VPD als Mittel zum Performancetuning im DWH

Mag. Dr. Thomas Petrik
Sphinx IT Consulting
Wien

Schlüsselworte

VPD, DWH, Virtual Private Database, Row Level Security, Performance, Bind Variable, Adaptive Cursor Sharing, Reporting

Einleitung

Die Virtual Private Database (VPD) wurde von Oracle mit der Version 8.1.5 als (kostenloser) Teil der Enterprise Edition eingeführt. Der Name selbst ist etwas irreführend, hat man es doch nicht mit einer eigenen Version der Datenbank zu tun, sondern vielmehr mit dem Thema Row Level Security: Basierend auf der Session Information (DB-User, OS-User, Client-Programm, Client-IP, etc.) werden (für den User vollkommen transparent) bestimmte Rows ausgeblendet. Die Abkürzung RLS (Row Level Security), die sich auch im Package-Namen wiederfindet (DBMS_RLS) beschreibt dieses Verhalten wohl besser. Weiters ist dieses Feature unter dem Namen Fine Grained Access Control (FGAC) bekannt und bildet die Basis der Advanced Label Security Option.

Mit der Version 10g wurde zusätzlich die Möglichkeit geschaffen, Daten auf Spaltenebene auszublenden (Column Security).

Dass die VPD sich allerdings auch hervorragend zum Performancetuning eignet, ist bis heute weitgehend unbekannt. Durch das implizite Einschleusen von Where-Klauseln unter bestimmten Bedingungen können bereits im Vorfeld bekannte Einschränkungen (beispielsweise auf einzelne Partitionen) erzwungen werden, was zu verlässlichen und sehr effizienten Ausführungsplänen führt. Diese Methode kann nicht nur in Batchprozessen sondern auch im Reportingbereich nutzbringend eingesetzt werden.

Da sich diese Technologie auch im DML-Kontext verwenden lässt, ist es mit den gleichen Steuerungsmethoden möglich, historische Daten vor ungewollten Veränderungen im Zuge der Batchverarbeitung (aufgrund von Unachtsamkeit oder Bugs) zu schützen – historische Rows werden auf diese Art zur Laufzeit gleichsam in einen Read-Only Modus gesetzt.

Grundlegende Funktionsweise der VPD

Eine VPD stellt eine Kombination aus Objekt (Tabelle, View oder Synonym) und einer PL/SQL-Funktion dar, die (mit weiteren Attributen versehen) durch eine Policy Function zusammengeführt werden. Das folgende Beispiel erläutert dies anhand einer einfachen Kundentabelle und einer Funktion, die die dynamische Where-Klausel zum Ausblenden von Rows auf Basis des Customer Types generiert:

```
CREATE TABLE customers
(
   cid     NUMBER (10)
  ,ctype   CHAR (3)
  ,cname   VARCHAR2(30)
);

CREATE FUNCTION vpd_customers (
   owner   IN VARCHAR2
  ,tabname IN VARCHAR2
) RETURN VARCHAR2
IS
BEGIN
   IF USER != 'REVISOR'
   THEN
      RETURN 'ctype = ''CUS''';
   ELSE
      RETURN NULL;
   END IF;
END;

BEGIN
   DBMS_RLS.add_policy (
       object_schema     => NULL
      ,object_name       => 'CUSTOMERS'
      ,policy_name       => 'PLC_CUSTOMERS'
      ,function_schema   => NULL
      ,policy_function   => 'VPD_CUSTOMERS'
      ,statement_types   => 'SELECT'
      ,policy_type       => DBMS_RLS.dynamic
      ,enable            => TRUE);
END;
```

Der Tabelle CUSTOMERS wird die Funktion VPD_CUSTOMERS für alle Select-Statements zugeordnet, was bewirkt, dass für alle User ausser dem "REVISOR" einem einfachen Select wie

```
select * from customers
```

implizit die Where-Klausel

```
where CTYPE = 'CUS'
```

angefügt wird. Zu sehen ist dies aus dem Execution Plan sowie aus der V$VPD_POLICY. Bei Ausführung des gleichen Statements erhalten also verschiedene User verschiedene Ergebnisse.

Bindvariablen als Performancefalle im DWH

Die Empfehlung für OLTP Systeme, bevorzugt Bindvariablen anstelle von Literalen zu verwenden, verliert im DWH-Umfeld ihre Gültigkeit, speziell dann, wenn mit Partitioning gearbeitet wird, was bei großen Datenmengen unumgänglich ist.

Die Verwendung von Bindvariablen für den Partitioning Key macht es dem Optimizer unmöglich, den optimalen Plan zu erstellen, da jede Partition einen anderen Füllgrad aufweist oder anderen Datenverteilungen unterliegt. Während für die eine Partition beispielsweise ein Indexzugriff besser

wäre, stellt sich für eine andere Partition ein Full Table Scan als optimal heraus. Aber auch bei nicht partitionierten Tabellen besteht das Problem in ähnlicher Weise.

Bei grossen Datenmengen führt der falsche Zugriffspfad leider nicht nur zu Performanceeinbußen, viele (vor allem komplexere) Statements können gar nicht mehr durchgeführt werden. Oft wird die eigentliche Problematik gar nicht erkannt und man versucht erfolglos, das Problem durch mehr Ressourcen zu lösen.

Mit Oracle 9i wurde im Optimizer erstmals das Bind-Peeking eingeführt. Dabei nimmt der Optimizer die Werte der Bindvariablen der ersten Execution zu Hilfe, um den richten Plan abzuschätzen. Dennoch ist dieser (aufgrund der zuvor ausgeführten Gegebenheiten) nicht notwendigerweise auch auf andere Partitionen anwendbar. Im Execution Plan ist das Bind Peeking mithilfe der Formatoption PEEKED_BINDS in DMBS_XPLAN.display_cursor zu sehen, ebenso wie im Optimizer Trace (event 10053, level 1).

Mit Oracle 11g kam das Adaptive Cursor Sharing hinzu. Dabei erkennt der Optimizer nach einigen Ausführungen, dass durch einen anderen Plan eine bessere Selektivität zu erreichen wäre und erstellt einen neuen Child Cursor. Vorraussetzung hierfür ist allerdings, dass Histogramme auf den relevanten Columns existieren und der Optimizer ein entsprechend starkes Data Skewing feststellt. Bind Sensitivity und Awareness eines Cursors sind in der V$SQL mithilfe der Spalten IS_BIND_AWARE (Child Cursor, CHILD_NUMBER > 0) und IS_BIND_SENSITIVE (Parent Cursor) dokumentiert.

Aufgrund der Tatsache, dass zumindest eine erfolgreiche Execution des Statements mit schlechter Performance notwendig wäre, um die Adaption des Plans zu ermöglichen, stellt diese Technik leider keine brauchbare Lösung des Problems dar. Darüberhinaus erkennt der Optimizer in vielen Fällen (speziell in Zusammenhang mit Partitioning) gar nicht die Notwendigkeit der Bind Sensitivity.

Herkömmlicher Lösungsansatz

Da nur die Verwendung von Literalen einen optimalen Plan ermöglicht, muss der Code entsprechend umgeschrieben werden. Abgesehen von dem damit verbundenen hohen Aufwand ist dies bei Fremdprodukten meist gar nicht möglich.

Einsatz der VPD

Gerade bei Batchprozessen sind der fachliche und vor allem der Datumskontext bekannt. Daher können auf Prozessebene die erforderlichen Where-Klauseln (mit Literalen) über eine VPD-Function, die allen beteiligten Basistabellen zugeordnet ist, implizit und ohne jede Code-Änderung eingeschleust werden. Es handelt sich dabei um eine Tuning-Maßnahme, die alleine vom DBA durchgeführt werden könnte.

Folgende Schritte stellen eine mögliche Vorgehensweise für ein DWH dar, das mit historisierten Tabellen arbeitet (jede Tabelle enthält ein Datumsfeld, die Batchverarbeitung sei datumsorientiert):

1. Alle Basistabellen werden mit der gleichen VPD versehen
2. Im Batchprozess wird eine eindeutige ACTION_ID vergeben
3. Die ACTION_ID wird in einer Steuertabelle CTL_ACTION mit dem gewünschten DATUM assoziiert (es kann sich natürlich auch um Datumsbereiche handeln und verschiedene Tabellen könnten unterschiedlichen Datumseinschränkungen unterliegen)
4. Die ACTION_ID wird in der Session z.B. mittels DBMS_APPLICATION_INFO.set_action hinterlegt
5. Die VPD-Function holt das DATUM für die Where-Klausel zur Runtime aus der Steuertabelle

Ein Grobablauf des Batchprozesses in PL/SQL könnte so aussehen:

```
begin
   -- ACTION in SYS_CONTEXT befüllen
   dbms_application_info.set_action('BATCH1');

   -- ACTION mit Datum verknüpfen
   insert into ctl_action (action_id, datum) values('BATCH1','20130304');
   commit;

   -- eigentliche Berechnung
   calculate(…);

   delete from ctl_action where action_id = 'BATCH1';
   commit;
end;
```

Beispiel für eine dazu passende VPD-Function:

```
CREATE FUNCTION vpd_batch (owner IN VARCHAR2, tabname IN VARCHAR2) RETURN
VARCHAR2
IS
   v_datum   VARCHAR2 (8);
BEGIN
   BEGIN
      SELECT datum
        INTO v_datum
        FROM ctl_action
       WHERE action_id = SYS_CONTEXT ('USERENV', 'ACTION');
   EXCEPTION
      WHEN NO_DATA_FOUND
      THEN
         RETURN NULL;
   END;
   RETURN 'datum = to_date(''' || v_datum || ''',''yyyymmdd'')';
END;
```

Jede beteiligte Tabelle wäre nun automatisch mit einer Where-Bedingung der Form

```
where DATUM = to_date(…)
```

versehen, ohne dass dadurch Abfragen in anderen Sessions beeinträchtigt wären.

Der häufig vorgebrachte Einwand, der Verzicht auf Bindvariablen würde die Datenbankperformance massiv beeinträchtigen, lässt sich leicht entkräften. Im Gegensatz zu OLTP Systemen hat man es im DWH bevorzugt mit vergleichsweise wenigen aber hinsichtlich der Laufzeit aufwendigeren Statements zu tun (klassisches Reporting, ETL). Daher führt weder die Parse-Phase noch die dadurch etwas erhöhte Cursor-Zahl im Library Cache zu Performanceengpässen.

Performancetuning im Reporting

Häufig besteht die Anforderung, je nach Beladungsfortschritt in einer View die Daten des aktuell geladenen Zeitraums (Tag, Monat, Quartal, etc.) anzuzeigen, ohne dass der Benutzer sich um die Datumseinschränkung kümmern muss. Üblicherweise wird zu diesem Zweck der gewünschte Zeitraum wiederum in einer Steuertabelle hinterlegt, um diese dann mit der Tabelle in einer View zu joinen, wie folgendes Beispiel zeigt:

```
CREATE VIEW umsatz_v
AS
   SELECT umsatz.*
     FROM umsatz, ctl_datum
    WHERE umsatz.datum = ctl_datum.datum
```

Die View UMSATZ_V zeigt also immer nur jene Umsätze an, die für den Tag, der in der Tabelle CTL_DATUM hinterlegt ist, angefallen sind.

Für den Optimizer ist dies wiederum eine Query mit Bindvariablen und führt zu den bereits geschilderten Schwierigkeiten. Auch in diesem Fall schafft eine VPD Abhilfe auf sehr elegante Art: Anstelle der View wird ein Synonym auf die Tabelle gesetzt und das Synonym selbst mit einer VPD versehen, die das jeweils hinterlegte Datum einsteuert. Somit bleibt gleichzeitig die historisierte Tabelle in vollem Umfang abfragbar. Alle Abfragen auf das neue Synonym werden vollkommen transparent durch die VPD unter Verwendung von Literalen in der Where-Bedingung eingeschränkt.

Schutz vor unerwünschtem DML

Transformationen im Rahmen des ETL-Prozesses laufen üblicherweise in einem bestimmten (Datums-)Kontext ab. Unkontrollierte Änderungen von Altdaten aufgrund von Software-Bugs oder Unachtsamkeiten beim Design des ETL-Prozesses stellen ein erhebliches Betriebsrisiko dar. Der Schutz einzelner Partitionen oder gar einzelner Rows (auch in unpartitionierten Tabellen) ist auch in diesem Fall leicht realisierbar durch Einsatz einer VPD.

Zu diesem Zweck reicht es aus, in der bereits beschriebenen Weise eine eigene (zusätzliche) VPD-Policy auf die betroffenen Tabellen aufzuschalten, die nun auch die Kontrolle über Inserts, Updates und Deletes übernimmt. Zwei Parameter sind dafür in DBMS_RLS.add_policy maßgeblich:

```
statement_types  => 'SELECT,INSERT,UPDATE,DELETE'
```

und

```
update_check => true
```

Unerwünschte Inserts resultieren in einem ORA-28115, während fehlgeleitete Updates und Deletes ohne weitere Warnung einfach nicht durchgeführt werden.

Zusammenfassung

Die Virtual Private Database bietet als kostenloses Feature der Enterprise Edition eine hocheffiziente und vergleichsweise einfache Möglichkeit, Performanceprobleme, die durch die Verwendung von Bindvariablen in Verbindung mit grossen Datenmengen verursacht werden, in den Griff zu bekommen, ohne den Sourcecode von ETL-Prozessen oder Reporting-Lösungen ändern zu müssen.

Aus derzeitiger Sicht und nach ersten Tests mit Oracle 12c behalten die dargestellten Aussagen und Konzepte auch in dieser neuen Datenbankversion ihre volle Gültigkeit.

Kontaktadresse:

Mag. Dr. Thomas Petrik
Sphinx IT Consulting
Aspernbrückengasse 2
A-1020 Wien

Telefon:	+43 664 155 8304
Fax:	+43 (1) 599 31-99
E-Mail	Thomas.Petrik@sphinx.at
Internet:	www.sphinx.at