WITH or WITHout you - Komfort-SQL in Oracle 12c

Dr.-Ing. Holger Friedrich
sumIT AG

Schlüsselworte
Subquery Factoring, WITH-Clause, Entwicklung, Performance, Optimiser

Einleitung
Oracle SQL enthält diverse Klauseln und Eigenschaften, die das Leben für Entwickler stark vereinfachen. Beispiele hierfür sind die WITH-Clause, Scalar Sub-Selects und neu in 12c Unterstützung bei der Verwendung mehrerer Zeitachsen. All diese Komfortfunktionem sind jedoch Segen und Fluch zugleich. Unsachgemäße Nutzung kann es dem Optimiser der Datenbank sehr schwer machen gute Ausführungsstrategien zu finden. Auch wird durch extensive Nutzung der Feature zuweilen stark fragmentierter Code entwickelt, der für Dritte schwer verständlich ist. Das Ergebnis ist allzu oft schlecht performender oder gar fehlerhafter Code.
Diesers Paper widmet sich der insbesondere dem SQL-Feature WITH-Clause, welches zum SQL-99-Standard gehört und in der Oracle längerem angeboten wird. Aus der Diskussion darüber, wie sie in der Datenbank abgearbeitet wird, werden ihre Möglichkeiten und Grenzen abgeleitet. Hinweise auf Änderungen der Verarbeitung im 12c-Release der Oracle Datenbank sind am Ende des Papers angefügt..

Optimiser und Query-Verarbeitung
Komfort-Sql-Funktion wie die WITH-Clause oder Scalar Sub-Selects erfüllen für Entwickler einen sehr sinnvollen und wertvollen Zweck. Sie erlauben es auf höherer Abstraktionsebene Code zu schreiben und für ihre Zwecke sinnvoll zu strukturieren. Den Code zu schreiben und ihn effizient ausführen zu können, sind aber zwei verschiedene Aspekte. Um zu verstehen, ob die Nutzung von Komfort-SQL-Funktionen zur Lösung eines Problems im Sinne der Abfrageperformance sinnvoll beiträgt, ist es wichtig sich zunächst die Arbeitsweise des DB-Optimisers anzuschauen.

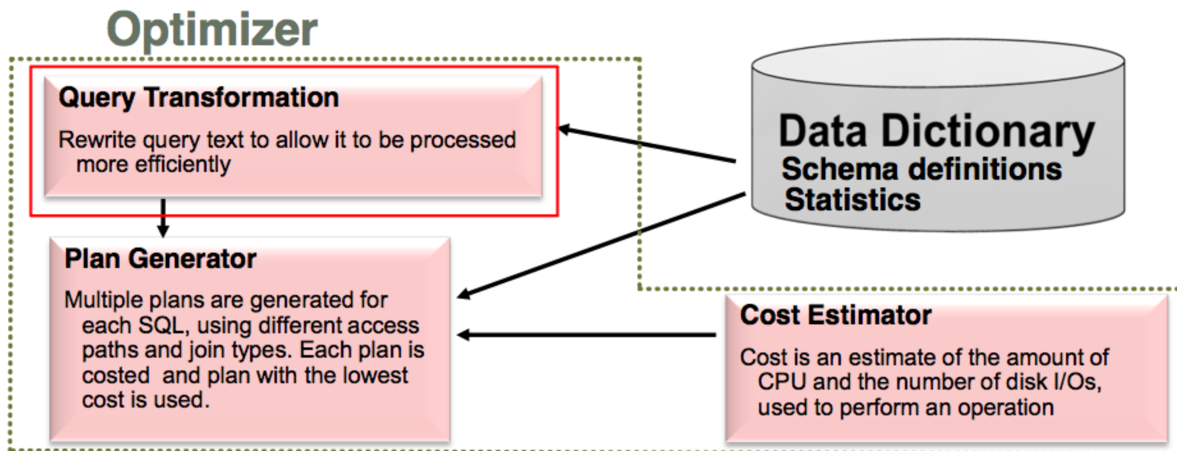

Abb. 1:Ddie drei Phasen der Bearbeitung einer Abfrage durch den Optimiser

Wie in Abbildung 1 dargestellt durchläuft die Bearbeitung einer Abfrage durch den Optimiser drei Phasen.

1. Query-Transformation:In diesem Schritt wird die gestellte Abfrage vom Optimiser umgeschrieben. Das Ziel ist es eine möglichst gute syntaktische Formulierung zu erreichen,

die den nachfolgenden Schritten die Bestimmung der effizientesten Ausführungsstrategie ermöglicht.
2. Plangenerierung: Der Optimiser nimmt den transformierten Abfragetext und erstellt eine ganze Reihe möglicher Ausführungspläne. Dies ist notwendig, da verschiedenste Operationen, wie Datenzugriffsarten, Join-Strategien etc. eine grosse Anzahl unterschiedlicher Pläne zulassen.
3. Kostenbestimmung und Planauswahl: Um sich für einen der erstellen Ausführungspläne entscheiden zu können, werden alle Einzeloperationen mit Kosten versehen und daraufhin die Gesamtkosten für jeden Plan bestimmt. Der günstigste Plan macht das Rennen und wird zur Ausführung an die Abarbeitungs-Engine übergeben.

Grundsätzlich verhält sich der Optimiser also wie ein Schachcomputer oder auch ein Mensch bei der Bestimmung des nächsten Zuges. Interpretiere die Situation auf dem Brett, evaluiere alle möglichen Züge, am besten mehrere Schritte in die Zukunft und entscheide dich für den Zug, der am wenigsten kostet, beim Schach die bestmögliche Folgesituation auf dem Brett mit dem Ziel den König des Gegners zu schlagen.

Viel wird in der Oracle-Community über den Schritt der Kostenermittlung gesprochen. Hier geht es um korrekte Statistiken, die Abschätzung von Kardinalität, Selektivität, Korrelation zwischen Datenfeldern etc. Oft wird dabei aber übersehen welches Potential für gute Performance und auch sehr bescheidene Performance in dem Schritt der Query-Transformation steckt. Hier versucht der Optimiser ja einen syntaktisch möglichst effizienten Plan zu generieren. Dazu werden eine Reihe von Techniken angewendet, zum Beispiel View Merging, Join Eliminatiuon, Join Factorization, Predicate Pushing oder die Transformation von Set-Operationen etc. zu Joins.

Die Anwendbarkeit oder eben gerade nicht vorhandene Anwendbarkeit dieser Transformationen ist im Hinblick auf die angesprochenen Komfort-SQL-Funktionen allerdings eingeschränkt. So lassen sich sowohl für die WITH-Clause, als auch Scalar-Sub-Selects einige Transformationen nicht oder nur eingeschränkt anwenden. Oder besser gesagt, es stehen für diese (noch) nicht in allen Konstellationen Transformationen zur Verfügung. Beispiele hierfür werden dies, sowohl für WITH-Clauses im Folgenden illustrieren.

Einführung WITH-Clause
Welche Möglichkeiten, aber auch potentiellen Fallstricke bringt die Nutzung der WITH-Clause nun, insbesondere im Hinblick auf die Query-Transformation? Die Basis-Syntax der WITH-Clause zeigt folgendes Listing 1.

```
WITH <alias_name> AS (subquery_sql_statement)
SELECT <column_name_list> FROM <alias>;
```
Listing. 1: WITH-Clause-Syntax

Die Clause gibt also die Möglichkeit Subqueries oder Inline-Views aus dem Hauptquery-Text herauszunehmen und abgesetzt, mit Alias-Namen versehen über dem Hauptquery zu plazieren. Im Gegensatz zu Views und Global Temporary Tables verbleibt dem Optimiser bei WITH-Clauses selbst die Entscheidung, ob er die Daten materialisiert oder das Statement im WITH-Teil in den Hauptquery hinein mergt. Ersteres Vorgehen entspricht grob dem Anlegen eines Temporary Table, während letzteres einem klassischen View-Merge entspricht.

Ein Beispiel für eine einfache Anwendung der WITH-Clause, anstatt eines Subqueries zeigen folgende beiden Listings.

```
SELECT e.ename AS employee_name,
       dc.dept_count AS emp_dept_count
FROM   emp e,
       (SELECT deptno, COUNT(*) AS dept_count
        FROM   emp
        GROUP BY deptno) dc
WHERE  e.deptno = dc.deptno;
```
Listing 2: Einfaches Beispiel vor Anwendung WITH-Clause

```
WITH dept_count AS (
  SELECT deptno, COUNT(*) AS dept_count
  FROM   emp
  GROUP BY deptno)
SELECT e.ename AS employee_name,
       dc.dept_count AS emp_dept_count
FROM   emp e,
       dept_count dc
WHERE  e.deptno = dc.deptno;
```
Listing 3: Einfaches Beispiel mit Anwendung WITH-Clause

Vorteile der WITH-Clause

Die Verwendung der WITH-Clause hat aus Entwicklersicht einige Vorteile.

- Ohne die Erstellung einer View oder der aufwendigeren Definition eines Global Temprary Tables entsteht so modularer, übersichtlicher Code.
- Mehrere With-Blöcke können in einem Statement verwendet werden, was die Modularität weiter erhöht.
- Ein With-Block kann beliebig oft im Haupt-Statement verwendet werden. Komplizierte Sub-Statements müssen so nur an einer Stelle der Abfrage gepflegt werdenn.
- With-Blöcke können in anderen With-Blocken, sozusagen als weitere ausfaktorierte Subqeries verwendet werden, was hilft komplexe Queries noch modularer zu schreiben.
- Indem dem Optimiser die Entscheidung überlassen wird, ob With-Clauses als Inline-Views gehandelt, oder ob sie bei der Abfrageausführung als Temporary table materialisiert werden, wird dem Entwickler diese Entscheidung abgenommen.

Viele Entwickler, insbesondere von komplexen Report-Queries auf relativ stark normalisierten Datenmodellen greifen daher die With-Clause dankbar auf und machen deren Anwendung de facto zur Entwicklungsbasis für ihr Berichtswesen. In der Folge werden Berichte nur noch in der im folgenden in Listing 4 skizzierten Art definiert.

```
WITH
alias1 AS (SELECT c1, c2, c3, ..., cm
                  FROM t1, t2, ... tl
                   ...),
alias2 AS (SELECT c1, c2, c3, ..., cm
                  FROM alias1, t1, ... tk
                   ...)
…
SELECT * from aliasN
WHERE …
```
Listing 3: Exzessive Anwendung der WITH-Clause

Listing 3 sieht zunächst nach gut wartbarem, einfachem Code aus. Und für den Moment ist er das auch. Aber wie steht es mit der Wiederverwandbarkeit? Und wie mit der Performance? Was macht der Query Transformator des Optimisers mit solchen WITH-Clauses?

Nachteile exzessiver Nutzung der WITH-Clause

Die Wiederverwendbarkeit wird dadurch gewährleistet, dass WITH-Blocks für andere Anfragen kopiert und gegebenenfalls leicht abgeändert werden. Abteilungen, die sich dieser Entwicklungsmethode verschreiben entwickeln praktisch eine Reihe von Building Blocks mittels der WITH-Clause, die sie dann für verschiedenste Abfragen nutzen und neu kombinieren. Dieses Vorgehen hat jedoch zwei gravierende Nachteile.

- Software Engineering: ändert sich die fachliche Logik für eine Abfrage, so dass eine Änderung der Implementierung notwendig wird, müssen alle erstellten Abfragen durchforstet werden, um herauszufinden an wievielen Stellen die WITH-Clause-Fragmente geändert werden müssen.
  Daher wären für diesen Zweck Database Views oftmals die bessere Variante. Sie erlauben es die fachliche Logik in einem Objekt zu kapseln und zentral zu pflegen. Ist die Logik komplex oder soll mehrfach in einer Abfrage verwendet werden, ist die Materialisierung mittels einer Materialized View, Hints oder die Verwendung von Temporary Tables zu erwägen.
- Performance: Building Blocks aus WITH-Block-Fragmenten beinhalten zumeist mehr Information, als für eine einzelne Abfrage benötigt wird. Das heisst, dass potentiell viele Storage-Zugriffe und Berechnung von Joins etc. durchgeführt werden, um Attribute zu berechnen, die im finalen Result Set der Gesamtabfrage überhaupt keine Verwendung finden und auch für Zwischenberechnungen, Filterung etc. nicht genutzt werden.

Aber sollte insbesondere die Performance-Problematik nicht automatisch vom Datenbank-Optimiser erledigt werden? Sollte er im Query-Transformation-Schritt nicht überflüssige Joins eliminieren und überflüssige Attribute durch Projektion in die WITH-Clauses hinein frühzeitig ausfiltern? In wie weit dies geschieht und das man hier schon früh auf grenzen stösst zeigt der folgende Absatz.

Query Transformation und die WITH-Clause

Der Query Transformator leistet im Zusammenhang mit komplexen Joins insbesondere einen wertvollen Beitrag durch seine Transformationen zur Join Elimination. Dies bedeutet, dass durch syntaktische Query-Analyse Joins bestimmt werden, die weder zur Ergebnisberechnung notwendig sind, noch deren Atribute in der Ergebnismenge Verwendung finden. Listing 4 zeigt den Effekt an einem einfachen Beispiel, in dem ein überflüssiger Join der EMP-Tabelle beseitigt wird.

```
-- simple join elimination
select e1.ename, d.dname
from emp e1, emp e2, dept d
where e1.empno = e2.empno
  and e2.deptno = d.deptno;

-----------------------------------------------------------------------------------------
| Id  | Operation                    | Name    | E-Rows |E-Bytes| Cost (%CPU)| E-Time   |
-----------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT             |         |     14 |   308 |     6  (17)| 00:00:01 |
|   1 |  MERGE JOIN                  |         |     14 |   308 |     6  (17)| 00:00:01 |
|   2 |   TABLE ACCESS BY INDEX ROWID| DEPT    |      4 |    52 |     2   (0)| 00:00:01 |
|   3 |    INDEX FULL SCAN           | PK_DEPT |      4 |       |     1   (0)| 00:00:01 |
|*  4 |   SORT JOIN                  |         |     14 |   126 |     4  (25)| 00:00:01 |
|   5 |    TABLE ACCESS FULL         | EMP     |     14 |   126 |     3   (0)| 00:00:01 |
-----------------------------------------------------------------------------------------
```

Listing 4: Join Elimination einfach

Join Elimination ist eine sehr wertvolle und mächtige Transformation, da sie auch komplexe Query-Konstruktion automatisch auf das notwendige Mindestmass reduzieren kann und hierdurch Ressourcenverschwendung verhindert. Diese würde sonst durch Berechnung überflüssiger Joins und Datenfelder oftmals massiv auftreten. Besonders vorteilhaft ist die Anwendung von Join Elimination im Zusammenhang mit der View-Merge-Transformation. Dort mergt der Optimiser zunächst eine Inline-View oder eine alleinstehende Datenbank View in den aufrufenden Query hinein und eliminiert daraufhin überflüssige Joins. Listing 5 illustriert Vorgehen und resultierenden Ausführungsplan.

```
-- vmerge simple w attr repl
select e1.ename, v.hiredate, v.dname
from emp e1, (select e2.empno, e2.hiredate, d.dname
              from emp e2, dept d
              where e2.deptno = d.deptno) v
where e1.empno = v.empno ;
```

```
-------------------------------------------------------------------------------------------
| Id  | Operation                    | Name    | E-Rows |E-Bytes| Cost (%CPU)| E-Time   |
-------------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT             |         |     14 |   476 |     6  (17)| 00:00:01 |
|   1 |  MERGE JOIN                  |         |     14 |   476 |     6  (17)| 00:00:01 |
|   2 |   TABLE ACCESS BY INDEX ROWID| DEPT    |      4 |    52 |     2   (0)| 00:00:01 |
|   3 |    INDEX FULL SCAN           | PK_DEPT |      4 |       |     1   (0)| 00:00:01 |
|*  4 |   SORT JOIN                  |         |     14 |   294 |     4  (25)| 00:00:01 |
|   5 |    TABLE ACCESS FULL         | EMP     |     14 |   294 |     3   (0)| 00:00:01 |
-------------------------------------------------------------------------------------------
```

Listing 5: Join Elimination mit View Merge

Doch was geschieht nun, wenn statt Inline-Views und Datenbank Views WITH-Clauses zur Subquery-Faktorisierung und Modularisierung von Codes eingesetzt werden? Funktioniert Join Elimination und View merging dann immer noch? Und wo sind die Grenzen? Die Antworten auf diese Fragen geben die folgenden Listings und Explain-Pläne.

```
-- with two small tables
with v as (select e2.empno, e2.hiredate, d.dname
              from emp e2, dept d
              where e2.deptno = d.deptno)
select e1.ename, v.dname
from emp e1,  v
where e1.empno = v.empno;
```

```
-------------------------------------------------------------------------------------------
| Id  | Operation                    | Name    | E-Rows |E-Bytes| Cost (%CPU)| E-Time   |
-------------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT             |         |     14 |   364 |     6  (17)| 00:00:01 |
|   1 |  MERGE JOIN                  |         |     14 |   364 |     6  (17)| 00:00:01 |
|   2 |   TABLE ACCESS BY INDEX ROWID| DEPT    |      4 |    52 |     2   (0)| 00:00:01 |
|   3 |    INDEX FULL SCAN           | PK_DEPT |      4 |       |     1   (0)| 00:00:01 |
|*  4 |   SORT JOIN                  |         |     14 |   182 |     4  (25)| 00:00:01 |
|   5 |    TABLE ACCESS FULL         | EMP     |     14 |   182 |     3   (0)| 00:00:01 |
-------------------------------------------------------------------------------------------
```

Listing 5: Join Elimination mit WITH-Clause

Listing 6 zeigt uns, dass bei einfacher Nutzung einer WITH-Clause der Optimiser diese in den meisten Fällen nicht materialisiert. Er behandelt sie wie eine View, vorausgesetzt es ist kein `MATERIALIZE`-Hint in der Select-Query des WITH-Blocks enthalten. In diesem Falle greift das View Merging und anschliessend die Join Elimination wie gewohnt. Wir erhalten den gleichen Explain Plan, wie bei Verwendung einer Datenbank- oder einer Inline-View (vergleiche Listing 5).

```
-- with two small tables, recursive reuse
with
emp1 (empno, ename, hiredate, deptno, comm) as
     (select e2.empno, e2.ename, e2.hiredate, e2.deptno, b.comm
      from emp e2, bonus b
      where e2.ename = b.ename (+)),
dep (empno, ename, dname) as
              (select e2.empno as empno, e2.ename as ename, d.dname as dname
                 from emp1 e2, dept d
                 where e2.deptno = d.deptno)
select emp1.ename, emp1.comm, dep.dname
from emp1, dep
where emp1.empno = dep.empno;
```

```
---------------------------------------------------------------------------------------------
| Id  | Operation                      | Name          | Rows  | Bytes | Cost (%CPU)| Time     |
---------------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT               |               |    14 |  1008 |    13  (16)| 00:00:01 |
|   1 |  TEMP TABLE TRANSFORMATION     |               |       |       |            |          |
|   2 |   LOAD AS SELECT               | SYS_TEMP_... |       |       |            |          |
|*  3 |    HASH JOIN OUTER             |               |    14 |   574 |     6  (17)| 00:00:01 |
|   4 |     TABLE ACCESS FULL          | EMP           |    14 |   294 |     3   (0)| 00:00:01 |
|   5 |     TABLE ACCESS FULL          | BONUS         |     1 |    20 |     2   (0)| 00:00:01 |
|*  6 |   HASH JOIN                    |               |    14 |  1008 |     8  (25)| 00:00:01 |
|   7 |    MERGE JOIN                  |               |    14 |   546 |     5  (20)| 00:00:01 |
|   8 |     TABLE ACCESS BY INDEX ROWID| DEPT          |     4 |    52 |     2   (0)| 00:00:01 |
|   9 |      INDEX FULL SCAN           | PK_DEPT       |     4 |       |     1   (0)| 00:00:01 |
|* 10 |     SORT JOIN                  |               |    14 |   364 |     3  (34)| 00:00:01 |
|  11 |      VIEW                      |               |    14 |   364 |     2   (0)| 00:00:01 |
|  12 |       TABLE ACCESS FULL        | SYS_TEMP_... |    14 |   476 |     2   (0)| 00:00:01 |
|  13 |    VIEW                        |               |    14 |   462 |     2   (0)| 00:00:01 |
|  14 |     TABLE ACCESS FULL          | SYS_TEMP_... |    14 |   476 |     2   (0)| 00:00:01 |
---------------------------------------------------------------------------------------------
```

Listing 7: Keine Join Elimination mit WITH-Clause

Werden jedoch mehrere komplexe WITH-Blöcke in einer Query genutzt oder wird ein WITH-Block mehrere Male in einer Query verwendet, entscheidet der Optimiser meistens, dass es besser ist die WITH-Blöcke zu materialisieren.

Das Ergebis dieser Entscheidung sehen wir in Listing 7, welches ein wirklich einfaches Beispiel zeigt. Der Entwickler hat beschlossen einen Building-Block für Angestellte (emp1) zu entwerfen und überall dort, wo Angestellte abgefragt werden zu nutzen. Damit aber ist jegliche Chance auf Eliminierung überflüssiger Joins und auch von frühzeitiger Prädikatreduktion dahin.

Dieser Verlust der Join Elimination ist ein grosses Problem, wenn Entwickler die oben beschriebene Strategie der Verwendung von WITH-Building-Blocks in ihren Applikationen oder Berichtsabfragen verwenden. Als Resultat wird in nicht wenigen Fällen mehr als die Hälfte der Rechenleistung ohne jeglichen Nutzen oder Notwendigkeit verschwendet.

WITH-Clause in Oracle 12c

In Oracle 12c ist für die WITH-Clause einige wichtige Erweiterungen eingebaut worden. Es ist nun möglich in einer WITH-Clause PL/SQL-Logik zu kapseln und diese im SELECT des hauptqueries zu nutzen. Listing 8 zeigt ein Beispiel aus der Oracle Dokumentation.

```
WITH
 FUNCTION get_domain(url VARCHAR2) RETURN VARCHAR2 IS
```

```
    pos BINARY_INTEGER;
    len BINARY_INTEGER;
  BEGIN
    pos := INSTR(url, 'www.');
    len := INSTR(SUBSTR(url, pos + 4), '.') - 1;
    RETURN SUBSTR(url, pos + 4, len);
  END;
SELECT DISTINCT get_domain(catalog_url)
  FROM product_information;
/
```

Listing 8: Definition und Verwewndung von PL/SQL-Funktionen und –Prozesuren in WITH-Clause

Fazit

Die WITH-Clause ist eine sehr mächtige und nützliche Funktion der Sprache SQL. Oracle hat eine standardkonforme Implementierung dazu vorgelegt, die vom Optimiser gut umgesetzt wird. Zudem ist die hinzugekommene Integration von PL/SQL in WITH-Blöcke eine hervorragende Erweiterung.
Aber, wie mit allen mächtigen Werkzeigen, ist auch bei der Nutzung der WITH-Clause Vorsicht geboten. Die Anzahl und Tiefe der Transformationen, die zur Zeit im Query Transformator des Optimisers implementiert sind ist gross, aber dennoch begrenzt. Die Verwendung mehrerer WITH-Blöcke, das rekursive verwenden und das (self-)joinen solcher Blöcke führt oftmals zu deren Materialisierung. Damit entfällt für diese Subqueries die Möglichkeit zur Join-Elimination und des Join-Predicate-Pushdown.
Exzessive Anwendung und aufblähen von WITH-Clause-Fragmenten als Building Blocks führt daher zu einer Verschwendung von Rechnerressourcen. Ein Overhead von mehr als 50% Arbeit zu viel, bei gleichzeitig schlechter Performance für Applikatiionen, Berichte und Analyse und damit für die Endanwender, sind dabei der Erfahrung nach leider keine Seltenheit.

Kontaktadresse:

Dr.-Ing. Holger Friedrich
sumIT AG
Täfernstrasse 28
CH-5405 Baden-Dättwill

Telefon:	+41 (0) 56 – 470 2500
Fax:	+41 (0) 79 – 320 8179
E-Mail	holger.friedrich@sumit.ch
Internet:	www.sumit.ch