Die View von der View von der View

PERFORMANTES SQL SCHREIBEN

Uwe Embshoff
Airpas Aviation AG
Braunschweig

Schlüsselworte
SQL, Performance, Optimizer

Einleitung

Es gibt viel Literatur zum Thema Oracle Performance Tuning. Aber wenig zum Thema, wie man performante Anwendungen entwickelt. Dazu gehört auch die Frage, wie man performantes SQL schreibt. Mit dieser Frage habe ich mich hier beschäftigt. Die im Folgenden aufgeführten Punkte sind teilweise in Oracle Dokumenten (vor allem im Oracle 11gR2 Performance Tuning Guide) nachzulesen, teilweise basieren sie auch auf praktischer Erfahrung bei der Entwicklung von airpas.

airpas

airpas ist ein Produkt, das von Fluggesellschaften u.a. zur automatischen Rechnungsprüfung und der Berechnung der Profitabilität eingesetzt wird. Dazu werden jede Nacht automatisch sehr große Datenmengen importiert. Auf Basis dieser aktuellen Flugdaten und der gespeicherten Verträge (z.B. mit Flughäfen, Kraftstofflieferanten) werden die erwarteten Rechnungen simuliert und dann automatisch mit den tatsächlich eingegangenen Rechnungen verglichen. Das geschieht mit PL/SQL und Java in einer Oracle Datenbank. Aufgrund der großen Datenmengen spielt Performance bei uns eine sehr große Rolle – nicht nur beim Betrieb von airpas, sondern auch bereits in der Entwicklung.

1. Keine impliziten Typkonvertierungen verwenden

Das gibt es in fast jeder Anwendung: Spalten, die eine Nummer enthalten, die fast immer eine Zahl ist. Bei airpas ist das z.B. bei den Flugnummern so. Das sind i.d.R. 4-stellige Zahlen, eine Flugnummer kann aber in Ausnahmefällen auch „4711A“ sein. Die Spalte definiert man dann natürlich als VAR-CHAR2.

Wenn ich nun schnell mal nach einer Flugnummer suchen will, dann gebe ich ein:

```
where flightnr = 4711
```

Der Optimizer wandelt dies um in

```
where to_number(flightnr) = 4711
```

Die Folgen: Wenn ich einen Index auf die `flightnr` habe, dann wird dieser nicht genutzt. Und noch schlimmer: Wenn es in der Tabelle eine Flugnummer „4711A“ gibt, dann bekomme ich eine Fehlermeldung

```
ORA-01722: Invalid number
```

da Oracle „4711A“ nicht in eine Zahl umwandeln kann.

Man fragt sich schon, warum Oracle die Typumwandlung nicht andersherum macht. Die Begründung ist: Ich könnte ja auch eine Flugnummer „04711“ haben. Die würde dann mit der o.g. Typumwandlung gefunden werden.

Das gleiche Problem gibt es natürlich auch beim Vergleich mit numerischen Spalten oder anderen numerischen Ausdrücken:

```
where flightnr = flightnr_numeric
```

Da kann das Verhalten übrigens davon abhängen, welche Spalte links vom Gleichheitszeichen steht und welche rechts.

D.h. letztendlich: Ob ich eine Fehlermeldung bekomme, hängt nicht nur von meinem Code ab, sondern auch von Daten, nach denen ich gar nicht suche und von einem Ausführungsplan, für den sich der Optimizer ohne mein Zutun entscheidet. Kurz gesagt: Im Test kann alles gut gehen, auf Produktion „knallt“ es dann.

Die Lösung ist einfach: Typumwandlungen immer selbst durchführen:

```
where flightnr = '4711'
```

2. Spaltentransformationen vermeiden

Ausdrücke wie

```
to_char(flightdate, 'dd.mm.yyyy') = '01.11.2015'
flightdate + 7 = trunc(sysdate)
nvl(flightnr, '0000') = '4711'
```

hindern den Optimizer daran, einen evtl. vorhandenen Index zu benutzen oder – wenn die Tabelle partitioniert ist – Partition Elimination einzusetzen. Besser wäre:

```
flightdate = to_date('01.11.2015', 'dd.mm.yyyy')
flightdate = trunc(sysdate) - 7
flightnr = '4711'
```

Bei einer Join- Bedinung wie

```
to_char(flights.flightdate, 'ddmmyy') = imp_flights.flightdate_char
```

kann man den Index auf `flightdate` nutzbar machen, indem man die Funktion auf die rechte Seite verlagert:

```
flights.flightdate = to_date(imp_flights.flightdate_char, 'ddmmyy')
```

Aber egal ob ich einen Index habe oder nicht: Spaltentransformationen führen auch noch aus einem anderen Grund zu ungünstigen Ausführungsplänen.

Durch die Objektstatistiken weiß der Optimizer ja nicht nur, wie viele Zeilen eine Tabelle hat, sondern auch wie viele unterschiedliche Werte eine Spalte hat.

Schränke ich nun meine Abfrage ein, so kann der Optimizer i.d.R. einschätzen, wie viele Zeilen dann noch bleiben. Die Frage, ob das dann 100 oder 100.000 Datensätze sind ist für den Optimizer essentiell, um beim Ausführungsplan nicht total daneben zu liegen.

Spaltentransformationen hindern den Optimizer daran, die Anzahl der selektierten Zeilen korrekt zu schätzen. Das kann z.B. dazu führen, dass eine vollkommen ineffiziente Joinmethode angewendet wird. Wenn es nicht anders geht und man tatsächlich Performanceprobleme hat, so kann man „extended statistics“ mit der betreffenden Funktion anlegen oder auch einen „function based index“.

3. Outer Joins vermeiden

Bei einem Join zwischen zwei Tabellen prüft der Optimizer zunächst, welche der beiden Tabellen die größte Einschränkung verspricht. Mit dieser Tabelle beginnt er („driving table“).

Diese Technik kann der Optimizer aber nur bei einem Inner Join anwenden. Bei einem Outer Join sieht es anders aus:

```
select *
from emp, dept
where emp.deptno = dept.deptno (+)
```

Der Optimizer muss hier immer mit der Tabelle `emp` beginnen. Wenn er mit `dept` beginnen würde, würde er auf die Mitarbeiter aus `emp`, die keiner Abteilung zugeordnet sind, gar nicht kommen.

D.h. durch den Outer Join schränken wir die Möglichkeiten des Optimizers ein.

Oft lässt sich so ein Outer Join ja gar nicht vermeiden. Dann ist das natürlich in Ordnung. Aber manchmal ist es auch so, dass man „sicherheitshalber“ lieber einen Outer Join macht – weil man eben nicht genau weiß, ob auch jeder Mitarbeiter einer Abteilung zugeordnet ist.

Woher könnte man es wissen? Z.B. wenn es einen Foreign Key von `emp.deptno` auf `dept.deptno` gibt und außerdem eine `not null` Constraint auf `emp.deptno`.

4. Views mit Gettern in einen Cursor verlagern

Getter Funktionen in Views, z.B.

```
create or replace view v_emp as
select ...
from emp, ...
where country = pck_emp.getCountry
```

erlauben kein Bind Peeking. D.h. Der Optimizer weiß nicht, ob die Country, die er bekommt auf 1% oder vielleicht auf 90% aller Datensätze zutrifft. Kritische Auswirkungen hat das nach unserer Erfahrung

insbesondere bei Spalten, nach denen partitioniert wurde: Wenn der Optimizer den Wert nicht kennt, kann er keine Partition Elimination durchführen.

Abhilfe kann man schaffen, indem man die View in einem Cursor in einem Package vorhält. In diesem Fall ist Bind Peeking möglich. Braucht man die View auch als View, so kann man den Cursor mit Hilfe einer „pipelined function“ als View darstellen.

5. Zwischenergebnisse speichern

Tabellen, um Zwischenergebnisse zu speichern benötigen zwar Ressourcen, sind aber manchmal doch sinnvoll:

- Wenn die Zwischenergebnisse mehrfach genutzt werden können.
- Wenn dadurch komplexe SQL Statements in mehrere kleinere Statements aufgeteilt werden. Diese sind oft nicht nur leichter verständlich sondern auch einfacher zu optimieren.

6. Materialized Views verwenden

Materialized Views sind eine Alternative wenn sich eine „normale“ View nicht tunen lässt oder auch wenn man einfach nur die Daten speichern möchte, die die View liefert.

7. Für unterschiedliche Zwecke spezielle SQL Statements schreiben

SQL ist keine prozedurale Sprache. Ein SQL Statement sollte immer nur eine Sache tun. Es sollte z.B. nicht unterschiedliche Dinge zusammenfassen, die dann – abhängig von übergebenen Parametern – zurückgegeben werden. Man sollte lieber ein einfaches, separates Statement schreiben anstatt ein komplexes Statement um weitere Funktionalitäten zu erweitern.

8. Das Joinen von komplexen Views vermeiden

Bild 1: Die View von der View von der View

Viele Entwickler oder auch DBAs werden das schon einmal erlebt haben: Man öffnet eine View, stellt fest dass diese auf weiteren Views basiert und wenn man dann eine von denen öffnet, dann scheint es immer so weiter zu gehen. Kann so was gut sein?

Die Antwort ist: Es kommt darauf an.

Zunächst einmal muss man sagen, dass es im Prinzip egal ist, ob man ein Statement in eine separate View schreibt oder ob man den SQL-Text der View direkt verwendet. Beides ist für den Optimizer das Gleiche – zumindest theoretisch. In der Praxis haben wir festgestellt, dass es Statements gibt, die schneller sind wenn man sie als „Monsterabfrage“, also ohne untergeordnete Views, schreibt. Der umgekehrte Fall könnte aber evtl. genauso vorkommen, ist uns nur noch nicht untergekommen.

Der Optimizer optmiert viel Überflüssiges wieder weg, indem er sogenannte „Query Transformations“ durchführt. Hier nur einige Beispiele:

- Views und Subselects werden – so weit wie möglich – „gemerged“, d.h. aufgelöst.
- Wende ich eine WHERE-Bedingung auf eine View an, so versucht der Optimizer diese Bedingung möglichst früh schon innerhalb der View anzuwenden („predicate push down“).
- Selektiere ich nur einige Spalten aus der View, dann wird der Optimizer die anderen Spalten gar nicht erst ermitteln („select list pruning“).

Je komplexer die Views sind, desto schwieriger wird es natürlich für den Optimizer. Richtig schwierig wird es, wenn komplexe Views gejoint werden – insbesondere auch wenn diese Gruppenfunktionen oder analytische Funktionen enthalten. Der Optimizer muss sich dann für eine der beiden Views entscheiden, die er komplett ausführt, um dann anschließend mit den Daten aus der View weitermachen zu können.

Zusammenfassend kann man folgende Faustregeln aufstellen:

- Die View von der View von der View ist nicht so katastrophal wie man denkt, aber das Joinen von komplexen Views sollte man vermeiden.
- Wenn man die Tabellen, die in einer View abgefragt werden gar nicht alle benötigt, dann sollte man lieber eine neue View schreiben anstatt eine bestehende zu „recyceln“.

Bild 2: Das Joinen von komplexen Views vermeiden

9. Dem Optimizer alles sagen

Der Optimizer kann nur so gut sein, wie die Informationen die er hat. Geben Sie ihm so viele Informationen wie möglich. Foreign Keys und Unique Keys sind für den Optimizer wertvolle Informationen. Statistiken über Spaltengruppen können dem Optimizer helfen, die Menge der selektierten Datensätze („Cardinality“) besser abzuschätzen.

10. Alles benutzen

Oracle bietet zahlreiche spezielle Funktionalitäten wie beispielsweise Aggregatfunktionen. Diese eingebauten Funktionalitäten sind schneller als wenn man sie per Hand selbst schreibt, daher sollte man sie nutzen.

Kontaktadresse:
Uwe Embshoff
Airpas Aviation AG
Theodor-Heuss-Str. 2
38122 Braunschweig

Telefon: +49 (0) 531-8852 9100
E-Mail uwe.embshoff@airpas.com
Internet: www.airpas.com