Die Unterlagen zum Workshop

Einsatz Oracle 11.2 für Anwendungs-entwickler

Impressum

Dies sind die Unterlagen zum Workshop »Einsatz Oracle 11.2 für Anwendungsentwickler« in der Fa. Sanacorp.

Dieser hausinterne Workshop wird zum Abschluss des Projekts »Einsatz Oracle 11.2« durchgeführt, nachdem die vorgesehenen Datenbanken vollständig umgestellt wurden. Mit dem Ziel, die interessierten Mitarbeiter von Entwicklung & Prozesse (Anwendungsentwicklung) auf einige nützliche neue Möglichkeiten hinzuweisen, welche die Oracle-Datenbank-Version 11.2 gegenüber der Version 10g bietet.

Für eine erfolgreiche Teilnahme sind zumindest grundlegende SQL- und PL/SQL-Kenntnisse Voraussetzung.

Termin: 11.12.2015

Referent: Carsten Kaftan
C.Kaftan.sanacorp@myway.de
IT / DBA-Team
Sanacorp Pharmahandel GmbH
Semmelweisstraße 4
82152 Planegg

Titelfoto: Carsten Kaftan (privat)

Inhaltsverzeichnis

Beispiel-Daten »FAHRTEN«

Zur Veranschaulichung der Ausführungen wird eine einfache Tabelle mit Zufallsdaten bereitgestellt, anhand derer die SQL-Statements demonstriert werden. Die Daten liegen in der Datenbank **DB.hv.test**, im Schema **SCHULUNG**, Tabelle **FAHRTEN**.

Diese Tabelle enthält elementare Logging-Informationen zu aufeinanderfolgenden Fahrten mehrerer Personen zwischen einigen Orten: Neben dem Fahrer werden Startort und Abfahrtszeit sowie Zielort und Ankunftszeit protokolliert. Aus den vorgegebenen Listen der Fahrer und der Standorte werden mit gewichteten Zufallswerten zusammenpassende Touren über die letzten Monate erzeugt.

ID	FAHRER	ZEIT_AB	ORT_AB	ZEIT_AN	ORT_AN
			...		
9877	Anke	07.05.2013 22:36:00	Hannover	08.05.2013 04:28:00	Saarbrücken
9878	Anke	08.05.2013 06:25:00	Saarbrücken	08.05.2013 22:14:00	Hannover
9879	Anke	09.05.2013 03:13:00	Hannover	09.05.2013 17:46:00	Asperg
9880	Anke	09.05.2013 21:09:00	Asperg	10.05.2013 08:45:00	Düsseldorf
9881	Anke	10.05.2013 11:36:00	Düsseldorf	10.05.2013 23:01:00	Bad Segeberg
9882	Anke	11.05.2013 05:19:00	Bad Segeberg	11.05.2013 21:48:00	Mainz
9883	Anke	12.05.2013 01:07:00	Mainz	12.05.2013 06:27:00	Ulm
9884	Anke	12.05.2013 08:18:00	Ulm	12.05.2013 18:42:00	Tuttlingen
9885	Anke	12.05.2013 21:40:00	Tuttlingen	13.05.2013 05:21:00	Saarbrücken
9886	Anke	13.05.2013 06:58:00	Saarbrücken	13.05.2013 10:39:00	Offenburg
			...		
56932	Stephan	27.08.2015 14:47:00	Hannover	28.08.2015 12:22:00	Planegg
56933	Stephan	28.08.2015 13:23:00	Planegg	29.08.2015 06:22:00	Chemnitz
56934	Stephan	29.08.2015 15:57:00	Chemnitz	30.08.2015 15:32:00	Planegg
56935	Stephan	30.08.2015 23:46:00	Planegg	31.08.2015 21:55:00	Ulm
56936	Stephan	01.09.2015 07:09:00	Ulm	01.09.2015 15:05:00	Fürth
56937	Stephan	01.09.2015 23:00:00	Fürth	02.09.2015 11:33:00	Mainz
56938	Stephan	02.09.2015 18:48:00	Mainz	03.09.2015 17:33:00	Ulm
56939	Stephan	04.09.2015 02:51:00	Ulm	04.09.2015 17:00:00	Potsdam
			...		

Standorte (ORT_AB/AN)

Stralsund
Bad Segeberg
Hamburg
Hannover
Potsdam
Herne
Düsseldorf
Chemnitz
Mainz
Fürth
Saarbrücken
Asperg
Offenburg
Ulm
Planegg
Tuttlingen

Fahrer (FAHRER)

Aaron
Abdellahi
Adrian
Alexander
Andrea
Andreas
Anke
Anselm
Antonia
Bela
Benjamin
Bernd
Birgitta
Christian
David
Dietmar
Florian
Günther
Gerhard
Hans-Josef
Heidi
Heike
Helga
Jan
Jens
Josef
Julia
Marcel
Maria
Marko
Michael
Michaela
Monia
Nikolaj
Oleg
Oliver
Ralf
Robert
Saad
Sabine
Sebastian
Stephan

Recursive Subquery Factoring

Rekursive Aufrufe — inline im SQL-Statement formuliert! — erlauben elegant und übersichtlich den Durchlauf durch verschachtelte Strukturen. Die einfachste Verwendung ist die iterative »Erzeugung« von Ergebniszeilen; ihre Stärke sind Baumdurchläufe.

Ein einzelner Wert lässt sich mit einer Abfrage auf dual schnell erzeugen.

```
select 12 as WERT from dual;
```

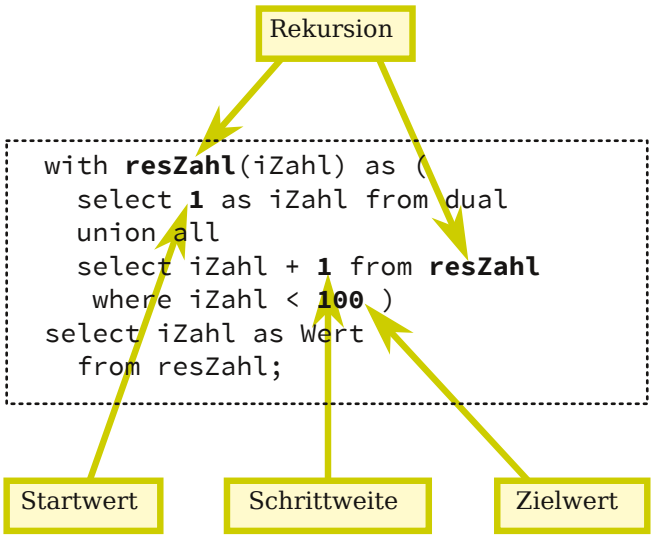
Wert
12

Kleine Zahlenfolgen sind entsprechend einfach zu erzeugen, bei größeren wird es schnell umständlich.

```
select 1 as WERT from dual
union all select 2 from dual
union all select 3 from dual;
```

Wert
1
2
3

Mit **Recursive Subquery Factoring** lassen sich Ergebniszeilen »aus dem Nichts« herstellen. Im nebenstehenden Beispiel werden die Zahlen von 1 bis 100 ausgegeben.



Wert
1
2
3
...
98
99
100

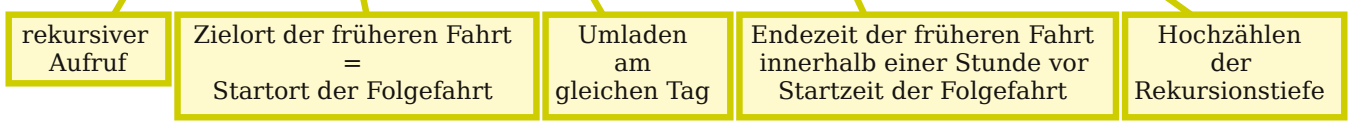
Der Datentyp ist nicht auf Zahlen eingeschränkt, es lassen sich auch Texte oder Datumswerte verwenden. Im Beispiel wird für jeden Tag des aktuellen Monats bis zum heutigen Tag eine Zeile ausgegeben.

```
with resDay(dDay) as (
  select trunc(SYSDATE, 'MM')
           as dDay from dual
  union all
  select dDay + interval'1'day
    from resDay
   where dDay < trunc(sysdate))
select * from resDay;
```

dDay
01.10.2015
02.10.2015
03.10.2015
...
12.10.2015
13.10.2015
14.10.2015

Viel interessanter als das Herbeizaubern von Werten sind die Möglichkeiten der Analyse von Datenbeziehungen. Zwischen welchen Standorten lassen sich Medikamente transportieren, ggf. mit Umladen? In der Tabelle FAHRTEN sollen alle diejenigen Verbindungen zwischen Standorten herausgesucht werden, bei denen jeweils innerhalb einer Stunde am selben Tag die nötige Anschlussfahrt stattfindet:

```
with resVERBUNDEN(VON, NACH, BIS, UMLADEN) as (
  select f.ORT_AB, f.ORT_AN, f.ZEIT_AN, 0 from FAHRTEN f
  union all
  select r.VON,    f.ORT_AN, f.ZEIT_AN, UMLADEN+1
    from resVERBUNDEN r, FAHRTEN f
   where r.NACH = f.ORT_AB
     and r.BIS < f.ZEIT_AB and r.BIS > f.ZEIT_AB - interval'1'hour
     and trunc(r.BIS) = trunc(f.ZEIT_AB))
select UMLADEN, VON, NACH from resVERBUNDEN;
```



Umladen	Von	Nach
...		
5	Mainz	Bad Segeberg
5	Saarbrücken	Düsseldorf
5	Herne	Planegg
5	Tuttlingen	Mainz
5	Chemnitz	Bad Segeberg
5	Mainz	Bad Segeberg
5	Mainz	Hannover
5	Stralsund	Mainz
6	Chemnitz	Düsseldorf
6	Tuttlingen	Bad Sege'
6	Herne	Mainz
6	Tuttlingen	Ba
7	Herne	

Die zeitliche Abfolge — die Folgefahrt muss nach der früheren Fahrt starten — verhindert Schleifen. Ohne diese (fachlich sinnvolle) Einschränkung treten Laufzeitfehler auf, sobald die Datenbank einen endlosen Schleifendurchlauf erkennt.

```
with resVERBUNDEN(VON, NACH, BIS, UMLADEN) as (
  select f.ORT_AB, f.ORT_AN, f.ZEIT_AN, 0 from FAHRTEN f
  union all
  select r.VON,    f.ORT_AN, f.ZEIT_AN, UMLADEN+1
    from resVERBUNDEN r, FAHRTEN f
   where r.NACH = f.ORT_AB
--    and r.BIS < f.ZEIT_AB and r.BIS > f.ZEIT_AB - interval'1'hour
     and trunc(r.BIS) = trunc(f.ZEIT_AB))
select VON, NACH, UMLADEN from resVERBUNDEN;
```

Keine Erzwingung einer zeitlichen Reihenfolge

```
ORA-32044: cycle detected while
executing recursive WITH query
```

Mit der **CYCLE**-Festlegung lässt sich dieser Fehler vermeiden: In einer zusätzlichen Spalte wird, bei Erkennung einer Schleife bezüglich der aufgeführten Parameter, der vorgegebene Wert angezeigt und der Schleifendurchlauf für die betroffene Zeile abgebrochen. Hinweis: Ohne die Einschränkung auf eine sinnvolle zeitliche Reihenfolge werden jetzt sehr viele Ergebnissätze gefunden.

```
with resVERBUNDEN(VON, NACH, BIS, UMLADEN) as (
  select f.ORT_AB, f.ORT_AN, f.ZEIT_AN, 0 from FAHRTEN f
  union all
  select r.VON,    f.ORT_AN, f.ZEIT_AN, UMLADEN+1
    from resVERBUNDEN r, FAHRTEN f
   where r.NACH = f.ORT_AB
--    and r.BIS < f.ZEIT_AB and r.BIS > f.ZEIT_AB - interval'1'hour
     and trunc(r.BIS) = trunc(f.ZEIT_AB))
  cycle VON, NACH set SCHLEIFE to 'J' default 'N'
select UMLADEN, SCHLEIFE, VON, NACH from resVERBUNDEN;
```

CYCLE-Schlüsselwort

Festlegung der Attribute zur Erkennung von Schleifen

Name der zusätzlichen Pseudo-Spalte

Bei Erkennung einer Schleife (beliebiger Text der Länge 1)

Ansonsten (beliebiger Text der Länge 1)

Uml.	Schl.	Von	Nach
		...	
1	N	Mainz	Potsdam
1	J	Mainz	Hannover
1	N	Mainz	Bad Segeberg
1	N	Mainz	Stralsund
1	N	Mainz	Fürth
1	J	Mainz	Hannover
1	N	Hannover	Asperg
1	N	Hannover	Planegg
1	N	Hannover	Mainz
1	N	Hannover	Stralsund
1	N	Hannover	Stralsund
1	N	Hannover	Herne
1	N	Potsdam	Tut
		...	

CYCLE ist aber nicht nur zur Fehlervermeidung sinnvoll, sondern auch zum Kappen unnötiger Umwege: Die ursprüngliche Abfrage gibt mit dem zusätzlichen cycle-Bestandteil weniger Ergebnisse zurück und die maximale Anzahl von Umladetätigkeiten sinkt, wenn bereits vorhandene VON-NACH-Ergebnisse nicht weiterverfolgt werden. Die letzten Ergebniszeilen mit den höchsten Werten zeigen so maximal 5 Umladevorgänge.

```
with resVERBUNDEN(VON, NACH, BIS, UMLADEN) as (
  select f.ORT_AB, f.ORT_AN, f.ZEIT_AN, 0 from FAHRTEN f
  union all
  select r.VON,    f.ORT_AN, f.ZEIT_AN, UMLADEN+1
    from resVERBUNDEN r, FAHRTEN f
   where r.NACH = f.ORT_AB
     and r.BIS < f.ZEIT_AB and r.BIS > f.ZEIT_AB - interval'1'hour
     and trunc(r.BIS) = trunc(f.ZEIT_AB))
  cycle VON, NACH set SCHLEIFE to 'J' default 'N'
select UMLADEN, SCHLEIFE, VON, NACH from resVERBUNDEN;
```

Uml.	Schl.	Von	Nach
		...	
4	N	Chemnitz	Düsseldorf
4	N	Düsseldorf	Herne
4	N	Herne	Chemnitz
4	N	Asperg	Potsdam
5	J	Tuttlingen	Düsseldorf
5	N	Bad Segeberg	Hannover
5	N	Mainz	Bad Segeberg
5	J	Saarbrücken	Düsseldorf
5	N	Mainz	Bad Segeberg

Die Reihenfolge des Baumdurchlaufes lässt sich mit **SEARCH** festlegen; insbesondere ob zuerst in die Breite (default) oder in die Tiefe gegangen werden soll. Für gleichwertige Geschwisterergebniszeilen muss eine Sortierreihenfolge festgelegt werden. Der gesamte Ausdruck muss benannt werden; dieser Name kann dann nachfolgend — etwa für ein »order by« — weiterverwendet werden.

```
with resVERBUNDEN(VON, NACH, BIS, UMLADEN) as (
  select f.ORT_AB, f.ORT_AN, f.ZEIT_AN, 0 from FAHRTEN f
  union all
  select r.VON,    f.ORT_AN, f.ZEIT_AN, UMLADEN+1
    from resVERBUNDEN r, FAHRTEN f
   where r.NACH = f.ORT_AB
     and r.BIS < f.ZEIT_AB and r.BIS > f.ZEIT_AB - interval'1'hour
     and trunc(r.BIS) = trunc(f.ZEIT_AB))
  search depth first by VON, NACH set SORT
  cycle VON, NACH set SCHLEIFE to 'J' default 'N'
select UMLADEN, SCHLEIFE, SORT, VON, NACH from resVERBUNDEN;
```

SEARCH-Schlüsselwort

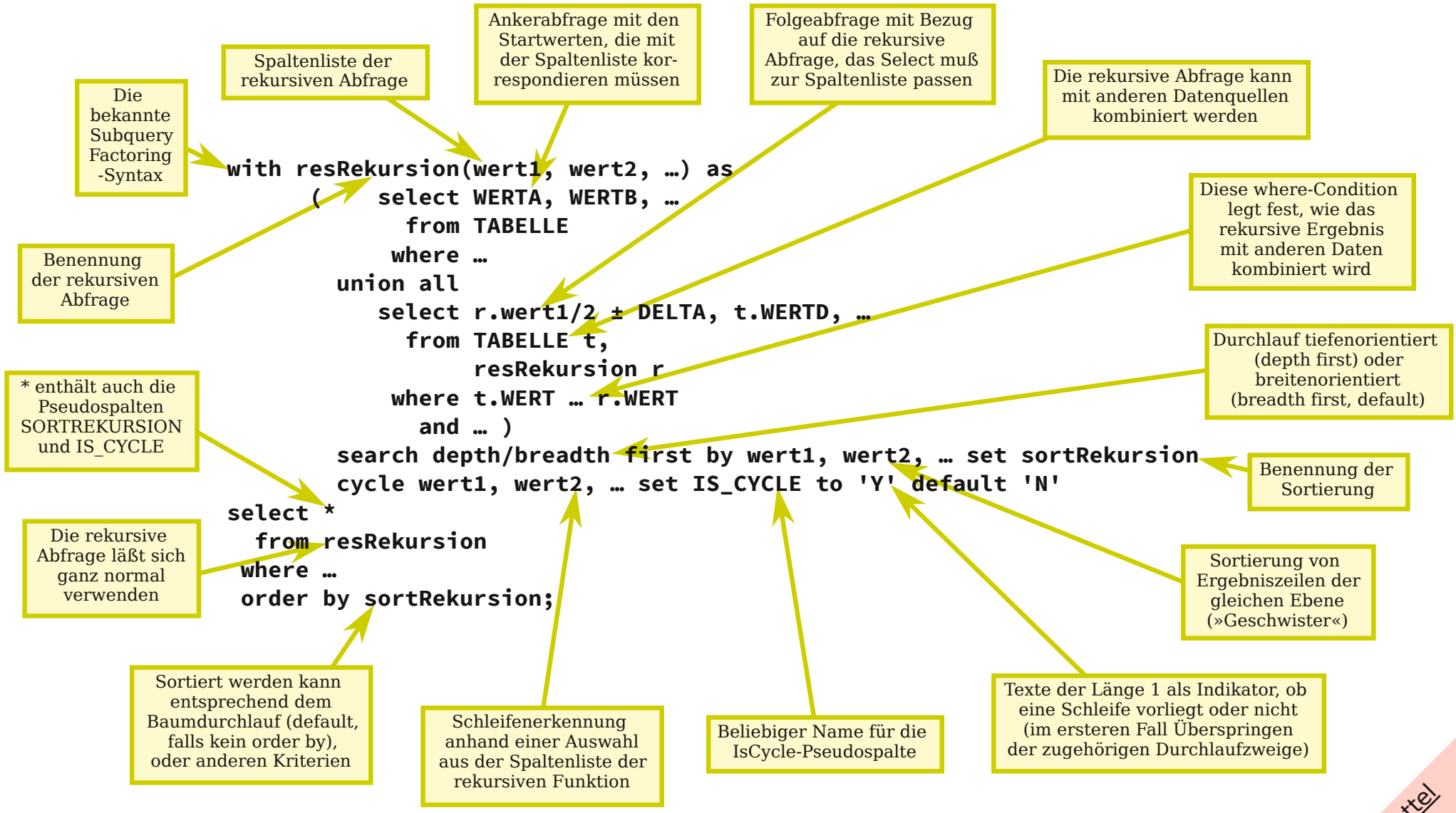
Zuerst in die Tiefe (»depth«) oder in die Breite (»breadth«)

Sortierung von Geschwisterzeilen

Benennung des Ausdrucks

Uml.	Schl.	Sort	Von	Nach
		...		
1	N	72820	Ulm	Chemnitz
0	N	72821	Ulm	Ulm
0	N	72822	Ulm	Ulm
0	N	72823	Ulm	Ulm
0	N	72824	Ulm	Ulm
0	N	72825	Ulm	Ulm
1	N	72826	Ulm	Düsseldorf
2	N	72827	Ulm	Hannover
2	N	72828	Ulm	Herne
3	J	72829	Ulm	Herne
2	N	72830	Ulm	Main[illegible]
1	N	72831	Ulm	M[illegible]
0	N	72832	Ulm	

Recursive Subquery Factoring




```
with resRekursion(wert1, wert2, …) as
       (    select WERTA, WERTB, …
              from TABELLE
             where …
            union all
            select r.wert1/2 ± DELTA, t.WERTD, …
              from TABELLE t,
                   resRekursion r
             where t.WERT … r.WERT
               and … )
      search depth/breadth first by wert1, wert2, … set sortRekursion
      cycle wert1, wert2, … set IS_CYCLE to 'Y' default 'N'
select *
  from resRekursion
 where …
 order by sortRekursion;
```

Online-Dokumentation (Oracle):
docs.oracle.com/cd/E11882_01/server.112/e41084/statements_10002.htm#sthref6730

Nth Value

Mit analytischen Funktionen lässt sich — auf Zeilenebene! — auf andere Datensätze der Ergebnismenge zugreifen. Dadurch sind sie weit universeller einsetzbar als Gruppenfunktionen, die auf zusammengefasste Ergebniszeilen Anwendung finden. Zunächst lassen sich mit analytischen Funktionen in vielen Fällen Self-Joins und Outer-Joins vermeiden; es gibt aber auch Verwendungen, die über einen performanten Ersatz von Gruppenfunktionen hinausgehen. Neu in Oracle 11.2 ist die analytische Funktion **NTH_VALUE**, für die es im Übrigen, anders als z.B. für **MAX** oder **AVG**, kein Pendant als Gruppenfunktion gibt.

Welche Pausenzeiten haben die Fahrer eingelegt? Zunächst eine »klassische« SQL-Abfrage mit einer Subquery, darunter die analytische Formulierung. Der für die Performance entscheidende Unterschied liegt in der Anzahl der Zugriffe auf die FAHRTEN-Tabelle — 2× bzw. 1×: Auch wenn der Execution-Plan vergleichbare Kosten ausweist, benötigt die analytische Abfrage weniger als eine Sekunde, während das erste SQL-Statement auch nach 10 Minuten noch nicht fertig ist. Abgesehen davon ist die untere Formulierung erheblich flexibler: Kleine Abwandlungen der Fragestellung, etwa die nach dem übernächsten Wert, lassen sich mit analytischen Funktionen ganz einfach formulieren, dagegen über herkömmliche Subqueries nicht mit erträglichem Aufwand darstellen.

```
select FAHRER, ORT_AB, ORT_AN,
       round((ZEIT_AN - ZEIT_AB)*24, 1) Zeit,
       round(((select min(ZEIT_AB) from FAHRTEN fi
                where fi.FAHRER = fo.FAHRER
                  and fi.ZEIT_AB > fo.ZEIT_AN) - ZEIT_AN)*24, 1) Pause
  from FAHRTEN fo
 order by FAHRER, ORT_AB, ZEIT_AB;
```

```
select FAHRER, ORT_AB, ORT_AN,
       round((ZEIT_AN - ZEIT_AB)*24, 1) Zeit,
       round((lead(ZEIT_AB, 1) over (partition by FAHRER order by ZEIT_AB) -
                 ZEIT_AN)*24, 1) Pause
  from FAHRTEN
 order by FAHRER, ORT_AB, ZEIT_AB;
```

Analytische Funktion **LEAD**: Entsprechender Wert aus »einer folgenden« Ergebniszeile

Startzeitpunkt der Folgefahrt

Tabellenzugriffe

Fahrer	Ort_Ab	Ort_An	Zeit	Pause
Aaron	Asperg	Hannover	15.3	1.8
Aaron	Asperg	Düsseldorf	13.3	5.6
Aaron	Asperg	Hannover	15.5	7.2
Aaron	Asperg	Bad Segeberg	12.5	0.9
Aaron	Asperg	Asperg	9	5.8
Aaron	Asperg	Ulm	3.9	7.1
Aaron	Asperg	Chemnitz	5.8	4.9
Aaron	Asperg	Hannover	7	1.9
Aaron	Asperg	Potsdam	12.7	8.2
Aaron	Asperg	Düsseldorf	13.3	9.9
Aaron	Asperg	Stralsund	11.9	7.4
Aaron	Asperg	Potsdam	12.3	9.7
Aaron	Asperg	Mainz	4.4	1.2
Aaron	Asperg	Hannover	9.9	3.2
Aaron	Asperg	Ulm	11	7.2
Aaron	Asperg	Fürth	8.6	
Aaron	Asperg	Hamburg	16.5	
Aaron	Asperg	Mainz	1(	
Aaron	Asperg	Stralsund		
			...	

LEAD (bzw. das im Prinzip funktionsgleiche **LAG**) sind für Zugriffe relativ zur aktuellen Zeile äußerst nützlich. Ab Version 11.2 steht diese Funktionalität endlich auch für den Beginn und das Ende des **»analytischen Fensters«** zur Verfügung: **NTH_VALUE** erweitert die Funktionen **FIRST_VALUE** und **LAST_VALUE** (und macht sie entbehrlich).

Es gibt dabei folgenden Fallstrick: Das analytische Fenster reicht, ohne abweichende Festlegung, von der ersten bis zur aktuellen Zeile. Also steht z.B. in der ersten Zeile der Ergebnismenge die zweite Zeile gar nicht zur Verfügung! Im oberen Beispiel kann man sehen, dass beim jeweils ersten Eintrag nichts berechnet wird (NULL); das untere Beispiel zeigt die »richtige« Formulierung.

Es sollen zu jedem Fahrer, dessen Name mit C-E beginnt, die Ankunftszeiten und die Dauer der ersten Pause ausgegeben werden:

```
select FAHRER, ZEIT_AN,
       round((
         nth_value(ZEIT_AB, 2) over (partition by FAHRER order by ZEIT_AB) -
         nth_value(ZEIT_AN, 1) over (partition by FAHRER order by ZEIT_AB)
           )*24, 1) ErstePause
  from FAHRTEN
 where regexp_like(FAHRER, '^[c-e]', 'imx')
 order by ZEIT_AB;
```

Startzeit der ersten Pause: Endezeit der ersten Fahrt

Endezeit der ersten Pause: Startzeit der zweiten Fahrt

Fahrer von C-E

Der erste Eintrag pro Fahrer ist leer — die Startzeit der zweiten Fahrt steht nicht zur Verfügung

Fahrer	Zeit An	Erste Pause
Dietmar	11.01.2013 15:14:00	
David	11.01.2013 08:18:00	
Christian	12.01.2013 03:11:00	
David	11.01.2013 17:10:00	1.6
Dietmar	12.01.2013 04:15:00	1.9
David	11.01.2013 23:43:00	1.6
David	12.01.2013 05:40:00	1.6
Christian	13.01.2013 03:52:00	1.3
Dietmar	12.01.2013 17:42:00	1.9
David	13.01.2013 10:50:00	1.6
Dietmar	13.01.2013 19:13:00	1.9
Christian	14.01.2013 02:59:00	1.3
David	14.01.2013 01:11:00	1.6
...		

Alle Zeilen der aktuellen Partition nach Fahrer werden berücksichtigt

Überall der richtige Eintrag zum Fahrer

```
select FAHRER, ZEIT_AN,
       round((
         nth_value(ZEIT_AB, 2) over (partition by FAHRER order by ZEIT_AB
                rows between unbounded preceding and unbounded following) -
         nth_value(ZEIT_AN, 1) over (partition by FAHRER order by ZEIT_AB
                rows between unbounded preceding and unbounded following)
           )*24, 1) ErstePause
  from FAHRTEN
 where regexp_like(FAHRER, '^[c-e]', 'imx')
 order by ZEIT_AB;
```

Fahrer	Zeit An	Erste Pause
Dietmar	11.01.2013 15:14:00	1.9
David	11.01.2013 08:18:00	1.6
Christian	12.01.2013 03:11:00	1.3
David	11.01.2013 17:10:00	1.6
Dietmar	12.01.2013 04:15:00	1.9
David	11.01.2013 23:43:00	1.6
David	12.01.2013 05:40:00	1.6
Christian	13.01.2013 03:52:00	1.3
Dietmar	12.01.2013 17:42:00	1.9
David	13.01.2013 10:50:00	1
Dietmar	13.01.2013 19:13:0(	
...		

Sobald **ORDER BY** verwendet wird, endet das **»Analytische Fenster«**, wenn nichts anderes festgelegt wird, bei der aktuellen Zeile. Dies läßt sich sehr deutlich am Beispiel mit der häufig verwendeten **COUNT**-Funktion (die im übrigen auch als Gruppenfunktion eingesetzt werden kann) darstellen.

```
select FAHRER, ZEIT_AN,
       count(1) over (partition by FAHRER)  FAHRTEN_GESAMT,
       count(1) over (partition by FAHRER
                      order by ZEIT_AN) FAHRTEN_BISHER,
       count(1) over (partition by FAHRER
                      order by ZEIT_AN
                       rows between current row
                            and unbounded following)-1 NOCH_ZU_FAHREN,
       count(1) over (partition by FAHRER
                      order by ZEIT_AN
                       rows between 2 preceding
                                and 2 following)       FUENF_FAHRTEN,
       count(1) over (partition by FAHRER
                      order by ZEIT_AN
                      range between interval'12'hour preceding
                                and interval'12'hour following) H24,
       count(1) over (partition by FAHRER, trunc(ZEIT_AN)) DIESEN_TAG
  from FAHRTEN
order by FAHRER, ZEIT_AN;
```

Kein **order by**, deshalb werden alle Zeilen zum jeweiligen Fahrer berücksichtigt; das Fenster umfasst die gesamte »Partition«.

Mit **order by** aber ohne weitere Festlegung: Jetzt enthält das Fenster die erste bis zur aktuellen Zeile (entsprechend der Sortierung nach ZEIT_AN und zum jeweiligen Fahrer).

Das Fenster wird mit **between current row and unbounded following** explizit festgelegt: Die aktuelle und alle folgenden Zeilen bis zum Schluss (-1 zum Abziehen der aktuellen Zeile, um die Ergebnisdarstellung zu verbessern).

Mit **preceding** und **following** werden hier bis zu fünf Zeilen ausgewählt: 2 vorhergehende, die aktuelle und 2 folgende (am Anfang und Ende der Partitionen entsprechend weniger).

range statt **rows** schränkt auf Werte um den aktuellen Wert ein. Hier werden die Fahrten im aktuellen Tageszeitraum — 12 Stunden vor und nach der aktuellen ZEIT_AN — gezählt.

Um dagegen auf Kalendertage einzuschränken, ist range nicht geeignet. Dazu muss einfach die Partitionierung entsprechend angepaßt werden.

Fahrer	Zeit_An	Fahrten_Gesamt	Fahrten_Bisher	Noch_Zu_Fahren	Fuenf_Fahrten	H24	Diesen_Tag
			...				
Günther	06.10.2015 14:43:00	1389	1386	3	5	3	3
Günther	06.10.2015 23:48:00	1389	1387	2	5	2	3
Günther	07.10.2015 16:11:00	1389	1388	1	4	1	1
Günther	08.10.2015 08:12:00	1389	1389	0	3	1	1
Hans-Josef	11.01.2013 16:26:00	1361	1	1360	3	1	1
Hans-Josef	12.01.2013 19:39:00	1361	2	1359	4	1	1
Hans-Josef	13.01.2013 14:38:00	1361	3	1358	5	1	1
Hans-Josef	14.01.2013 11:19:00	1361	4	1357	5	1	1
			...				

NTH_VALUE funktioniert natürlich nur mit **ORDER BY** (ohne Sortierung keine Reihenfolge); deshalb ist hier stets eine besondere Aufmerksamkeit für das verwendete analytische Fenster nötig. Insbesondere wenn »von hinten« gerechnet wird, was sich mit **FROM LAST** (im Gegensatz zur voreingestellten Einstellung **FROM FIRST**) festlegen lässt, kann es sonst zu Überraschungen kommen.

In manchen Fällen kann auch die Bestimmung **IGNORE NULLS** (im Gegensatz zum normalen **RESPECT NULLS**) nötig sein, also dass »leere Zeilen« nicht mitgezählt werden.

Im abschließenden Beispiel soll zu allen Fahrern der Tag ausgegeben werden, an dem zum letzten Mal eine Fahrt komplett innerhalb eines Tages durchgeführt wurde (also nicht über Mitternacht hinweg):

```
select FAHRER,
       trunc(ZEIT_AN) ANKUNFTSTAG,
       case when UNTERTAGS is not NULL then 'X' else NULL end ISTTAGESFAHRT,
       NTH_VALUE(UNTERTAGS, 1) from LAST ignore NULLS
         over (partition by FAHRER order by ZEIT_AN
                     rows between unbounded preceding
                      and unbounded following ) LETZTE_TAGESFAHRT
  from (
select FAHRER,
       ZEIT_AB,
       ZEIT_AN,
       case when trunc(ZEIT_AB) = trunc(ZEIT_AN) then trunc(ZEIT_AN)
            else NULL end UNTERTAGS
  from FAHRTEN f)
 order by FAHRER, ZEIT_AN;
```

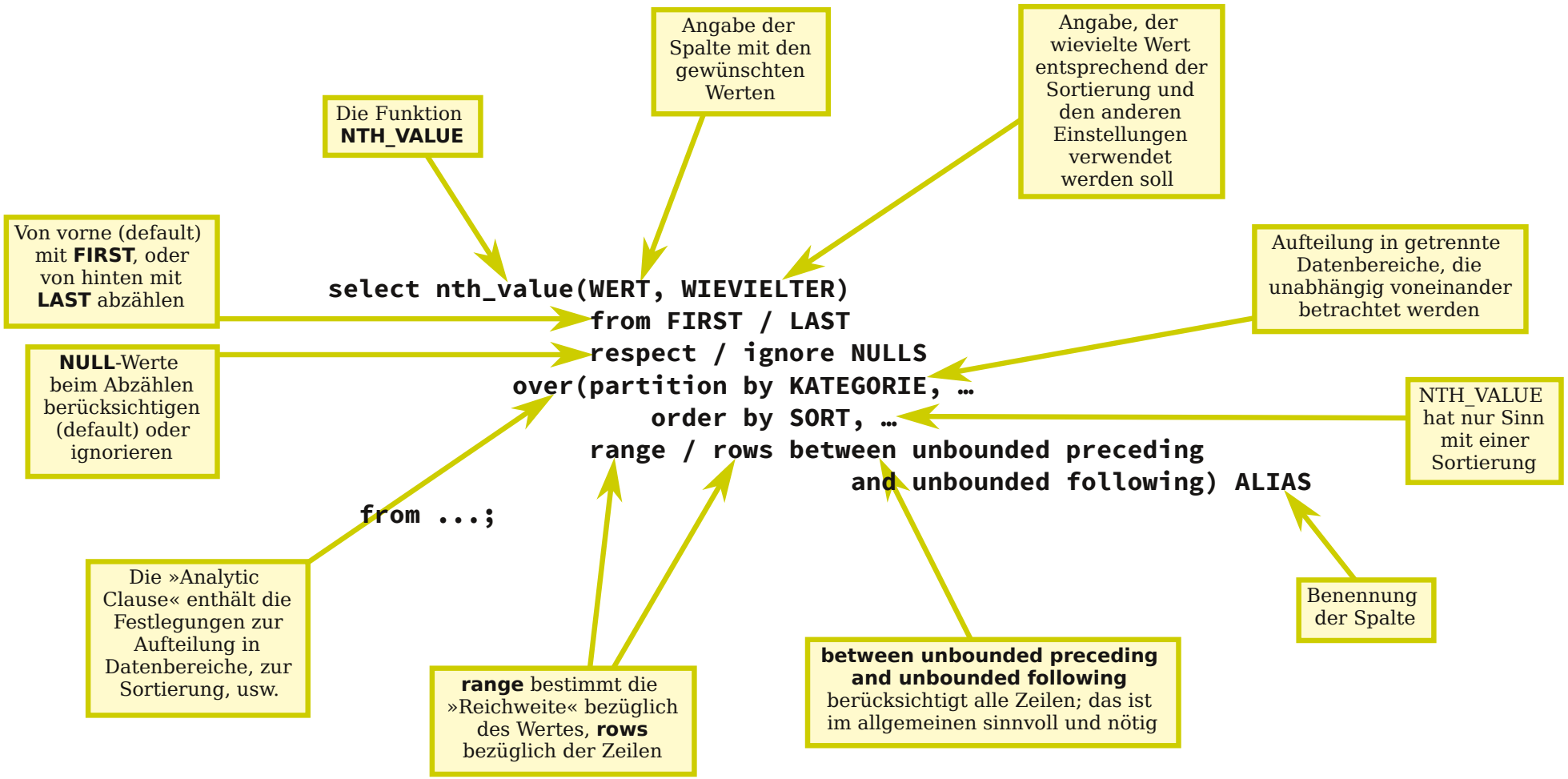
NULL falls über Mitternacht, sonst Datum der Tagesfahrt

Zählung von hinten

NULL-Werte nicht berücksichtigen — also nur Tagesfahrten

Fahrer	Ankunfts-tag	Ist Tages-fahrt	Letzte Tages-fahrt
		...	
Heidi	05.10.2015		06.10.2015
Heidi	05.10.2015	X	06.10.2015
Heidi	06.10.2015	X	06.10.2015
Heidi	06.10.2015	X	06.10.2015
Heidi	07.10.2015		06.10.2015
Heidi	08.10.2015		06.10.2015
Heike	11.01.2013	X	06.10.2015
		...	
Heike	04.10.2015	X	06.10.2015
Heike	05.10.2015		06.10.2015
Heike	05.10.2015	X	06.10.2015
Heike	06.10.2015	X	06.10.2015
Heike	07.10.2015		06.10.2015
Heike	08.10.2015		06.10.2015
Helga	11.01.2013	X	07.10.2015
		...	
Helga	04.10.2015	X	07.10.2015
Helga	05.10.2015		07.10.2015
Helga	06.10.2015		07.10.2015
Helga	06.10.2015	X	07.10.2015
Helga	07.10.2015	X	07.10.20?
Helga	07.10.2015	X	07.10
Helga	08.10.2015		07
Jan	11.01.2013	X	
		...	

Nth Value



Online-Dokumentation (Oracle):
docs.oracle.com/cd/E11882_01/server.112/e41084/functions114.htm#CJAFEJBE

Result Cache

Aufwendig ermittelte Funktions-Ergebnisse lassen sich zwischenspeichern! Und das vollautomatisch! Und alle Sessions können diese Ergebnisse verwenden! Selbstverständlich werden die an die Funktion übergebenen Parameter berücksichtigt, und bei Änderungen der Datenbasis die Ergebnisse neu berechnet. Reicht der vorgesehene Speicherplatz für die zwischengespeicherten Ergebnisse nicht aus, wird aufgeräumt.

Die Verwendung von Result Cache kann damit allerdings zu starken Laufzeitschwankungen und einer erhöhten Systembelastung führen — es gibt keine Kontrolle darüber, ob ein Wert bei einer Funktionsabfrage tatsächlich verwendbar im Cache liegt oder (zeitaufwendig) neu ermittelt werden muss; und die Zwischenergebnisse müssen natürlich auch im Speicher vorgehalten werden.

Result Caching ist besonders gut geeignet für Funktionsaufrufe in Select-Statements: Anstatt für jede Ergebniszeile die Funktion tatsächlich auszuführen, kann auf die zwischengespeicherten Ergebnisse zurückgegriffen werden.

Die Funktion **IstPersonDort** soll überprüfen, ob sich ein Fahrer des angegebenen Namens am angegebenen Ort befindet — also ob die letzte Fahrt dort endete. Die verwendete Abfrage ist ungeschickt formuliert und benötigt entsprechend lange.

```
create or replace function
  IstPersonDort(
    sPerson in varchar2,
    sDort in varchar2)
    return integer is
  iCount integer;
begin
  select count(1)
    into iCount
    from FAHRTEN f
   where FAHRER = sPerson
     and ORT_AN = sDort
     and ZEIT_AN >= (
       select max(ZEIT_AN)
         from FAHRTEN fi
        where fi.FAHRER =
              f.FAHRER);
  return case when iCount = 0
              then 0 else 1 end;
end;
```

```
SQL> declare
  tStart timestamp;
  iCount integer;
begin
  tStart := SYSTIMESTAMP;
  for i in 1 .. 1000 loop
    select IstPersonDort(
      'Willi', 'Planegg')
      into iCount from dual;
  end loop;
  DBMS_OUTPUT.Put_Line
  (SYSTIMESTAMP - tStart);
end;

+000000000 00:00:03.851469000

 PL/SQL procedure successfully
completed.
```

≈ 3,9 Sekunden für 1000 Aufrufe ist nicht schnell

Die winzige Ergänzung **RESULT_CACHE** führt zu einer dramatischen Beschleunigung fast um den Faktor 100 — ohne weitere Anpassungen.

0,041 Sekunden für 1000 Aufrufe ist schnell

```
create or replace function
  IstPersonDort(
    sPerson in varchar2,
    sDort in varchar2)
    return integer
    result_cache is
  iCount integer;
begin
  select count(1)
    into iCount
    from FAHRTEN f
   where FAHRER = sPerson
     and ORT_AN = sDort
     and ZEIT_AN >= (
       select max(ZEIT_AN)
         from FAHRTEN fi
        where fi.FAHRER =
              f.FAHRER);
  return case when iCount = 0
              then 0 else 1 end;
end;
```

```
SQL> declare
  tStart timestamp;
  iCount integer;
begin
  tStart := SYSTIMESTAMP;
  for i in 1 .. 1000 loop
    select IstPersonDort(
      'Willi', 'Planegg')
      into iCount from dual;
  end loop;
  DBMS_OUTPUT.Put_Line
  (SYSTIMESTAMP - tStart);
end;

+000000000 00:00:00.041116000

 PL/SQL procedure successfully
completed.
```

Wenn die Ergebnisse allerdings nicht wiederverwendet werden können, ist die Beschleunigung hinfällig.

1000 verschiedene Orte werden abgefragt

Wieder langsame 3,5 Sekunden für 1000 Aufrufe (bei der ersten Durchführung!)

```
SQL> declare
  tStart timestamp;
  iCount integer;
begin
  tStart := SYSTIMESTAMP;
  for i in 1 .. 1000 loop
    select IstPersonDort(
      'Willi', 'Planegg'||i)
      into iCount from dual;
  end loop;
  DBMS_OUTPUT.Put_Line
  (SYSTIMESTAMP - tStart);
end;

+000000000 00:00:03.532989000

 PL/SQL procedure
successfully completed.
```

Eine geschickte Neuaufteilung der Funktionen ist wesentlich effektiver: Für die Personen werden die Aufenthaltsorte bestimmt und zwischengespeichert; die eigentliche Funktion vergleicht dann nur noch das gespeicherte Ergebnis mit dem übergebenen Ort. Eine entsprechende Abfrage ist wieder richtig schnell.

0,041 Sekunden für 1000 Aufrufe ist schnell

```
create or replace function
  WoIstPerson(
    sPerson in varchar2)
    return varchar2
    result_cache is
  sOrt varchar2(4000);
begin
  select max(ORT_AN)
    into sOrt
    from FAHRTEN f
   where FAHRER = sPerson
     and ZEIT_AN >= (
         select max(ZEIT_AN)
           from FAHRTEN fi
          where fi.FAHRER =
                f.FAHRER);
  return sOrt;
end;
```

```
SQL> declare
  tStart timestamp;
  iCount integer;
begin
  tStart := SYSTIMESTAMP;
  for i in 1 .. 1000 loop
    select IstPersonDort(
      'Willi', 'Stralsund'||i)
      into iCount from dual;
  end loop;
  DBMS_OUTPUT.Put_Line
  (SYSTIMESTAMP - tStart);
end;

+000000000 00:00:00.055221000

 PL/SQL procedure successfully
completed.
```

Bei verschiedenen Personen hilft diese Umstellung dagegen nichts — für ein unter allen Umständen wirklich gutes Ergebnis sollten die tatsächlich vorhandenen Personen-/Ortskombinationen gecacht und dann mit den übergebenen Parametern verglichen werden.

1000 verschiedene Personen werden abgefragt

Wieder langsame 3,5 Sekunden für 1000 Aufrufe

```
create or replace function
  IstPersonDort(
    sPerson in varchar2,
    sDort in varchar2)
    return integer is
begin
  if sDort = WoIstPerson(sPerson)
  then return 1;
  else return 0;
  end if;
end;
```

```
SQL> declare
  tStart timestamp;
  iCount integer;
begin
  tStart := SYSTIMESTAMP;
  for i in 1 .. 1000 loop
    select IstPersonDort(
      'Willi'||i, 'Planegg')
      into iCount from dual;
  end loop;
  DBMS_OUTPUT.Put_Line
  (SYSTIMESTAMP - tStart);
end;

+000000000 00:00:03.532989000

 PL/SQL procedure
successfully completed.
```

Wie lassen sich nun »dünn gesäte« Schlüssel-Wert-Paare — es gibt ja sehr viel mehr denkbare Orte und Namen, als in der Fahrten-Tabelle abgelegt — effektiv cachen?

- Beliebige Namen und Orte als Abfrage-Parameter
- Der Typ einer index-by table zur Verknüpfung zweier Texte wird deklariert
- Eine Tabelle des zuvor deklarierten Typs, zum Ablegen des Aufenthaltsorts zum Fahrer
- Inline-deklarierte Unterfunktion mit einer entsprechenden Tabelle als Rückgabewert
- result_cache — tatsächliche Ausführung nur beim ersten Aufruf oder nach Datenänderungen
- Abfrage aller tatsächlichen Wertepaare von Fahrer und Ort
- Befüllen der index-by table-Tabelle
- Rückgabe der Tabelle — in den Cache
- Beginn der »eigentlichen« Funktion
- Aufruf der Unterfunktion — diese wird nur einmal »wirklich« ausgeführt
- Gibt es den Fahrer?
- Und falls ja — ist er am angegebenen Ort?

```
create or replace function
 IstPersonDort(
 sPerson in varchar2,
 sDort in varchar2)
  return integer is
 type typeTblIsNowIn is
       table of varchar2(4000)
       index by varchar2(4000);
 tblIsNowIn typeTblIsNowIn;
 function WerIstWo
       return typeTblIsNowIn
       result_cache is
  tblIsNowIn typeTblIsNowIn;
 begin
  for rec in (
   select distinct FAHRER,
     NTH_VALUE(ORT_AN, 1) over(
      partition by FAHRER
      order by ZEIT_AN desc) ORT
   from FAHRTEN) loop
   tblIsNowIn(rec.FAHRER) :=
                          rec.ORT;
  end loop;
  return tblIsNowIn;
 end;
begin
 tblIsNowIn := WerIstWo;
 if NOT tblIsNowIn.Exists(sPerson)
     then return 0; end if;
 return case when
   tblIsNowIn(sPerson) = sDort
   then 1
   else 0
   end;
end;
```

Das nebenstehende Beispiel stellt eine fortgeschrittene Musterlösung vor: Sämtliche vorhandenen Datenpaare werden eingelesen und in einer **index-by table** (auch **»associative array«**) abgelegt; und zwar in einer inline-deklarierten Unterfunktion mit result_cache-Anweisung.

```
SQL>declare
  tStart timestamp;
  iCount integer;
begin
  tStart := SYSTIMESTAMP;
  for i in 1 .. 1000 loop
    select IstPersonDort(
      'Willi'||i, 'Stralsund'||i)
      into iCount from dual;
  end loop;
  DBMS_OUTPUT.Put_Line
  (SYSTIMESTAMP - tStart);
end;
```

1. Aufruf, ½ Sekunde

```
+000000000 00:00:00.549800000

PL/SQL procedure successfully
completed.

SQL>declare
  tStart timestamp;
  iCount integer;
begin
  tStart := SYSTIMESTAMP;
  for i in 1 .. 1000 loop
    select IstPersonDort(
      'Egon'||i, 'Mainz'||i)
      into iCount from dual;
  end loop;
  DBMS_OUTPUT.Put_Line
  (SYSTIMESTAMP - tStart);
end;
```

2. Aufruf, 10× schneller

```
+000000000 00:00:00.058198000

PL/SQL procedure
successfully completed.
```

RESULT_CACHE belegt systemweite Datenbankressourcen, und sollte deshalb wirtschaftlich genutzt werden.

Zur Information und Steuerung stehen verschiedene Views, Funktionen und Procedures zur Verfügung.

Folgende Views enthalten Informationen über die Verwendung des Result Cache:

👉 **V$RESULT_CACHE_OBJECTS**
Detaillierte Informationen zu allen gecachten Ergebnissen.

👉 **V$RESULT_CACHE_STATISTICS**
Verschiedene Gesamtstatistiken bezüglich der Benutzung.

👉 **V$RESULT_CACHE_MEMORY**
Detaillierter Status zu allen verwendeten Speicherblöcken.

👉 **V$RESULT_CACHE_DEPENDENCY**
Liste der Abhängigkeiten, also was bei Bearbeitungen ggf. invalidiert wird.

Im Package **DBMS_RESULT_CACHE**:

👉 **MEMORY_REPORT()**
Eine kurze Status-Übersicht der Speicherbelegung.

👉 **FLUSH()**
Löscht alle gespeicherten Werte.

```
select NAME,
       count(1) ImCache,
       sum(nvl(SCAN_COUNT,0))
                Nutzung
  from V$RESULT_CACHE_OBJECTS
 where TYPE = 'Result'
 group by NAME
 order by 2 desc, 1;
```

Name	Im Cache	Nutzung
"SCHULUNG"."ISTPERSONDORT"::8."ISTPERSONDORT"#5ae8425de6797f6f #2	1000	0
"SCHULUNG"."ISTPERSONDORT"::8."ISTPERSONDORT.WERISTWO"#f4800b9721a15048 #10	1	999

```
select *
  from V$RESULT_CACHE_STATISTICS;
```

ID	Name	Value
1	Block Size (Bytes)	1024
2	Block Count Maximum	10496
3	Block Count Current	1024
4	Result Size Maximum (Blocks)	524
5	Create Count Success	1002
6	Create Count Failure	0
7	Find Count	2998
8	Invalidation Count	1001
9	Delete Count Invalid	1
10	Delete Count Valid	0
11	Hash Chain Length	1
12	Find Copy Count	2998
13	Latch (Share)	0

```
begin
DBMS_RESULT_CACHE.MEMORY_REPORT();
end;
/
```

```
R e s u l t   C a c h e   M e m o r y   R
[Parameters]
Block Size          = 1K bytes
Maximum Cache Size  = 10496K bytes (10496
Maximum Result Size = 524K bytes (524 blo
[Memory]
Total Memory = 161312 bytes [0.009% of the
... Fixed Memory =  bytes [% of the Shared Pool]
... Dynamic Memory = 161312 bytes [0.009% of the Shared Pool]
....... Overhead = 128544 bytes
....... Cache Memory = 32K bytes (32 blocks)
.......... Unused Memory = 28 blocks
.......... Used Memory = 4 blocks
.............. Dependencies = 2 blocks (2 count)
.............. Results = 2 blocks
................. PLSQL   = 2 blocks (1 count).
```

Result Cache

Die Verwendung des Schlüsselwortes **RESULT_CACHE** reicht aus — der Rest funktioniert automatisch.

```
create or replace function NAME
  ( ParameterA in typ,
    ParameterB in typ,
    … )
    return typ
    result_cache is
    VariableA typ;
    …;
  begin
    …
  end;
```

Online-Dokumentation (Oracle):
docs.oracle.com/cd/E11882_01/appdev.112/e25519/subprograms.htm#LNPLS00817

Anhang: Skripte

Füllen der Tabelle mit Musterdaten

```
declare
  type typeTblVarchar is table of varchar2(4000);
  tblFahrer typeTblVarchar := typeTblVarchar(
    'Aaron', 'Abdellahi', 'Adrian', 'Alexander', 'Andrea', 'Andreas',
    'Anke', 'Anselm', 'Antonia', 'Bela', 'Benjamin', 'Bernd',
    'Birgitta', 'Christian', 'David', 'Dietmar', 'Florian', 'Günther',
    'Gerhard', 'Hans-Josef', 'Heidi', 'Heike', 'Helga', 'Jan', 'Jens',
    'Josef', 'Julia', 'Marcel', 'Maria', 'Marko', 'Michael', 'Michaela',
    'Monia', 'Nikolaj', 'Oleg', 'Oliver', 'Ralf', 'Robert', 'Saad',
    'Sabine', 'Sebastian', 'Stephan');
  tblOrte   typeTblVarchar := typeTblVarchar(
    'Stralsund', 'Bad Segeberg', 'Hamburg', 'Hannover', 'Potsdam',
    'Herne', 'Düsseldorf', 'Chemnitz', 'Mainz', 'Fürth', 'Saarbrücken',
    'Asperg', 'Offenburg', 'Ulm', 'Planegg', 'Tuttlingen');
  iCount integer := 0;
  dDate date;          iOrtAb integer;
  iMinuten integer;    iOrtAn integer;
begin
  delete from FAHRTEN;
  for iFahrer in tblFahrer.First .. tblFahrer.Last loop
    dDate := trunc(SYSDATE) - interval'1000'day + (10 + mod(ADMIN.DBA_TOOL.Random, 600))/1440;
    iOrtAb := 1 + mod(ADMIN.DBA_TOOL.Random, tblOrte.Count);
    while dDate < trunc(SYSDATE) loop
      iCount := iCount + 1;
      iOrtAn := 1 + mod(ADMIN.DBA_TOOL.Random, tblOrte.Count);
      iMinuten := 10 + 45*(abs(iOrtAn-iOrtAb)) + mod(ADMIN.DBA_TOOL.Random, 600)*(1+iFahrer/30);
      insert into FAHRTEN values (iCount, tblFahrer(iFahrer), dDate, tblOrte(iOrtAb), dDate + iMinuten/1440,
                                  tblOrte(iOrtAn));
      dDate := dDate + iMinuten/1440 + (10 + mod(ADMIN.DBA_TOOL.Random, 600))/1440;
      iOrtAb := iOrtAn;
    end loop;
  end loop;
  commit;
end;
/
```

Tabelle FAHRTEN

```
create table FAHRTEN
(
  ID         integer not null primary key,
  FAHRER     varchar2(4000),
  ZEIT_AB    date,
  ORT_AB     varchar2(4000),
  ZEIT_AN    date,
  ORT_AN     varchar2(4000)
)
tablespace USERS;

create index FAHRTEN_NX001_DAYAN on
             FAHRTEN(trunc(ZEIT_AN));
create index FAHRTEN_NX002_DAYAB on
             FAHRTEN(trunc(ZEIT_AB));
```