Performanceprognosen im Test, trotz Datenschutzauflagen

Daniel Stein
Debeka
Koblenz, Deutschland

Schlüsselworte
Datenbanken, Test, Entwicklung, Performance, Optimizer

Einleitung
Wenn Unternehmen Software entwickeln, werden auch Testdaten benötigt, idealerweise die kompletten Originaldaten. Allerdings bedeutet das, dass bei Softwaretests durch die zu testende Software Tätigkeiten im Sinne §3 Abs. 3-5 BDSG durchgeführt werden. Originaldaten dürfen daher nur in Teilmengen und anonymisiert zum Testen von Software eingesetzt werden. Leider bringt diese Regelung in der Softwareentwicklung - gerade im Bereich Datenbanken - auch Nachteile mit sich. Wenn in einer Datenbank-Test-Umgebung SQL-Statements entwickelt und getestet werden, trifft der Oracle Cost-Based-Optimizer seine Entscheidungen auf Grundlage der Statistiken, die zu den Testdaten ermittelt wurden. Diese sind dann aber nicht repräsentativ für die Produktions-Umgebung.

In der Praxis hat dies dazu geführt, dass ineffiziente SQL-Statements in Produktion gelangt sind und die Datenbank unnötig belastet haben. Bei einer anstehenden Optimizer-Migration (FIRST_ROWS nach ALL_ROWS) ist eine Technik entstanden, die es über die Migration hinaus erlaubt, in Test-Umgebungen valide Aussagen zur späteren Performance von Statements in Produktion zu treffen. Im Vortrag werden darüber hinaus noch beispielhaft Erfahrungen aus der Praxis mit dieser Technik besprochen.

Wie aus einer Optimizer-Migration ein Werkzeug für Performance Tests entsteht
Bis März 2016 wurde in den Datenbanken der Debeka der Optimizer -Mode FIRST_ROWS eingesetzt. Dies führte wiederholt zu Problemen. Ein Beispiel:

```
SELECT *
FROM ZV01_AUSGZAHLUNG
JOIN ZV01_OB ON  ZV01_OB.OBID = ZV01_AUSGZAHLUNG.OBID
WHERE BETRAG = 47.11
ORDER BY AUSGZAHLID ;
```

Die Felder **BETRAG** und **AUSZAHLUNGSID** sind indiziert. Dieses simple Statement führt unter FIRST_ROWS zu folgendem Plan:

```
-------------------------------------------------------------------------------------
| Id  | Operation                     | Name                 | Rows  | Cost (%CPU)|
-------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT              |                      |   107 |  2297K  (1)|
|   1 |  NESTED LOOPS                 |                      |   107 |  2297K  (1)|
|   2 |   NESTED LOOPS                |                      |   107 |  2297K  (1)|
|*  3 |    TABLE ACCESS BY INDEX ROWID| ZV01_AUSGZAHLUNG     |   107 |  2296K  (1)|
|   4 |     INDEX FULL SCAN           | PK_ZV01_AUSGZAHLUNG  |    56M|   111K  (1)|
|*  5 |    INDEX UNIQUE SCAN          | PK_ZV01_OB           |     1 |     1   (0)|
|   6 |   TABLE ACCESS BY INDEX ROWID | ZV01_OB              |     1 |     2   (0)|
-------------------------------------------------------------------------------------

Predicate Information (identified by operation id):
---------------------------------------------------

   3 - filter("ZV01_AUSGZAHLUNG"."BETRAG"=47.11)
   5 - access("ZV01_OB"."OBID"="ZV01_AUSGZAHLUNG"."OBID")
```

Wie man sieht, liest Oracle als erstes (Id 4) 56 Mio Zeilen (Spalte ROWS) aus der Tabelle per „INDEX FULL SCAN“, also sortiert. Bei diesem Tabellenzugriff bleiben dann aber nur 107 Zeilen übrig (Id 3), alle anderen Sätze werden wieder verworfen, was nicht effizient ist. Übrig bleiben nur Sätze, die laut Statistik den Betrag 47,11 haben. Oracle geht so vor, da FIRST_ROWS so programmiert ist, dass Sorts im Speicher verhindert werden.

Unter ALL_ROWS ergibt sich folgender Plan:

```
-------------------------------------------------------------------------------------
| Id  | Operation                     | Name                   | Rows  | Cost (%CPU)|
-------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT              |                        |   107 |   324   (1)|
|   1 |  SORT ORDER BY                |                        |   107 |   324   (1)|
|   2 |   NESTED LOOPS                |                        |   107 |   323   (0)|
|   3 |    NESTED LOOPS               |                        |   107 |   323   (0)|
|   4 |     TABLE ACCESS BY INDEX ROWID| ZV01_AUSGZAHLUNG      |   107 |   109   (0)|
|*  5 |      INDEX RANGE SCAN         | I_ZV01_AUSGZAHLUNG_03  |   107 |     3   (0)|
|*  6 |     INDEX UNIQUE SCAN         | PK_ZV01_OB             |     1 |     1   (0)|
|   7 |    TABLE ACCESS BY INDEX ROWID| ZV01_OB                |     1 |     2   (0)|
-------------------------------------------------------------------------------------

Predicate Information (identified by operation id):
---------------------------------------------------

   5 - access("ZV01_AUSGZAHLUNG"."BETRAG"=47.11)
   6 - access("ZV01_OB"."OBID"="ZV01_AUSGZAHLUNG"."OBID")
```

Oracle liest direkt die gewünschten Zeilen (Betrag 47,11) wie im Plan (Id 5) und führt später den Sort im Speicher aus (Id 1). Dieser Plan ist wesentlich effizienter.

Weitere Untersuchungen und Schilderungen zu FIRST_ROWS bedingten Problemen findet man bei Jonathan Lewis. [1] Auf Grund der Probleme musste ein Migrationspfad weg von FIRST_ROWS hin zu einem anderen Optimizer-Mode gefunden werden. Der Optimizer-Mode gehört zu den zentralen Einstellungen in einer Oracle-DB und beeinflusst potentiell die Pläne aller Statements. Daraus ergab sich die Aufgabe, Zehntausende von Statements und deren Pläne auf Änderungen zu untersuchen, denn die Pläne aller Statements konnten sich potentiell ändern. Um diese Aufgabe zu bewältigen, wurde zunächst ein PL/SQL-Script geschrieben, das die V$SQL der Produktionsanlage - unter Berücksichtigung von Bind-Variablen - überwacht und Pläne für ALL_ROWS und FIRST_ROWS

[1] https://jonathanlewis.wordpress.com/?s=first_rows

berechnet. Statements, die gleiche Pläne (Plan-Hash) in den beiden Modi hatten, wurden verworfen, da hier demzufolge kein Handlungsbedarf bestand.
Jetzt waren von den Zehntausenden Statements nur noch Hunderte Statements übrig, deren geänderte Pläne überprüft werden mussten. An diesem Punkt wurde dem Vorhaben jedoch aus Datenschutzgründen der notwendige Zugriff auf die Produktionsanlage verweigert.

Ein Mitarbeiter hatte schließlich eine Idee zur Lösung des Dilemmas:

Ein Plan in Test ist kein Plan in Produktion!
Zum Erzeugen von Plänen verwendet der Optimizer die Objekt- und System-Statistiken und nicht die Daten. Es war daher zu klären, ob es Datenschutzbedenken gibt, die Statistiken der Produktionsdatenbank zu exportieren und diese ins Testsystem zu bringen, wo ein Zugriff dann gestattet ist. Da es keine Datenschutzbedenken gab, werden die Produktionsstatistiken seit diesem Zeitpunkt tagesaktuell aus dem Produktionssystem ins Testsystem importiert. Dazu wird vom Enterprise-Scheduler der Debeka jede Nacht ein User mit dem Namen DKS_STATS in der Test-Datenbank erstellt. Aus der Produktionsumgebung (DKS_PROD) werden nur das Schema und die Statistiken per Oracle-Datapump (expdb und impdb) in den User übernommen. Die eigentlichen Daten werden nicht übernommen! Erreicht wird dies durch die Datapump-Option METADATA_ONLY. Die Statistiken werden vom Datapump immer übernommen, solange sie nicht per EXCLUDE=STATISTICS ausgeschlossen werden.
Die Datenbankadministration stellt des Weiteren sicher, dass andere Parameter wie die System-Statistiken identisch zum Produktionssystem sind, was auch sonst Grundvoraussetzung beim Aufbau von Testumgebungen ist. Somit konnten im Testsystem Pläne wie im Produktionssystem erzeugt werden.
Die oben genannten veränderten Pläne wurden anschließend auf mögliche Performance-Probleme hin untersucht. Mittlerweile ist DKS_STATS nicht nur der Name des Users in der Testumgebung, der die Produktionsstatistiken zur Verfügung stellt, sondern er ist zum Synonym für diese Vorgehensweise geworden. Nachfolgend findet sich eine schematische Darstellung der DB-Umgebungen.

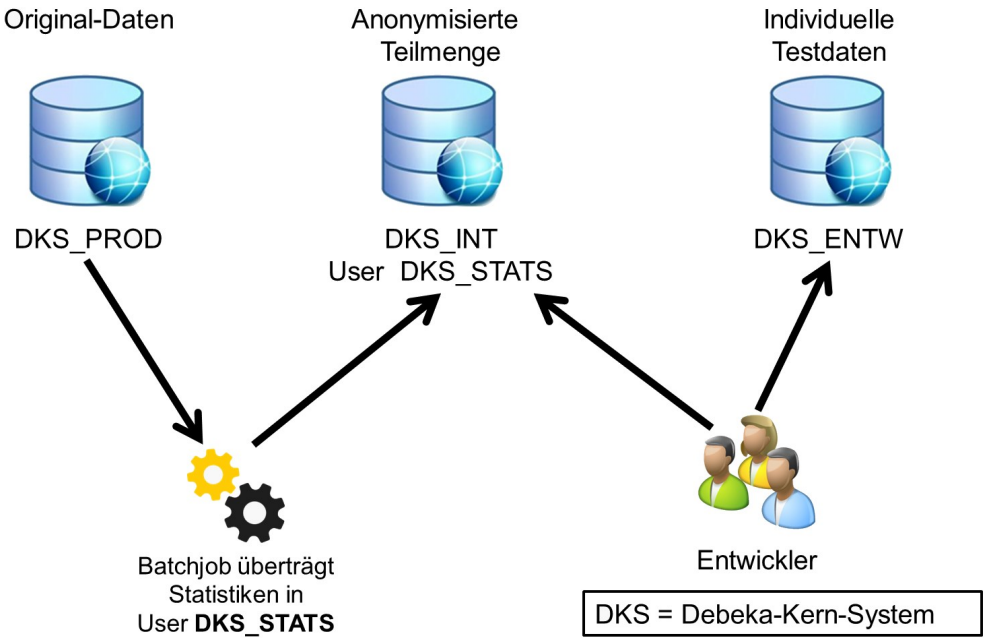

Abb. 1: grober Aufbau DB-Umgebungen DKS mit DKS-STATS

Überprüfen der Statements mit abweichenden Plänen aus der Migration
Um die Statements zu überprüfen, meldete sich ein Benutzer bei dem User DKS_STATS an und erzeugte für das fragliche Statement einen Plan. Anschließend wurden in einem ersten Schritt die Cardinalities und dann der Plan-Aufbau auf Plausibilität hin untersucht.

Prüfen der Cardinality
Zu diesem Zweck wurde die von Wolfgang Breitling beschriebene Methode „Tuning by Cardinality Feedback“ [2] für die Zwecke der Debeka abgewandelt. Breitlings Methode beruht, vereinfacht gesagt, auf dem Gedanken, die erwarteten Zeilen eines Plans (E-ROWS) mit den tatsächlichen Zeilen (A-ROWS) zu vergleichen und bei Abweichung zu reagieren. Denn bei Abweichungen kann ein Plan ab dieser Stelle auch leicht ungünstig kippen. Da der User DKS_STATS keine Daten enthält, ist es nur möglich, die E-ROWS im Plan zu liefern. Zum Abschätzen der A-ROWS wird auf die Erfahrung der Fachentwickler zurückgegriffen, um zu verifizieren, ob die E-ROWS passen könnten. Wenn dies nicht der Fall ist, muss mit den von Wolfgang Breitling beschriebenen Maßnahmen gegengesteuert werden. Hier wird einer der Kerngedanken beim Ansatz DKS_STATS deutlich: Einbeziehung der Fachentwickler. Anders als der DBA kennt der Fachentwickler meistens die Daten-Grundlage und ihre Semantik und kann wertvolle Hinweise für die Arbeit der DBAs liefern. Falls dem Anwender die Datengrundlage nicht vertraut ist, gibt es dafür auch einen Lösungsansatz im Abschnitt **„**Umgang mit Statements die Bindvariablen enthalten / Aufbereitung Statistikinformationen“

Plan-Aufbau
In vielen Fällen reicht der gerade besprochene Schritt aus. Sollte dies nicht der Fall sein, ist es zielführend, das Statement zu visualisieren[3]. So lässt sich ein optimaler Ausführungsweg finden und der Optimizer muss dann im Anschluss dazu gebracht werden, das Statement genauso auszuführen.

Zu schade für die Tonne
Nach erfolgreichem Abschluss der Migration, sollte DKS_STATS zunächst wieder verschwinden. Da es sich aber bei den Performance-Prognosen im Rahmen der Optimizer-Migration bewährt hatte, wurde es beibehalten, um Fachentwicklern beim Erkennen nicht optimaler SQL-Statements zu helfen. Die Fachentwickler konnten nämlich vorher in der Testumgebung ohne die entsprechenden Statistiken aus Produktion nur prüfen, ob das SQL logisch richtig ist. Sie konnten aber keine Aussage über das spätere Laufzeitverhalten treffen.
Außerdem stellt DKS_STATS eine sehr kosteneffiziente Testmöglichkeit dar, da es einfach mit in die bestehende Testumgebung integriert wurde. In Feldern, wo Datenschutzbedenken keine Rolle spielen, ist es ja sonst üblich, stattdessen die Produktionsdaten zu duplizieren, um dann im Test verlässliche Performance-Prognosen zu machen.

Nutzung DKS_STATS im Entwicklungsalltag
DKS_STATS steht allen Fachentwicklern über den SQL-Developer zur Verfügung. Beim ersten Kontakt ist die Begeisterung häufig erst einmal verhalten, da sie sich nun mit den für Außenstehende meist kryptischen Oracle-Plänen beschäftigen müssen. Die Erfahrung zeigt aber, dass das Lesen von Plänen für Programmierer gar nicht so schwierig ist, da sie im Prinzip eine prozedurale Anweisungsabfolge zum Beschaffen der gewünschten Daten sind. Ein Entwickler, der die möglichen Laufzeitunterschiede eines guten Plans gegenüber einem schlechten Plan dann einmal erlebt hat (der eigene Batchjob ist plötzlich in 30 min statt in 5:30 Std fertig), investiert auch gerne ein bisschen Zeit in das Lesen von Plänen. Nachfolgend sollen einige Verwendungen von DKS_STATS aus der Praxis aufgezeigt werden. Ergänzendes Material ist auf den entsprechenden Folien zu finden.

[2] http://www.centrexcc.com/Tuning%20by%20Cardinality%20Feedback.pdf
[3] https://www.simple-talk.com/sql/performance/designing-efficient-sql-a-visual-approach/

Beispiel: Abschätzen von Statement-Effizienz
Meistens gibt es mehrere Varianten, ein Statement zu schreiben (wie auch das Beispiel in den Folien zeigt) und viele verschiedene mögliche Pläne pro Statement. Erschwerend kommt hinzu, dass der Optimizer nicht aus jedem Statement selbständig einen optimalen Plan machen kann. Wenn der Optimizer einen Plan erstellt, ermittelt er aus vielen Varianten die günstigste (gemessen an der Cost). Da der Optimizer aber einigen Limitierungen unterliegt, wie auch DKS_STATS (siehe Abschnitt „Grenzen von DKS _STATS“), kann man die Kosten von Plänen von verschiedenen Statements nicht direkt vergleichen. Hier muss der Fachentwickler erkennen, ob der Weg, den der Optimizer genommen hat, auch sein Weg gewesen wäre. Stimmt die Zugriffsreihenfolge der Tabellen? Machen die ausgewählten Join-Methoden Sinn? (siehe das oben beschriebene Verfahren). Ein großer Vorteil, der sich in der Praxis gezeigt hat, ist dass die Fachentwickler plötzlich erkennen, dass es nicht nur die Statistiken sind, die die Optimizer-Aktionen bedingen, sondern dass das Schema auch eine große Rolle spielt. Das führt dazu, dass beim Design von Schemata mehr darauf geachtet wird, welche Statements (Zugriffswege) später einmal vorgesehen sind.

Umgang mit Statements, die Bindvariablen enthalten / Aufbereitung Statistikinformationen
Abhängig von der Werteverteilung einer Spalte in einer Tabelle, kann ein jeweils völlig anderer Plan sinnvoll sein. Also musste eine Möglichkeit gefunden werden, repräsentative Literale zu ermitteln. Zu diesem Zweck wurde der DKS-Table-Report geschrieben. Dieser Report nutzt die Reportfunktionalitäten im SQL-Developer, um Informationen aus den Oracle-User-Views abzufragen und aufzubereiten. Ausgehend von einer Tabelle kann sich der Fachentwickler übersichtlich Statistiken zu dem Datenbestand in Produktion abrufen, u.a. Anzahl der Zeilen, Indizes auf den Spalten einer Tabelle und deren Zusammensetzung und eben auch die Werteverteilung einer Spalte (min / max Value, Anzahl distinct Werte etc.). Außerdem kann ermittelt werden, ob eine Gleichverteilung (kein Histogramm) oder eine Ungleichverteilung vorliegt. Mit diesen Informationen kann der Fachentwickler dann über repräsentative Literale entscheiden. Die Fachentwickler nutzen diese Möglichkeit auch, um sich über den Datenbestand in Produktion zu informieren. Dabei lernen die Fachentwickler nebenbei, wie Oracle Informationen zu den Daten (die Statistiken) ablegt, welche Grenzen es dabei gibt und welche Auswirkungen auf die spätere Performance. Dieses Bewusstsein führt dann meistens zu der Erkenntnis, dass es wenig sinnvoll ist, Schema-Design und die Anlage von Indizes losgelöst von der Datenbank zu betreiben. Denn ein Schema, das die Anforderungen des Oracle-Optimizers und die Grenzen der Datenabbildbarkeit der verschiedenen Histogramme und Statistiken unberücksichtigt lässt, wird in der Praxis später nicht sehr performant laufen. Häufig ist dies im Umfeld des OR-Mapping zu beobachten.

Beispiel: Aufspüren inkorrekter Statistiken in Produktion
Ein Fachentwickler schreibt ein Statement mit einen Equi-Join mehrerer Tabellen. Er erwartet pro Tabellen-Zugriff ein Ergebnis von mehr als 100.000 Zeilen. Der Optimizer veranschlagt aber nur eine Zeile und entscheidet daher auf Nested Loop als Join-Methode. Hier wäre aber ein Hash-Join die bessere Wahl gewesen. Die Ursache in solchen Fällen sind häufig veraltete oder nicht korrekte Statistiken. Als Faustformel kann man sich folgendes merken:
Eine Cardinality von eins ist meistens verdächtig und immer ein guter Kandidat für eine Untersuchung.
Insbesondere im Umfeld von Partitionen muss man aufpassen, da es sowohl globale Statistiken für die Tabelle, als auch Statistiken für die einzelnen Partitionen gibt.

Beispiel: Index-Verwendung
Fachentwickler haben die Möglichkeit zu prüfen, ob ihre Indizes in Produktion verwendet werden. Darüber hinaus können sie sich mit Hilfe der oben genannten Reports Informationen beschaffen, die das Anlegen der Indizes erleichtern. Eine große Hilfe ist dies insbesondere beim Anlegen von zusammengesetzten Indizes, um eine fundierte Entscheidung über die Spaltenreihenfolge treffen zu können. In der Praxis hat sich jetzt schon öfters gezeigt, dass mit Hilfe von DKS_STATS „`Index Skip Scans`“ in „`Index Range Scans`“ umgewandelt werden konnten. Wenn Oracle auf „`Index Skip Scan`“ entscheidet, ist meistens die Anzahl der verschiedenen Werte (num_distinct) der ersten Spalte sehr gering, denn nur so kann ein „`Index Skip Scan`“ effizient funktionieren, da Oracle für jeden Wert in der ersten Spalte einen „`Index Range Scan`“ macht. In diesem Fall ist es manchmal möglich, durch nur kleinere Änderungen am SQL-Statement, z.B. durch Hinzufügen eines weiteren Filter-Predicates auf der ersten Spalte, ein logisch gleichwertiges, aber schnelleres Statement zu erhalten. Funktioniert dies nicht, ist eine Änderung bzw. Ergänzung der Indexstruktur notwendig. Auch wird durch die Möglichkeiten von DKS_STATS nochmal klar, dass ein Index auf der gedachten Spalte „Name“ keinesfalls Filter-Predicates wie „…`upper(name)`…“ unterstützen kann (Stichwort function based indexes).

Grenzen von DKS _STATS
Bisher sind wir in der Praxis auf einige Einschränkungen bei der Nutzung von DKS_STATS gestoßen. Diese stellen aber Sonderfälle dar, die den grundsätzlich Nutzen von DKS_STATS nicht mindern, aber der Vollständigkeit halber nicht unerwähnt bleiben sollen:

1. Cost-Abschätzung rekursiver Abfragen
Oracle scheint bei der Cost-Ermittlung eine bestimmte Tiefe der Rekursion anzunehmen. Daher ist es schwierig, mit DKS_STATS dort verlässliche Aussagen zu treffen. Hier kommt es auf die Erfahrung der Fachentwickler an, abzuschätzen, in wie weit die Rekursion zu den „Gesamtkosten“ des Plans beiträgt.

2. 1=0 Joins → Hohe Cost, aber Null Ausführungszeit

Manchmal ist es notwendig, Abfragen um Felder aus anderen Tabellen zu erweitern. Dazu werden diese Tabellen mit der Bedingung (1=0) gejoint, da so das ResultSet um die Felder der anderen Tabelle erweitert wird. Daten werden dabei nicht aus den Tabellen geladen, trotzdem gibt Oracle für die einzelnen Schritte eine Cost an, z. B. wenn man zwei Abfragen mit einem Mengenoperator (Union etc.) verbinden will, die unterschiedliche Abfragespalten haben. Des Weiteren wird ein „1=0 Join" auch manchmal eingesetzt, um Teile von Abfragen an- und ausschalten zu können.
Wie der folgende Plan zeigt, relativiert sich das Problem, da zwar die Kosten für den Objektzugriff im jeweiligen Abschnitt des Plans auftauchen, aber ignoriert werden können, denn sie addieren sich nicht zu den Gesamtkosten.

```
-------------------------------------------------------------------------------
| Id  | Operation              | Name   | Rows  | Cost (%CPU)| Pstart| Pstop |
-------------------------------------------------------------------------------
|   0 | SELECT STATEMENT       |        |  3923K|   324K  (1)|       |       |
|   1 |  MERGE JOIN OUTER      |        |  3923K|   324K  (1)|       |       |
|   2 |   PARTITION LIST ALL   |        |  3923K|   324K  (1)|     1 |    10 |
|*  3 |    TABLE ACCESS FULL   | KB_TAR |  3923K|   324K  (1)|     1 |    10 |
|   4 |   BUFFER SORT          |        |     1 |            |       |       |
|   5 |    VIEW                |        |     1 |            |       |       |
|*  6 |     FILTER             |        |       |            |       |       |
|   7 |      PARTITION LIST ALL|        |  4632K| 28910   (2)|     1 |    10 |
|   8 |       TABLE ACCESS FULL| KB_VN  |  4632K| 28910   (2)|     1 |    10 |
-------------------------------------------------------------------------------

Predicate Information (identified by operation id):
---------------------------------------------------

   3 - filter("KB_TAR"."K44TA"=253 OR "KB_TAR"."K44TA"=254 OR
              "KB_TAR"."K44TA"=362)
   6 - filter(NULL IS NOT NULL)
```

Der Filter „`NULL IS NOT NULL`" (Id 6) wird vom Optimizer eingefügt, wenn er erkennt das eine Filterbedingung (hier 1=0) niemals wahr werden kann. Dann führt die Datenbank diesen Teil des Plans nie aus.

Kontaktadresse:
Daniel Stein
Debeka
Abt. BO/S
Ferdinand Sauerbruch Str. 18
D-56072 Koblenz

E-Mail daniel.stein@debeka.de
Internet: www.debeka.de