Performance-Stabilisierung mit Einsatz von SQL Plan Baselines

Katja Werner
OPITZ CONSULTING GmbH
Standort München

Schlüsselworte

SQL Plan Baseline, Performance, Stabilisierung, SQL Tuning Set, Test

Einleitung

"Die Performance ist schlecht", meldet der Anwender. "Die Tests waren gut", sagt der Entwickler. Der Datenbankadministrator hat die Ursache schnell gefunden - die Ausführungspläne schwanken stark. Aber: welcher Ausführungsplan ist der Richtige? Und: wie sah das ganze in der Testumgebung aus? So kommt eins zum anderen. Im Vortrag wird konkret eingegangen auf:
- Identifizieren von Problemstatements
- Einsatz von SQL Plan Baselines zur Stabilisierung wechselnder Ausführungspläne
- Tests und Aufbewahrung von Testergebnissen

Ausgangslage

Im Rahmen einer Wartungstätigkeit über mehrere Monate wurden immer wieder Performanceprobleme an den DBA gemeldet. Bei Tests der Applikationssoftware vor der Auslieferung war nie eine schlechte Performance aufgefallen. Allerdings wurden weder Testfälle noch –ergebnisse aufbewahrt, so dass der Wartungs-DBA keinen Vergleich mit dem aktuellen Zustand hatte.

Etwas fiel auf: Die Performance war nicht durchweg schlecht. Oft war sie sogar sehr gut. Es gab allerdings starke Schwankungen bei den Ausführungsplänen. Immer wieder wurden für bestimmte Statements Ausführungspläne vom Optimizer entwickelt, die Nestes Loop Joins über große Datenmengen enthielten, obwohl Hash Joins letztlich die deutlich schnellere Variante waren.

Zwei Fragen stellten sich:
1.) Warum wählt der Optimizer so viele unterschiedliche und teils auch schlechte Ausführungspläne?
2.) Was kann getan werden, damit die „richtigen“ Ausführungspläne genommen werden und die Performance so stabilisiert werden kann?

Die Umgebung war wie folgt: Oracle Enterprise Edition 11.2.0.3, Diagnostic + Tuning Pack

Funktionsweise des Cost Based Optimizers

Um zu verstehen, warum die Ausführungspläne ungünstig sein können, muss man sich vor Augen führen, was der Cost Based Optimizer eigentlich macht.

Das Statement wird vom Optimizer in einzelne Komponenten zerlegt. Es wird geprüft, auf welche Tabellen und Spalten zugegriffen wird. An Hand von Statistiken versucht der Optimizer zu ermitteln, wie viele Zeilen wohl für die Abfrage zurückgegeben werden und wie hoch die Selektivität ist. Weiterhin wird geprüft, ob der Zugriff direkt auf die Tabellenblöcke erfolgt oder ob besser ein Index genutzt wird. Sind zudem Bindevariablen vorhanden, prüft der Optimizer auch die enthaltenen Werte. All diese Dinge und noch viel mehr berücksichtigt der Optimizer.

Aufbauend auf dieser Schätzung prüft der Optimizer verschiedene Joinvarianten. Wird eine Tabelle mit wenig zurückgegebenen Zeilen mit einer Tabelle mit vielen zurückgegebene Zeilen gejoint, tendiert der Cost Based Optimizer zu einem Nested Loop Join, enthalten beide Tabellen viele zurückgegebene Zeilen wird ein Hash Join bevorzugt.

Im nächsten Schritt rechnet der Optimizer für viele verschiedene Kombinationen die Kosten aus. Daraus wird dann der Ausführungsplan mit den geringsten Kosten gewählt.

Dem Optimizer sind Grenzen gesetzt: Sind Funktionen oder Prozeduren im Statement enthalten, kann der Optimizer nicht den genauen Wert bestimmen und nimmt irgendeinen Schätzwert. Werden zu viele Tabellen gejoint werden unter Umständen nicht alle möglichen Kombinationen geprüft, so dass eine bessere Variante von vornherein rausfallen kann.

Auf jeden Fall steigt mit der Komplexität des Statements auch die Vielfältigkeit der Ausführungspläne. Abweichungen bei der Schätzung von Kardinalität und Selektivität für zurückgegebene Datenmengen von der Realität potenzieren sich mit der Anzahl der zu joinenden Tabellen. Und damit auch die Wahrscheinlichkeit, dass der Optimizer mal bessere und mal gänzlich ungeeignete Pläne ermittelt.

Folgende Maßnahmen sind zur Stabilisierung der Performance geeignet: SQL Plan Baselines, SQL-Profile, Hints oder Outlines. Sie geben dem Optimizer – mal mehr, mal weniger – Richtlinien, wie ein bestimmtes Statement auszuführen ist. Im betrachteten Projekt erwiesen sich SQL Plan Baselines als die herausragendste Möglichkeit, die Performance zu stabilisieren.

SQL Plan Baselines – die Theorie

Mit SQL Plan Baselines bekommt der Optimizer Hinweise, wie ein bestimmtes Statement abzuarbeiten ist. Es kann pro Statement mehrere Baselines geben. Sie sind in der SQL Plan Management Base (SMB) aufbewahrt.

Der Vorteil gegenüber Hints, Profiles und Outlines ist, dass der Optimizer trotzdem beim Parsing des Statements einen Ausführungsplan entwickelt. Erst nach der Parsing-Phase sieht er nach, ob es zu diesem Statement SQL Plan Baselines vorhanden sind. Der Optimizer bewertet die vorhandenen Baselines und wählt die aus seiner Sicht kostengünstigste Variante. Ist nur eine Baseline für das Statement vorhanden, wird diese gewählt. Der beim anfänglichen Parsen ermittelte Ausführungsplan wird der SQL Plan Management Base (SMB) hinzugefügt. Lässt sich eine Baseline nicht nachstellen, weil zum Beispiel ein für die Baseline verwendeter Index gelöscht wurde, so entwickelt der Optimizer einen Alternativplan.

SQL Plan Baselines sind letztlich ein Set von Hints für einen SQL-Handle. Der SQL-Handle ist eine Codierung für den Text des SQL-Statements. Achtung: Die SQL_ID ist ebenfalls konform zum Text

eines SQL-Statements, ist aber doch nicht ganz dasselbe wie der SQL-Handle. Deshalb sollte in Verbindung mit SQL Plan Baselines auch möglichst nur vom SQL-Handle gesprochen werden.

SQL Plan Baselines können auf verschiedene Weise erstellt werden:

- Automatische oder manuelle Sammlung für alles Statements aus dem Cache
- Erstellung aus SQL Tuning Sets (und hierüber indirekt aus dem AWR)

Damit sie für ein Statement „ziehen“, müssen folgende Voraussetzungen gegeben sein:

- Init.ora-Parameter optimizer_use_sql_plan_baselines=TRUE;
- Ausführungsplan der Baseline muss den Status „enabled“ haben und „accepted“ sein
- Das Statement muss hard geparst werden, das heisst, es darf noch nicht im Cache liegen

Praktische Umsetzung

Für eine Historie von SQL-Statements wurden auf der Produktionsdatenbank aus der DBA_HIST_SQL-Tabelle alle Informationen zu Applikationsstatements in regelmäßigen Abständen in eine separat erstellte Tabelle abgezogen. So lagen Daten zu Ausführungszeiten und –plänen für die letzten Monate vor.

An Hand dieser Tabelle konnte ermittelt werden, welche und wieviel Statements schwankende Ausführungspläne aufweisen. Im vorliegenden Fall war die Zahl nicht sehr hoch: ca. 20 Statements waren von der Problematik betroffen. Diese hatten es allerdings in sich – bis zu 20 unterschiedliche Ausführungspläne pro Statement wurden im Lauf von ca. 12 Wochen protokolliert.

Die Statementsammlung wurde auch genutzt, um Laufzeiten einzelner Statements sowie der Applikationsprozesse zu ermitteln. Damit wurde erstmals „bewiesen“, welche Applikationsprozesse und Statements die Top-Ressourcenverbraucher über lange Zeit sind und wo das Tuningpotenzial am höchsten ist.

Zur Priorisierung der zu tunenden und zu stabilisierenden Statements wurden nun die oben ermittelte Zahl unterschiedlicher Ausführungspläne pro Statement sowie die SQL-Laufzeiten verwendet.

Für die Statements mit wechselnden Ausführungsplänen wurde jeweils der optimale Ausführungsplan auf einer Testumgebung ermittelt. Dafür wurden die Statements jeweils mit kleinen und großen Datenmengen ausgeführt und auf Laufzeiten getestet, um auch für die jeweilige Datenmenge den passenden Ausführungsplan vorzuhalten. Insbesondere für Statements, die Bindevariablen oder Funktionen verwenden, war dieser Schritt wichtig. An diesem Punkt der Tests war die Zusammenarbeit mit Entwicklern bzw. Applikationsbetreuern unabdingbar. Als Resultat konnten neben den optimalen Ausführungsplänen auch gleich die Testfälle protokolliert und in Form von SQL Tuning Sets aufbewahrt werden. Sie können zukünftig bei Upgrades oder Hardwarewechseln leicht wieder verwendet und verglichen werden.

Im Anschluss an die Tests wurden die entwickelten SQL Plan Baselines in eine Referenzumgebung und von dort überall hin – auch auf die Produktionsdatenbanken übertragen. Dort wurde zu festen Zeitpunkten die Tabelle dba_sql_plan_baselines abgegriffen. Hier sind auch die Ausführungspläne erfasst, die der Optimizer an Stelle der SQL Plan Management Base enthaltenen SQL Statements gewählt hätte. Diese „Alternativpläne“ werden auch dann erzeugt, wenn der init.ora-Parameter optimizer_capture_sql_plan_baselines auf FALSE steht. Sie konnten in der Testumgebung an Hand

der vorhandenen Testfälle auf bessere Performance getestet und – bei Bedarf – in Produktion umgesetzt werden.

Fazit

Das beste Tuning nutzt nichts, wenn der Optimizer immer mal wieder andere, in der Realität langsamere Ausführungspläne wählt. Die Stabilisierung der Ausführungspläne ist ebenfalls wichtig. Im vorliegenden Beispiel waren SQL Plan Baselines das Mittel der Wahl.

Eine Historie der Tabelle dba_hist_sqlstat gibt über lange Zeiträume einen guten Einblick in lang laufende SQL Statements und Prozesse sowie Anzahl von Ausführungsplänen pro Statement. Diese sollten vordergründig im Rahmen von Tuningmaßnahmen behandelt werden.

Die Testfallszenarien sollten unbedingt protokolliert sowie in Form von SQL Tuning Sets aufbewahrt werden. So können sie jederzeit für Performancetests bei Release- oder Hardwarewechseln sowie Datebankupgrades wiederverwendet werden.

Kontaktadresse:
Katja Werner
OPITZ CONSULTING Deutschland GmbH
Standort München
Weltenburger Straße, 4
D-81677 München

Telefon: +49 (0) 89-680 098 0
Fax: +49 (0) 89-680 098 4400
E-Mail katja.werner@opitz-consulting.com
Internet: www.opitz-consulting.com