_array(6.0, 47.0, 15.0, 55.0))
        , 0.05))
/
```

Listing 3: PointInPolygon-Abfrage als Table Function (Table Operator)

Abb.1: Abfrage-Polygon (optimiertes Rechteck – rote Fläche, überdeckt) im Vergleich zum Deutschland-Umriss (gelbe Fläche)

Ausführungsverhalten von Table Functions

Die *Table Function*, so wie weiter oben beschrieben, wird seriell abgearbeitet (Abbildung 2) und gibt die *Collection* immer als Ganzes an das aufrufende Statement zurück. Das kann in Abhängigkeit von der Größe der *Collection*

- länger dauern,
- ggf. größere Bereiche der PGA belegen und
- es besteht keine Möglichkeit, die Abarbeitung zu parallelisieren.

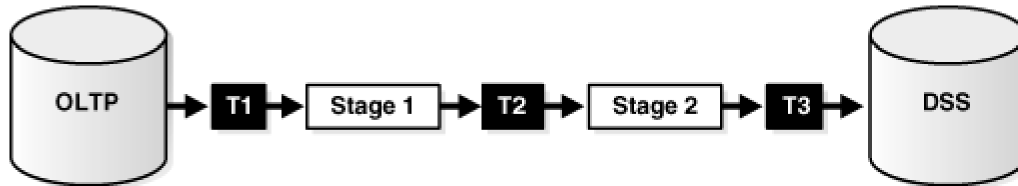

Abb.2: Typische Abarbeitung von non-parallel, non-pipelined Table Functions [4]

Dieses Verhalten lässt sich am Beispiel in Listing 4, einer kleinen Erweiterung des Beispiels aus Listing 1 sehr schön beobachten.

```
create or replace function my_favorite_cities_delayed (
  p_delay in integer)
return list_of_cities_t
is
  cities list_of_cities_t := list_of_cities_t (
    'Berlin'
    ,'Sydney'
    ,'Vancouver'
    ,'Seattle'
    ,'Stockholm');
begin
   -- Hierfür wird grant execute on sys.dbms_lock ... benötigt
   dbms_lock.sleep(p_delay);
   return cities;
end;
/

-- Aufruf der Funktion im Table Operator als Table Function
select * from table(my_favorite_cities_delayed(10))
/
```

Listing 4: Beispiel für optimierbares Verhalten von Table Functions

Diese Gründe (und möglicherweise auch noch andere) führten zu mehreren Ansätzen, um das Ausführungsverhalten und damit die Antwortzeiten zu verbessern. Diese Ansätze sind

1. *Pipelining*,
2. *Streaming* und
3. Parallelisierung.

Streaming beschreibt Steven Feuerstein im Teil 4 in [1]. Die Nutzung von Pipelining und Parallelisierung ist Gegenstand der nachfolgenden Ausführungen.

Pipelined Table Functions
Mittels *Pipelining* wird die Möglichkeit eröffnet, die Collections nicht als Ganzes sondern zeilenweise an das aufrufende Statement zurückzugeben. Dieses muss nicht unnötig auf das Ende der Abarbeitung der Funktion warten, sondern kann sofort mit den zurückgegebenen Zeilen arbeiten. Das passiert über eine Erweiterung in der Syntax bei der Definition der Funktion.

- Der `CREATE`-Befehl wird um das Schlüsselwort `PIPELINED` ergänzt
- Die einzelnen Zeilen der Collection werden mit `PIPE ROW` zurückgegeben
- Jede Funktion muss eine `RETURN`-Befehl enthalten, auch wenn kein Wert zurückgegeben wird.

Listing 5 zeigt die Syntax am Beispiel der Berechnung der Fibonacci-Zahlen.

```
-- Berechnung Fibonacci Zahlen mit Pipelined Table Function
create or replace type fibonacci_t as table of number(10);
/

create or replace function list_fibonacci (
  p_count in number)
return fibonacci_t pipelined  -- Schlüsselwort PIPELINED
as
  n0 number :=0;
  n1 number :=1;
  n2 number :=0;
begin
                   -- Schlüsselwort PIPE ROW
  pipe row(n0);    -- Rückgabe Zeile 1 an aufrufendes Statement
  pipe row(n1);    -- Rückgabe Zeile 2
  for i in n0 .. p_count-3
  loop
    n2 := n0 + n1 ;
    pipe row(n2);  -- Rückgabe jeder weiteren Zeilen
    n0 := n1;
    n1 := n2;
  end loop;
end list_fibonacci;
/
-- Aufruf der Pipelined Table Function
select column_value fibonacci from table(list_fibonacci(1000))
/
```

Listing 5: Beispiel für Pipelined Table Functions

Die Spalte mit den Fibonacci-Zahlen erhält übrigens den Default-Namen `COLUMN_VALUE`. Dieser taucht im Ergebnis in Listing 1 bereits auf.

Eine weitere Optimierungsstufe wird gezündet: Parallel-enabled Pipelined Table Function
Soll nun noch erreicht werden, dass die standardmäßig serielle Abarbeitung (PL/SQL ist nun mal keine *Multi-Threading*-fähige Sprache) parallelisiert wird, ist die Syntax noch einmal zu erweitern. Diesmal um das Schlüsselwort `PARALLEL_ENABLE`.

Die prinzipielle Art und Weise, wie dann das aufrufende Statement ausgeführt wird, ist in Abbildung 3 verdeutlicht. Der erwartete Nutzen in Bezug auf schnellere Ausführungszeiten wird dadurch schon sehr deutlich.

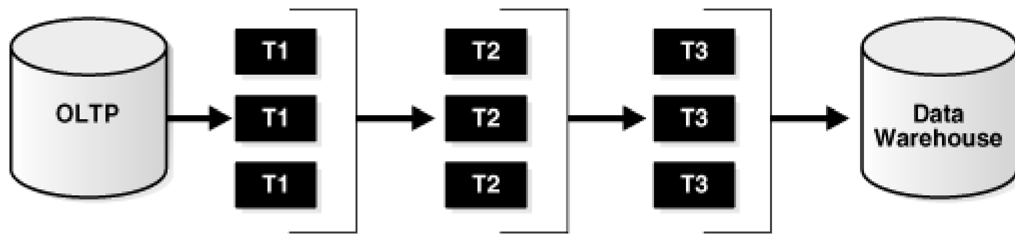

Abb.3: Typische Abarbeitung von parallel, pipelined Table Functions [4]

Damit Parallelisierung auch wirklich funktioniert, greife ich einen Hinweis von Carsten Czarski aus seinem Blog Posting in 2014 [3] auf. Darin beschreibt er, dass es sinnvoll ist, einen Cursor als Parameter für die *Table Function* zu verwenden. Genau dieser Ansatz wird auch im Beispiel aus Listing 6 genutzt. Dieses Beispiel stammt aus einem Workshop über die Nutzung von Oracle Spatial and Graph [8], bezieht sich also wieder auf raumbezogene Daten. Autor des Codes ist Daniel Geringer. Im Beispiel sollen Adressdaten in großen Mengen möglichst schnell geokodiert werden. Mit Geokodierung wird die Umrechnung direkt in der Datenbank (Teil der Datenbank Option *Oracle Spatial and Graph*) einer Adresse in eine Geokoordinate mit Längen- und Breitengrad bezeichnet. Eine Funktion, welche eine Adresse geokodiert, müßte für sehr viele Daten entsprechend oft aufgerufen werden. Das beansprucht Ressourcen und dauert möglicherweise sehr lange (in Abhängigkeit der eingelesen Adressdaten).
Lösung für dieses Problem ist eine parallel ausgeführte *Pipelined Table Function.* Die *Table Function* gibt die Ergebnismenge als eine Tabelle mit den Geokoordinaten und zusätzlichen adress-relevanten Informationen (Instanzen des Objekttyps `SDO_GEO_ADDR_TABLE_TYPE`) zurück. Eingabewerte sind Adressen (Tabelle `CUSTOMER_ADDRESSE_EXT`), welche über einen Cursor eingelesen werden. `PARALLEL QUERY` gepaart mit `PARALLEL` Hints sorgen dafür, dass die Geokodierung auf mehrere parallel laufende Prozesse wird. Am Ende sorgt ein Koordinator-Prozess für das Zusammenführen der Teilergebnisse der einzelnen Prozesse.
Hinweis: Eine ausführliche Darstellung, wie Parallelisierung von Abfragen und DML funktioniert, ist nicht Gegenstand dieser Arbeit. In [5] wird aber ein verständlicher Überblick in deutscher Sprache gegeben. [7] ist ein Whitepaper in englischer Sprache.
Die eigentliche *Table Function* (parallel und pipelined) `GEOCODE_PARSED` ist im PL/SQL Package `GEOCODE_UTILS` im Listing 5definiert.

```
-- Objekttyp SDO_GEO_ADDR_TABLE_TYPE
create or replace type sdo_geo_addr_table_type
  as table of sdo_geo_addr;
/

-- PL/SQL Package Definition
create or replace package geocode_utils
  as type in_address_row_type is record (
    id         number,
    streetname varchar2(1000),
    city       varchar2(100),
    state      varchar2(100),
    zip        varchar2(100),
```

```
    housenum   varchar2(100));
    type in_address_cursor_type is ref cursor
      return in_address_row_type;
    type in_lon_lat_row_type is record (
      id         number,
      lon        number,
      lat        number);
    type in_lon_lat_cursor_type is ref cursor
      return in_lon_lat_row_type;

-- Deklaration der Table Function GEOCODE_PARSED
  function geocode_parsed (
    source_table_cursor in in_address_cursor_type,
    schema              in varchar2 := 'odf_gbl',
    country             in varchar2 := 'us',
    matchmode           in varchar2 := 'relax_postal_code')
    return sdo_geo_addr_table_type deterministic
-- PIPELINED and PARALLEL_ENABLE
      pipelined parallel_enable (
-- Verteilung auf parallele Prozesse mittels
--   PARTITION … BY
        partition source_table_cursor by hash (id));
end;
/

-- Definition  Package Body
create or replace package body geocode_utils as
  type num_table_type is table of number;
  type string_table_type is table of varchar2(1000);

  function geocode_parsed (
    source_table_cursor in in_address_cursor_type,
    schema              in varchar2 := 'odf_gbl',
    country             in varchar2 := 'us',
    matchmode           in varchar2 := 'relax_postal_code')
    return sdo_geo_addr_table_type deterministic
      pipelined parallel_enable (
        partition source_table_cursor by hash (id))
  as
    id_table         num_table_type;
    streetname_table string_table_type;
    city_table       string_table_type;
    state_table      string_table_type;
    zip_table        string_table_type;
    housenum_table   string_table_type;
    geo_addr         sdo_geo_addr;
    upper_country    varchar2(100) := upper(country);
    empty_addr       sdo_geo_addr := sdo_geo_addr(
      0,null,null,null,null,null,
      null,null,null,null,null,null,
      null,null,null,null,null,null,
```

```
      null,null,null,null,null,null,
      null,null,null,null,null);
  begin
    loop
      fetch source_table_cursor bulk collect into
        id_table,
        housenum_table,
        streetname_table,
        city_table,
        state_table,
        zip_table
        limit 100;

      exit when id_table.count = 0;

      for i in id_table.first .. id_table.last loop
        begin
          geo_addr := sdo_gcdr.geocode_addr(
            schema,
            sdo_geo_addr(
              0, null, null,
              streetname_table(i), null, null,
              city_table(i), null,
              state_table(i), upper_country,
              zip_table(i), null, null, null,
              housenum_table(i), null, null, null, null,
              null, null, null, null, null, null, null,
              matchmode,null,null));
            if geo_addr is not null then
              geo_addr.id := id_table(i);
              pipe row (geo_addr);
            else
              empty_addr.id := id_table(i);
              pipe row (empty_addr);
            end if;
        exception
          when others then
            empty_addr.id := id_table(i);
            pipe row (empty_addr);
        end;
      end loop;

      -- exit when less than limit rows were fetched.
      exit when source_table_cursor%notfound;
    end loop;
    close source_table_cursor;
    return;
  end;
end;
/
```

```
select count(*) from customer_addresses_ext
/
-- 77216 customers

set timing on

-- Session für PARALLEL QUERY vorbereiten
alter session force parallel query
/

-- Aufruf mit PARALLEL 2 Hint (Abarbeitung mit 2 Prozessen)
select /*+ parallel (2) */
  a.id customer_id,
  a.longitude,
  a.latitude,
  --a.housenumber,
  --a.streetname,
  --a.settlement,
  --a.region,
  --a.postalcode,
  --a.matchcode,
  a.edgeid link_id,
  a.percent percentage
from table(
  geocode_utils.geocode_parsed (
    cursor(
      select in_customer_id,
        in_housenumber,
        in_streetname,
        in_city,
        in_state,
        in_zip
      from customer_addresses_ext),
       'here_sf')) a
/

select customer_id, longitude, latitude from customers
where rownum < 10
/

-- Ergebnis: Tabelle mit Geokoordinaten
CUSTOMER_ID  LONGITUDE   LATITUDE
----------- ---------- ----------
         61 -122.32247   37.93102
         66 -122.32151 37.9298011
         67 -122.32049 37.9283545
         71 -122.31912 37.9261773
         90 -122.30937 37.9125475
         99 -122.31204 37.9161567
        106 -122.31292   37.91765
        108 -122.31335 37.9182771
```

```
        110 -122.31371 37.9187704

9 rows selected.
```

Listing 6: Geokodierung mittels Parallel-enabled Pipelined Table Function

Abbildung 4 zeigt die 2 parallel laufende Prozesse.

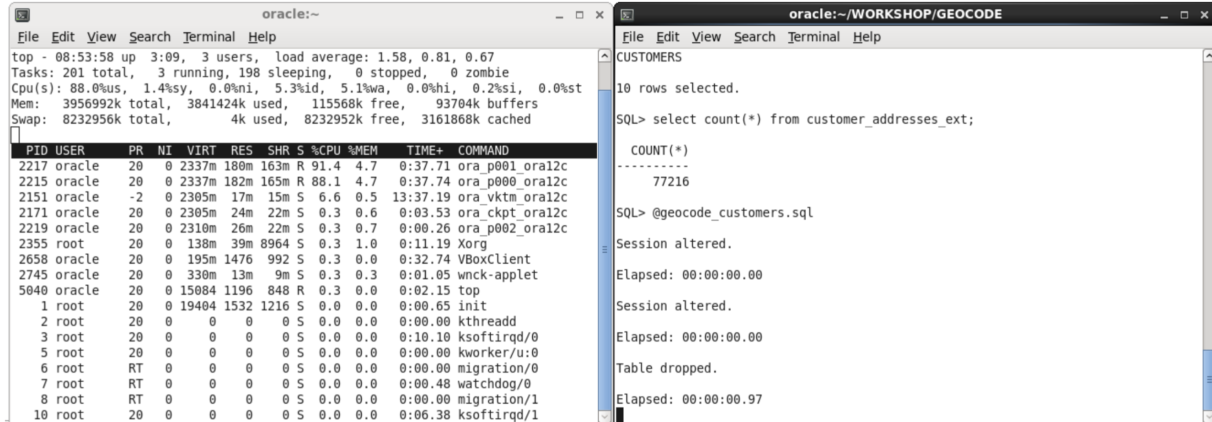

Abb. 4: Aufruf der Geocodierung (rechts) und Anzeige der laufenden Prozesse mit TOP

Alternativ kann auch nur der Ausführungsplan überprüft werden, wie in Listing 7 aufgezeigt.

```
explain plan for
select /*+ parallel (2) */
  a.id customer_id,
  a.longitude,
  a.latitude,
  --a.housenumber,
  --a.streetname,
  --a.settlement,
  --a.region,
  --a.postalcode,
  --a.matchcode,
  a.edgeid link_id,
  a.percent percentage
from table(
  geocode_utils.geocode_parsed (
    cursor(
      select in_customer_id,
        in_housenumber,
        in_streetname,
        in_city,
        in_state,
        in_zip
      from customer_addresses_ext),
       'here_sf')) a
/
```

```
select * from table(dbms_xplan.display(format=>'basic +note'))
/

-- Ergebnis/Ausführungsplan

PLAN_TABLE_OUTPUT
--------------------------------------------------------------------------------
Plan hash value: 4009970411

--------------------------------------------------------------------------------
| Id  | Operation                                     | Name                     |
--------------------------------------------------------------------------------
|   0 | SELECT STATEMENT                              |                          |
|   1 |  PX COORDINATOR                               |                          |
|   2 |   PX SEND QC (RANDOM)                         | :TQ10001                 |
|   3 |    BUFFER SORT                                |                          |
|   4 |     VIEW                                      |                          |
|   5 |      COLLECTION ITERATOR PICKLER FETCH        | GEOCODE_PARSED           |

PLAN_TABLE_OUTPUT
--------------------------------------------------------------------------------
|   6 |       PX RECEIVE                              |                          |
|   7 |        PX SEND HASH                           | :TQ10000                 |
|   8 |         PX BLOCK ITERATOR                     |                          |
|   9 |          EXTERNAL TABLE ACCESS FULL           | CUSTOMER_ADDRESSES_EXT   |
--------------------------------------------------------------------------------

Note
-----
   - Degree of Parallelism is 2 because of hint
```

Zusammenfassung

Pipelined Table Functions sind ein vielfach bewährtes Mittel, um

1. Häufig zur Ausführung kommenden PL/SQL Code in Funktionen zu kapseln und
2. deren Ausführung mittels Pipleining und Parallelisierung erheblich zu beschleunigen.

Sie entfalten ihren Nutzen insbesondere bei der Anwendung auf viele einzelne Datensätze, wie dies häufig im Bereich von Geodaten vorkommt. Geokodierung ist dabei nur ein Beispiel. Weitere Anwendungsfälle sind

- das Berechnen von Nachbarschaftspunkten in sogenannten Punktwolken, die im Zuge von Laserscankampagnen erzeugt werden und u.a. dem Aufbau von 3D Stadtmodellen dienen oder
- das Mappen von Zugpositionen (viele einzelne Geokoordinaten) auf bestimmte Strecken oder
- das stufenweise Vorprozessieren von OLTP-Daten für die weitere Verwendung in einem Data Warehouse.

Es gibt wie immer viele Wege, ans Ziel zu kommen oder anders ausgedrückt, eine Anforderung programm-technisch umzusetzen. Dieser Artikel soll dazu anregen, sich bei jeder neuen Anforderung oder auch für bestehenden Code (bei Performanceproblemen) Gedanken zu machen, ob (Pipelined) Table Functions einen guten Weg aufzeigen können.

Wer sich mehr zum Thema informieren möchte, findet in den nachfolgenden Links genug Stoff zum Lesen und Ausprobieren.

Weitere Informationen

[1] Blog „Steven Feuerstein on Oracle PL/SQL": Serie zu Table Functions
http://stevenfeuersteinonplsql.blogspot.co.uk/search/label/table%20function

[2] Oracle Spatial and Graph Partners' Data Download: HERE (Formerly NAVTEQ)
http://www.oracle.com/technetwork/database/options/spatial/spatial-partners-data-087203.html

[3] Blog „Oracle SQL and PL/SQL" von Carsten Czarski
http://sql-plsql-de.blogspot.co.uk/2007/03/eine-funktion-selektieren-wie-eine.html
http://sql-plsql-de.blogspot.co.uk/2014/03/parallele-ausfuhrung-von-table-functions.html

[4] Database Data Cartridge Developer´s Guide 12.1: Using Pipelined and Parallel Table Functions
https://docs.oracle.com/database/121/ADDCI/pipe_paral_tbl.htm#ADDCI2140

[5] Deutschsprachige Oracle Datenbank-Community Seiten: Parallel Query ganz automatisch mit Oracle 11g
http://www.oracle.com/webfolder/technetwork/de/community/dbadmin/tipps/parallel_query/index.html

[6] Database 12.1 PL/SQL Language Reference: PL/SQL Optimization and Tuning
http://docs.oracle.com/database/121/LNPLS/tuning.htm#LNPLS916

[7] Whitepaper: Parallel Execution with Oracle Database 12c Fundamentals
http://www.oracle.com/technetwork/database/bi-datawarehousing/twp-parallel-execution-fundamentals-133639.pdf

[8] Materials for the 2014 Oracle Spatial Summit Workshops: Workshop 4 – Learn To Build An Analytics Solution With Oracle's Spatial Tools and Platform
http://www.oracle.com/technetwork/database/options/spatialandgraph/community/sagsummit-2014-2196705.html#AnalyticsSolution

Kontaktadresse

Karin Patenge
Oracle Deutschland B.V. & Co. KG
Schiffbauergasse 14
D-14467 Potsdam

Telefon:	+49 (0) 331-200 7214
E-Mail	karin.patenge@oracle.com
Internet:	www.oracle.com