IN versus EXISTS oder doch ein Join?

Andrej Pashchenko
Trivadis GmbH
Düsseldorf

Schlüsselworte
Unterabfrage, Subquery, Semi-Join, Anti-Join, Unnesting, Performance, Best Practice

Einleitung
Wenn man über das Schreiben effizienter SQL-Abfragen spricht, gibt es kaum ein Thema, das dermaßen mit Vorurteilen besetzt ist, wie der Einsatz von Unterabfragen mit (NOT) IN oder (NOT) EXISTS. Die Meinungen und Erfahrungen von SQL-Entwicklern gehen häufig ziemlich auseinander. Das Spektrum reicht von unflexiblem Einsatz nur einer bestimmten Variante über Auswahl der Methode nach bestimmten Best Practices bis hin zu Verzicht von Unterabfragen und deren Ersatz durch einen Join. Mit jeder neuen Datenbankversion macht der Oracle Optimizer einen immer besseren Job. Sind die Best Practices von gestern noch aktuell und anwendbar? Kann man überhaupt eine allgemeingültige Best Practice ausformulieren? Im Vortrag wird auf die Themen wie Subquery Unnesting, Semi- und Anti-Joins, deren Unterschied zu klassischen Joins, Join-Methoden und Performance eingegangen.

Semi-Joins
Bevor wir uns den Performance-Aspekten widmen, schauen wir uns zunächst ein paar theoretische Grundlagen an. Es ist wichtig zu verstehen, welche semantische Bedeutung die Abfragen haben und als Folge, ob und unter welchen Bedingungen sie austauschbar sind.

SQL-Abfragen mit Unterabfragen, die durch IN oder EXISTS ausgedrückt sind, stellen einen sogenannten Semi-Join dar. Der Unterschied zu einem herkömmlichen Join von zwei Tabellen - der linken und der rechten - besteht darin, dass aus der linken Tabelle höchstens ein Datensatz zurückgeliefert werden kann, unabhängig von der Anzahl der Treffer in der rechten Tabelle. Außerdem können aus der rechten Tabelle keine Daten selektiert werden: sie dient ausschließlich zur Einschränkung der Ergebnisse.

Wir werden unsere Beispiele auf dem mit einer Oracle Datenbank mitgelieferten Beispielschema SH (Sales History) aufbauen. Der einzige kleine Unterschied: Die Tabelle SALES wurde für die Zwecke dieses Artikels ohne Partitionen neu angelegt. Dies macht die Ausführungspläne deutlich kompakter und besser lesbar.

Szenario: Wir müssen folgende Frage beantworten: Wie viele Kunden aus Köln haben bereits Bestellungen aufgegeben? Im Listing 1 stellen wir mögliche SQL-Abfragen zusammen, die diese Frage beantworten sollen.

```
--A
SELECT  c.cust_last_name, c.cust_first_name, c.cust_id
FROM    sh.customers c
WHERE   c.cust_city = 'Koeln'
AND     EXISTS (SELECT  1
                FROM    sh.sales s
                WHERE   c.cust_id = s.cust_id );
--B
SELECT  c.cust_last_name, c.cust_first_name, c.cust_id
FROM    sh.customers c
WHERE   c.cust_city = 'Koeln'
AND     c.cust_id IN (SELECT  s.cust_id
                      FROM    sh.sales s );
--C
SELECT  c.cust_last_name, c.cust_first_name, c.cust_id
```

```
FROM    sh.customers c JOIN sh.sales s  ON (c.cust_id = s.cust_id)
WHERE   c.cust_city = 'Koeln';

--D
SELECT  DISTINCT c.cust_last_name, c.cust_first_name
,       c.cust_id
FROM    sh.customers c JOIN sh.sales s ON (c.cust_id = s.cust_id)
WHERE   c.cust_city = 'Koeln';
```

Die Abfragen A und B aus dem Listing 1 sind semantisch identisch. Der Join in der Abfrage C liefert dagegen mehr Datensätze zurück als nötig, und zwar, wenn einige Kunden aus Köln mehrere Käufe getätigt haben. Um die Semantik von den Abfragen A und B beizubehalten, benötigt man noch eine DISTINCT-Operation: die Abfrage D ist dann wieder semantisch gleich.

Wenn man sich vorstellen würde, dass die Ausführung einer SQL-Abfrage eins zu eins der Syntax folgen wird, erscheinen die Abfragen A und B auf den ersten Blick im Vergleich zum Join sehr ineffizient. Das könnte auch der Grund für viele Vorurteile sein. Die Unterabfrage wird einmal für jeden Datensatz aus der Tabelle CUSTOMERS ausgeführt.

Wir können dieses Verhalten durch den Optimizer-Hint NO_UNNEST erzwingen. Dabei verbieten wir dem Optimizer, weitere Optimierungen vorzunehmen. Der Ausführungsplan sieht dann so aus:

```
SQL> SELECT  c.cust_last_name, c.cust_first_name, c.cust_id
  2  FROM    sh.customers c
  3  WHERE   c.cust_city = 'Koeln'
  5  AND     EXISTS (SELECT /*+ no_unnest */ 1
  6                  FROM    sh.sales s
  7                  WHERE   c.cust_id = s.cust_id );
...
44 rows selected.
SQL> SELECT * FROM TABLE(DBMS_XPLAN.DISPLAY_CURSOR(NULL, 0, 'ALLSTATS LAST'));
...
-------------------------------------------------------------------------------
| Id  | Operation          | Name      | Starts | E-Rows | A-Rows |   A-Time   |
-------------------------------------------------------------------------------
|   0 | SELECT STATEMENT   |           |      1 |        |     44 |00:01:32.64 |
|*  1 |  FILTER            |           |      1 |        |     44 |00:01:32.64 |
|*  2 |   TABLE ACCESS FULL| CUSTOMERS |      1 |     13 |    532 |00:00:00.12 |
|*  3 |   TABLE ACCESS FULL| SALES     |    532 |      2 |     44 |00:01:32.42 |
-------------------------------------------------------------------------------
```

Listing 2:Korrelierte Unterabfrage ohne Unnesting

Dabei kann man folgendes erkennen. Es sind insgesamt 532 Kunden aus Köln („A-Rows“ in Zeile 2). Die Unterabfrage wurde einmal für jeden Datensatz aus CUSTOMERS gestartet („Starts“ in Zeile 3). Der Ausführungsplan für die Abfrage B sieht übrigens exakt gleich aus. Wie verhält sich der Join?

```
SQL> SELECT  DISTINCT c.cust_last_name, c.cust_first_name, c.cust_id
  2  FROM    sh.customers c JOIN sh.sales s ON (c.cust_id = s.cust_id)
  3  WHERE   c.cust_city = 'Koeln';
...
44 rows selected.
SQL> SELECT * FROM TABLE(DBMS_XPLAN.DISPLAY_CURSOR(NULL, 0, 'ALLSTATS LAST'));
...
-------------------------------------------------------------------------------
| Id  | Operation           | Name      | Starts | E-Rows | A-Rows |   A-Time   |
-------------------------------------------------------------------------------
|   0 | SELECT STATEMENT    |           |      1 |        |     44 |00:00:14.86 |
|   1 |  HASH UNIQUE        |           |      1 |   1099 |     44 |00:00:14.86 |
|*  2 |   HASH JOIN         |           |      1 |   1716 |   4208 |00:00:14.82 |
|*  3 |    TABLE ACCESS FULL| CUSTOMERS |      1 |     13 |    532 |00:00:00.05 |
```

```
|   4 |    TABLE ACCESS FULL| SALES     |      1 |    918K|    918K|00:00:03.86 |
--------------------------------------------------------------------------------
```

Listing 3: Klassischer Join

Der CBO hat sich hier für einen Hash-Join entschieden. Die Laufzeit ging von 1,5 Minuten auf knapp 15 Sekunden herunter. Also ist der klassische Join tatsächlich die bessere Wahl? Nein! Denn meistens ist der Optimizer in der Lage, ineffiziente Ausführungspläne, wie im Listing 2, auch bei IN- oder EXISTS-Unterabfragen zu vermeiden.

Subquery Unnesting

Bevor der Optimizer zum Auswerten der Zugriffsmethoden und Kosten kommt, führt er mögliche Query Transformationen durch, die die Semantik der Abfrage nicht verändern, dafür aber, zum Beispiel, weitere Zugriffsmöglichkeiten erlauben. Im Fall von Unterabfragen wie in A und B, kommt sogenanntes Subquery Unnesting als Transformation infrage.

Die Tabelle aus der Unterabfrage wird in die Hauptabfrage „eingezogen“ und es wird ein Semi-Join durchgeführt. Der Ausführungsplan sieht dann zum Beispiel wie im Listing 4 aus:

```
SQL> SELECT  c.cust_last_name, c.cust_first_name, c.cust_id
  2  FROM    sh.customers c
  3  WHERE   c.cust_city = 'Koeln'
  5  AND     EXISTS (SELECT  1
  6                  FROM    sh.sales s
  7                  WHERE   c.cust_id = s.cust_id );
...
44 rows selected.
...
--------------------------------------------------------------------------------
| Id  | Operation          | Name      | Starts | E-Rows | A-Rows |   A-Time   |
--------------------------------------------------------------------------------
|   0 | SELECT STATEMENT   |           |      1 |        |     44 |00:00:14.62 |
|*  1 |  HASH JOIN SEMI    |           |      1 |    514 |     44 |00:00:14.62 |
|*  2 |   TABLE ACCESS FULL| CUSTOMERS |      1 |    532 |    532 |00:00:00.05 |
|   3 |   TABLE ACCESS FULL| SALES     |      1 |    918K|    918K|00:00:03.86 |
--------------------------------------------------------------------------------
```

Listing 4:Semi-Join

Der Unterschied in der Laufzeit zu einem Join ist kaum da. Diese Art der Abfrage hat aber einige Vorteile. Zum einen, kann man erkennen, dass man sich den Schritt HASH UNIQUE sparen kann, weil Semi Joins per Definition keine Dubletten zurückliefert. Zum zweiten, machen die Semi-Join Algorithmen von dieser Tatsache Gebrauch und führen tendenziell weniger Arbeit durch. Die Weiterverarbeitung für einen Satz aus der äußeren Tabelle wird unterbrochen, sobald ein Treffer in der inneren Tabelle gefunden wurde.

Das kann man sich noch gut vorstellen, wenn der Join mit Nested Loops durchgeführt wird. Aber wie funktioniert es bei einem Hash Join? Bei einem Hash Join wird eine Tabelle (in der Regel die kleinere) als treibende gewählt, auf die Join-Kriterien eine Hash-Funktion angewendet und damit eine Hash-Tabelle erzeugt. Dann wird die getriebene Tabelle gescannt, dieselbe Hash-Funktion auf die Join-Kriterien angewendet und mit dem Ergebnis die Hash-Tabelle geprobt. Bei einem Treffer werden die Join-Kriterien exakt abgeglichen und der Datensatz bei Übereinstimmung zurückgeliefert. Die Optimierung des Semi-Joins besteht darin, dass beim ersten Treffer der Eintrag aus der Hash-Tabelle gelöscht wird. Weitere potenziellen Treffer finden den Eintrag in der Hash-Tabelle nicht mehr und werden abgewiesen – somit entfällt die Notwendigkeit, das Ergebnis am Ende noch einmal zu deduplizieren, wie es bei einem normalen Join der Fall wäre. Des Weiteren, wenn durch dieses Löschen die Hash-Tabelle leer wird, stoppt die komplette Verarbeitung, noch bevor die getriebene Tabelle komplett gescannt wurde.

Am Beispiel im Listing 5 sieht man es deutlich. Wir haben die Menge extra so eingeschränkt, dass nur ein Datensatz aus der Kundentabelle zutrifft. Während bei einem normalen Hash-Join die Tabelle Sales komplett gelesen wurde („A-Rows“ = 918K), wurden bei einem Hash-Semi-Join nur 13567 Datensätze gelesen, bis der Treffer für unseren Kunden gefunden wurde. Dann war die Abfrage fertig: das weitere Lesen der Tabelle Sales kann das Ergebnis der Abfrage nicht mehr beeinflussen und wird gestoppt. Das lässt sich auch direkt an der Laufzeit merken - 15 Sekunden versus 0,3 Sekunden.

```
SQL> SELECT  DISTINCT c.cust_last_name, c.cust_first_name, c.cust_id
  2  FROM    sh.customers c JOIN sh.sales s ON (c.cust_id = s.cust_id)
  3  WHERE   c.cust_city = 'Koeln'
  5  AND     c.cust_last_name LIKE 'Nappi%';
Nappier                                Beryl                     2397

...
-------------------------------------------------------------------------------------
| Id  | Operation           | Name      | Starts | E-Rows | A-Rows |   A-Time   |
-------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT    |           |      1 |        |      1 |00:00:14.99 |
|   1 |  HASH UNIQUE        |           |      1 |      1 |      1 |00:00:14.99 |
|*  2 |   HASH JOIN         |           |      1 |      2 |     86 |00:00:14.99 |
|*  3 |    TABLE ACCESS FULL| CUSTOMERS |      1 |      1 |      1 |00:00:00.04 |
|   4 |    TABLE ACCESS FULL| SALES     |      1 |    918K|    918K|00:00:04.13 |
-------------------------------------------------------------------------------------
SQL> SELECT  c.cust_last_name, c.cust_first_name, c.cust_id
  2  FROM    sh.customers c
  3  WHERE   c.cust_city = 'Koeln'
  5  AND     c.cust_last_name LIKE 'Nappi%'
  6  AND     c.cust_id IN (SELECT  s.cust_id
  7                        FROM    sh.sales s );
Nappier                                Beryl                     2397

...
-------------------------------------------------------------------------------------
| Id  | Operation           | Name      | Starts | E-Rows | A-Rows |   A-Time   |
-------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT    |           |      1 |        |      1 |00:00:00.28 |
|*  1 |  HASH JOIN SEMI     |           |      1 |      1 |      1 |00:00:00.28 |
|*  2 |   TABLE ACCESS FULL| CUSTOMERS |      1 |      1 |      1 |00:00:00.06 |
|   3 |   TABLE ACCESS FULL| SALES     |      1 |    918K|  13567 |00:00:00.06 |
-------------------------------------------------------------------------------------
```

Listing 5:Semi-Join im Vergleich zu Join

Durch Subquery Unnesting bekommt der Optimizer die Möglichkeit, außer Filter-Operation die bekannten Join-Methoden Nested Loops, Hash oder Sort/Merge je nach Mengengerüsten und Join-Kriterien anzuwenden. Auch die Join-Reihenfolge kann noch optimiert werden. Bei der Operation HASH JOIN SEMI wird die äußere Tabelle immer als Hash Tabelle verwendet, auch wenn sie größer ist als die innere. Ab Version 10g gibt es eine Verbesserung: HASH JOIN RIGHT SEMI. Bei dieser Operation ist es möglich, die Hash-Tabelle aus der Unterabfrage zu erzeugen.

Listing 5 zeigt auch, dass die IN-Unterabfrage genauso vom Subquery Unnesting profitiert. Sie wird intern in die korrelierte EXISTS-Form transformiert.

Wir haben gesehen: Wenn Subquery Unnesting stattfindet, ist der Optimizer in der Lage, effiziente Wege für Zugriffsmethoden, Join-Methoden sowie Join-Reihenfolge zu finden. Und dies unabhängig davon, ob die Abfrage mit IN oder EXISTS formuliert ist, oder davon, welche Seite stärkere Selektivität aufweist. Man sollte den CBO einfach seine Arbeit machen lassen.

Anti Join

Wie sieht es mit Unterabfragen aus, die mit NOT IN oder NOT EXISTS an die Hauptabfrage angebunden sind? In der Tabelle CUSTOMERS sind alle Kunden in die Einkommensgruppen eingeteilt. Wie beantworten wir die Frage, ob unter den Kunden in Hamburg alle Einkommensgruppen vertreten sind, die es auch bei Kölnern gibt? Vorstellbar sind zwei Varianten mit Unterabfragen (Listing 6, A und B).

```
--A
SQL> SELECT  DISTINCT c.cust_income_level
  2  FROM    sh.customers c
  3  WHERE   c.cust_city = 'Koeln'
  5  AND     NOT EXISTS (SELECT 1
  6                      FROM   sh.customers c2
  7                      WHERE  c2.country_id = 52776
  8                      AND    c2.cust_city = 'Hamburg'
  9                      AND    c2.cust_income_level=c.cust_income_level);
K: 250,000 - 299,999

--B
SQL> SELECT  DISTINCT c.cust_income_level
  2  FROM    sh.customers c
  3  WHERE   c.cust_city = 'Koeln'
  5  AND     c.cust_income_level NOT IN (SELECT   c2.cust_income_level
  6                                      FROM   sh.customers c2
  7                                      WHERE  c2.country_id = 52776
  8                                      AND    c2.cust_city = 'Hamburg');
K: 250,000 - 299,999

-- Update auf NULL
SQL> UPDATE  sh.customers c
  2  SET     c.cust_income_level = NULL
  3  WHERE   c.cust_city = 'Hamburg'
  5  AND     rownum = 1;

1 row updated.

--B erneut ausführen
SQL> SELECT  DISTINCT c.cust_income_level
  2  FROM    sh.customers c
  3  WHERE   c.cust_city = 'Koeln'
  5  AND     c.cust_income_level NOT IN (SELECT   c2.cust_income_level
  6                                      FROM   sh.customers c2
  7                                      WHERE  c2.country_id = 52776
  8                                      AND    c2.cust_city = 'Hamburg');

no rows selected
```

Listing 6: Anti-Joins

Hierbei ist wichtig zu verstehen, dass diese Abfragen semantisch nicht gleich sind. Der Unterschied besteht in der Behandlung von NULL-Werten. Wir setzen die Einkommensgruppe bei einem beliebigen Kunden aus Hamburg auf NULL und starten die Abfrage B erneut. Sie liefert kein Ergebnis. Dies ist keineswegs unerwartet und liegt an der Art und Weise, wie NULL-Werte generell in SQL behandelt werden. Während die NOT EXISTS-Variante die NULL-Werte durch die Where-Bedingung C2.CUST_INCOME_LEVEL=C.CUST_INCOME_LEVEL gar nicht in Betracht zieht, können bei NOT IN die NULLS sowohl in der Hauptabfrage als auch in der Unterabfrage dazu führen, dass das Ergebnis unbekannt ist.

Wie werden die Abfragen A und B ausgeführt? Hier muss die Datenbank einen sogenannten Anti-Join durchführen. Dabei werden Datensätze zurückgeliefert die das Join-Kriterium nicht erfüllen. Die Variante mit NOT EXISTS ist nicht von der Null-Werte-Problematik betroffen. Hier kann das oben

beschriebene Subquery Unnesting stattfinden und dem Optimizer stehen die bekannten Anti-Join Methoden zur Verfügung: Nested Loops, Hash oder Sort/Merge Anti-Joins.

Ohne Subquery Unnesting bei der Variante mit NOT IN sehen wir wieder die bekannte FILTER-Operation im Ausführungsplan (Listing 8 A). Diese Operation ist schlecht skalierbar, wobei man auch hier eine Art Optimierung beobachten kann. Im Schritt 3 wurden 532 Datensätze selektiert. Man würde doch erwarten, dass die Unterabfrage auch 532 Mal ausgeführt werden soll? Sie wird aber nur 12 Mal gestartet (Spalte „Starts“ für den Schritt 4). Sie wird somit nur einmalig für jeden eindeutigen Wert des Join-Kriteriums (CUST_INCOME_LEVEL) gestartet.

Damit Subquery Unnesting auch mit NOT IN funktioniert, muss man in den Versionen vor 11g sicherstellen, dass keine NULLs auf beiden Seiten des Joins vorkommen. Dies kann man durch Constraints erreichen, aber auch durch entsprechende Filter-Bedingungen wie im Listing 8 (B). In diesem Fall kann die Datenbank Subquery Unnesting durchführen und eine der Anti-Join Methoden anwenden, in diesem Fall Hash-Anti-Join.

Ab 11g kann die Datenbank die sogenannten Null-Aware Anti-Joins durchführen. Auch wenn NULLs zu erwarten sind, kann die Datenbank einen Hash-Anti-Join verwenden (siehe Listing 9). Die Join-Operation erscheint im Ausführungsplan als HASH JOIN ANTI NA. Sind nur auf einer Seite des Joins die Null-Werte zu erwarten, geht es um Single Null-Aware Anti-Join. Den sieht man als HASH JOIN ANTI SNA.

```
--A
SQL> alter session set optimizer_features_enable='10.2.0.4';
Session altered.
SQL>
SQL> SELECT  DISTINCT c.cust_income_level
  2  FROM    sh.customers c
  3  WHERE   c.cust_city = 'Koeln'
  5  AND     c.cust_income_level NOT IN (SELECT c2.cust_income_level
  6                                      FROM   sh.customers c2
  7                                      WHERE  c2.country_id = 52776
  8                                      AND    c2.cust_city = 'Hamburg');

no rows selected

-------------------------------------------------------------------------------------
| Id  | Operation           | Name      | Starts | E-Rows | A-Rows |   A-Time   |
-------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT    |           |      1 |        |      0 |00:00:00.07 |
|   1 |  HASH UNIQUE        |           |      1 |      8 |      0 |00:00:00.07 |
|*  2 |   FILTER            |           |      1 |        |      0 |00:00:00.07 |
|*  3 |    TABLE ACCESS FULL| CUSTOMERS |      1 |     13 |    532 |00:00:00.04 |
|*  4 |    TABLE ACCESS FULL| CUSTOMERS |     12 |      1 |     12 |00:00:00.02 |
-------------------------------------------------------------------------------------

--B
SQL> SELECT  DISTINCT c.cust_income_level
  2  FROM    sh.customers c
  3  WHERE   c.cust_city = 'Koeln'
  5  AND     c.cust_income_level IS NOT NULL
  6  AND     c.cust_income_level NOT IN (SELECT c2.cust_income_level
  7                                      FROM   sh.customers c2
  8                                      WHERE  c2.country_id = 52776
  9                                      AND    c2.cust_city = 'Hamburg'
 10                                      AND    c2.cust_income_level IS NOT NULL);
K: 250,000 - 299,999

-------------------------------------------------------------------------------------
| Id  | Operation           | Name      | Starts | E-Rows | A-Rows |   A-Time   |
-------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT    |           |      1 |        |      1 |00:00:00.12 |
```

```
|   1 |  HASH UNIQUE        |           |      1 |      1 |      1 |00:00:00.12 |
|*  2 |   HASH JOIN ANTI    |           |      1 |      1 |     10 |00:00:00.12 |
|*  3 |    TABLE ACCESS FULL| CUSTOMERS |      1 |     13 |    532 |00:00:00.06 |
|*  4 |    TABLE ACCESS FULL| CUSTOMERS |      1 |     13 |     43 |00:00:00.04 |
--------------------------------------------------------------------------------
```

Listing 8: Wirkung von NULLs vor 11g

Natürlich führt die Verwendung von einer Null-Aware Anti-Join Operation nicht dazu, dass wir doch ein Ergebnis bekommen, sollten im Join-Kriterium tatsächlich Null-Werte vorkommen. Die Semantik von der SQL-Sprache bleibt natürlich erhalten. Das kann man gut erkennen, wenn man die Abfragen A und B aus dem Listing 9 vergleicht.

```
SQL> alter session set optimizer_features_enable='11.1.0.6';

Session altered.

--A

SQL> SELECT  DISTINCT c.cust_income_level
  2  FROM    sh.customers c
  3  WHERE   c.cust_city = 'Koeln'
  5  AND     c.cust_income_level NOT IN (SELECT c2.cust_income_level
  6                                      FROM   sh.customers c2
  7                                      WHERE  c2.country_id = 52776
  8                                      AND    c2.cust_city = 'Hamburg');

no rows selected

--------------------------------------------------------------------------------
| Id  | Operation           | Name      | Starts | E-Rows | A-Rows |   A-Time   |
--------------------------------------------------------------------------------
|   0 | SELECT STATEMENT    |           |      1 |        |      0 |00:00:00.08 |
|   1 |  HASH UNIQUE        |           |      1 |      1 |      0 |00:00:00.08 |
|*  2 |   HASH JOIN ANTI NA |           |      1 |      1 |      0 |00:00:00.08 |
|*  3 |    TABLE ACCESS FULL| CUSTOMERS |      1 |     13 |    532 |00:00:00.05 |
|*  4 |    TABLE ACCESS FULL| CUSTOMERS |      1 |     13 |      1 |00:00:00.01 |
--------------------------------------------------------------------------------

--B

SQL> SELECT  DISTINCT c.cust_income_level
  2  FROM    sh.customers c
  3  WHERE   c.cust_city = 'Koeln'
  5  AND     c.cust_income_level NOT IN (SELECT c2.cust_income_level
  6                                      FROM   sh.customers c2
  7                                      WHERE  c2.country_id = 52776
  8                                      AND    c2.cust_city = 'Hamburg'
  9                                      AND    c2.cust_income_level IS NOT NULL);
K: 250,000 - 299,999

--------------------------------------------------------------------------------
| Id  | Operation           | Name      | Starts | E-Rows | A-Rows |   A-Time   |
--------------------------------------------------------------------------------
|   0 | SELECT STATEMENT    |           |      1 |        |      1 |00:00:00.11 |
|   1 |  HASH UNIQUE        |           |      1 |      1 |      1 |00:00:00.11 |
|*  2 |   HASH JOIN ANTI SNA|           |      1 |      1 |     10 |00:00:00.11 |
|*  3 |    TABLE ACCESS FULL| CUSTOMERS |      1 |     13 |    532 |00:00:00.05 |
|*  4 |    TABLE ACCESS FULL| CUSTOMERS |      1 |     13 |     43 |00:00:00.04 |
--------------------------------------------------------------------------------
```

Listing 9: Null-Aware Anti-Join ab 11g

Kann man unsere Abfrage auch über einen Join formulieren? Das geht auch, siehe Listing 10. Wenn man einen Outer Join schreibt und anschließend nur Datensätze ohne Treffer berücksichtigt, bekommt

man sein Ergebnis. Allerdings leidet die Lesbarkeit und Verständlichkeit dieser Abfrage schon deutlich im Vergleich zu Unterabfrage-Varianten.

```
SQL> SELECT  DISTINCT c.cust_income_level
  2  FROM    sh.customers c LEFT JOIN sh.customers c2
  3              ON (c.cust_income_level = c2.cust_income_level
  4                  AND     c2.country_id = 52776
  5                  AND    c2.cust_city = 'Hamburg')
  6  WHERE   c.cust_city = 'Koeln'
  8  AND     c2.cust_id IS NULL;
K: 250,000 - 299,999

-------------------------------------------------------------------------------------
| Id  | Operation               | Name      | Starts | E-Rows | A-Rows |   A-Time
-------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT        |           |      1 |        |      1 |00:00:00.20
|   1 |  HASH UNIQUE            |           |      1 |      5 |      1 |00:00:00.20
|*  2 |   FILTER                |           |      1 |        |     10 |00:00:00.19
|*  3 |    HASH JOIN RIGHT OUTER|           |      1 |      5 |   2678 |00:00:00.18
|*  4 |     TABLE ACCESS FULL   | CUSTOMERS |      1 |     44 |     44 |00:00:00.09
|*  5 |     TABLE ACCESS FULL   | CUSTOMERS |      1 |    532 |    532 |00:00:00.06
-------------------------------------------------------------------------------------
```

Listing 10: Anti-Join “selbst gemacht”

Fazit

Die aufgeführten Beispiele zeigen, dass die Unterabfragen ihren Ruf der „Performance Killer“ nicht verdient haben. Sie müssen nicht langsamer als ein Join sein, können aber durchaus schneller sein. Man sollte sie nicht pauschal ablehnen. Kontinuierliche Verbesserungen der Datenbanksoftware führen dazu, dass der Cost Based Optimizer mehr Mittel zur Verfügung hat, um effiziente Ausführungspläne beim Einsatz von Unterabfragen zu generieren. Wir als Entwickler können uns auf den Fachanforderungen konzentrieren und lassen den CBO seine Arbeit machen.

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

Telefon: +49 (0) 211-58 66 64 70
Fax: +49 (0) 211-58 66 64 71
E-Mail andrej.pashchenko@trivadis.com
Internet: www.trivadis.com