Wenn es darum geht, effiziente Abfragen zu erzeugen, gibt es einige wenige Schlüsselkonzepte, die es zu verstehen gilt. Obwohl es „kostenbasierter Optimierer" heißt, sind es nicht primär die Kosten, auf die wir schauen müssen, um zu verstehen, warum der Cost-Based Optimizer (CBO) bestimmte Entscheidungen für die Erstellung von Ausführungsplänen getroffen hat.

Grundlagen zum Cost-Based Optimizer

Randolf Geist, unabhängiger Berater

Der Artikel stellt die Schlüsselkonzepte vor und zeigt, warum die Mengen- und Selektivitäts-Abschätzungen des CBO so elementar wichtig sind und welche Informationen er derzeit (noch) nicht berücksichtigt. Dies bedeutet auch, dass man manchmal wesentlich mehr als der CBO über seine Daten und Applikationen weiß und ihn daher in die richtige Richtung lenken muss, um effiziente Ausführungspläne zu generieren.

Grundlegend gibt es bei der Suche nach einem effizienten Ausführungsplan drei entscheidende Fragen, die zu klären sind:

1. Wie viele Zeilen werden erzeugt? / Welche Datenmenge wird erzeugt?
2. Wie sind die zu durchsuchenden Daten organisiert – eher zusammenhängend oder durcheinandergemischt (gemäß dem Suchkriterium)?
3. Wie groß ist die Wahrscheinlichkeit, dass die zu durchsuchenden Daten bereits im Cache sind?

Die Beantwortung dieser drei Fragen ist so wichtig, weil es vereinfacht ausgedrückt zwei Strategien gibt, wie die Daten verarbeitet werden können:

- *Ein großer „Job"*
  Hier ist vor allem die Frage nach der Datenmenge entscheidend für die Effizienz-Abschätzung der Operation
- *Mehrere kleinere „Jobs"*
  Hier sind die Fragen nach „Wie viele Zeilen?" (Wie oft muss der kleine Job ausgeführt werden?), „Wie sind die Daten organisiert?" (Wie groß ist der Aufwand pro Ausführung?) und dem „Caching" (Welcher Anteil der Daten liegt bereits im Cache?) entscheidend.

Werden die Fragen in diesem Zusammenhang richtig beantwortet, lässt sich die passende Strategie auswählen. Umgekehrt ist es wahrscheinlich, dass man sich bei falscher Beantwortung der Fragen für eine unpassende Strategie entscheidet, die weniger effizient sein kann, zu längerer Ausführungszeit der Abfrage führt und damit auch andere, konkurrierende Prozesse beeinflussen kann.

Der Cost-Based Optimizer

Es ist nicht überraschend, dass der CBO auf der Suche nach dem effizientesten Ausführungsplan versucht, ganz ähnliche Fragestellungen zu beantworten. Interessanterweise adressiert er aber nicht alle drei oben genannten Fragen im gleichen Umfang: Die erste Frage nach der Anzahl der Zeilen und des Datenvolumens wird ausführlich behandelt, wobei der CBO hier auf recht einfache Weise in die Irre geführt werden kann.

Schon bei der zweiten Frage nach der Organisation der Daten gibt es nur eine einzige Information, die dem CBO zur Verfügung steht: der sogenannte „Clustering Factor" von Indizes. Während dieser eine in vielen Fällen ausreichende Information für die Entscheidung darstellt, wie auf eine einzelne Tabelle zugegriffen wird (Zugriff per vorsortiertem Index oder Full Table Scan über die gesamte Tabelle), gibt es je nach Ausführungsplan für Operationen, die verschiedene Datenquellen verknüpfen, überhaupt keine Information für den CBO, wie diese in Relation zueinander organisiert sind.

Dies festzustellen ist eine Aufgabe, die nach heutigem Stand der Technik wahrscheinlich zu viel Zeit und Ressourcen in Anspruch nehmen würde, da es viele verschiedene Möglichkeiten gibt, wie Datenquellen verknüpft werden können: Die Reihenfolge der Datenquellen kann unterschiedlich sein und pro Datenquelle gibt es potenziell viele mögliche Zugriffsarten. Beides kann die Reihenfolge des Datenzugriffs verändern, zum Beispiel „Full Table Scan Tabelle C -> Index1 Tabelle A -> Index Tabelle B" gegenüber „Index2 Tabelle A -> Index Tabelle C -> Index Tabelle B", was bedeutet, dass alle möglichen Kombinationen evaluiert werden müssten, um diese Information sinnvoll verarbeiten und gegeneinander abwägen zu können.

Die dritte Frage nach dem Caching von Daten wird derzeit vom CBO überhaupt nicht berücksichtigt. Dieser geht in seinen Berechnungen immer davon aus, dass die Daten nicht im Cache liegen.

Dies alles bedeutet, dass der CBO je nach Daten und Ausführungsplan nur über unzureichende Informationen verfügt, um die genannten Fragen korrekt beantworten zu können und daher leicht eine unpassende Strategie auswählen kann. Weiterhin bedeutet dies bei ausreichendem Wissen über seine Daten und die Abfragen darauf, dass man unter Umständen diese Fragen wesentlich besser als der CBO beantworten und beurteilen kann, ob der CBO eine passende Strategie ausgewählt hat, beziehungsweise dass man ihn dabei unterstützen muss, die optimale zu finden.

Statistiken

Der CBO wendet im Grunde ein mathematisches Modell auf die ihm zur Verfügung stehenden Eingangsda-

ten (wie Objekt- und System-Statistiken) an und erstellt als Ergebnis einen Ausführungsplan. Eins der grundlegenden Probleme dieser Herangehensweise ist die Tatsache, dass diese Eingangsdaten, also die Statistiken, je nach Art der Abfrage die Daten eventuell nur unzureichend repräsentieren. Verwendet die Abfrage zum Beispiel Ausdrücke, die in den vorberechneten Objekt-Statistiken nicht abgebildet sind (zum Beispiel ein simples „UPPER(T1.NAME) = ‚MEIER'"), dann hat es der CBO unter Umständen schon schwer, die eigentlich sehr einfache Frage zu beantworten, wie viele Zeilen der Tabelle „T1" dieses Suchkriterium erfüllen.

Mengenabschätzungen

Wie bereits gesagt, versucht der CBO seine Abschätzungen standardmäßig basierend auf den vorberechneten Objekt-Statistiken zu erzeugen. Der Grund dafür liegt darin, dass der CBO darauf optimiert ist, den Ausführungsplan möglichst schnell zu erzeugen – für Abfragen, die nur den Bruchteil einer Sekunde dauern, würde das Untersuchen der tatsächlichen Daten („Wie viele Zeilen entsprechen Filterkriterium X?") im Verhältnis viel zu lange dauern. Der CBO verfügt jedoch unter dem Namen „Dynamic Sampling" über diese Funktionalität. Diese kommt allerdings standardmäßig nur in bestimmten Fällen zum Einsatz, zum Beispiel wenn überhaupt keine Objekt-Statistiken vorliegen. Das hat aber zur Folge, dass Abfragen auf Informationen, die von den Objekt-Statistiken nicht oder nur unzureichend abgedeckt sind, den CBO sehr leicht in die Irre führen können.

```
select
        count(t2.attr2)
from
        t1
      , t2
where
/*-------------------*/
        t1.attr1 = 1
and     t1.attr2 = 1
/*-------------------*/
and     t1.fk = t2.id
;
```

Listing 2

Dazu ein einfaches Beispiel: Zwei Tabellen „T1" und „T2" (jeweils 1.000.000 Zeilen) sollen miteinander verknüpft werden, „T1" hat einen Fremdschlüssel auf „T2" und „T2" hat einen eindeutigen Index auf dem Primärschlüssel. Zusätzlich wird ein Filter auf „T1" angewendet. Listing 1 zeigt, dass die gefilterten Spalten „ATTR1" und „ATTR2" der Tabelle „T1" eine ungleichmäßige Datenverteilung haben.

Wird nun auf „ATTR1" oder „ATTR2" gefiltert, scheint ein Histogramm von Vorteil zu sein, um den CBO über die ungleiche Verteilung der Daten in den Spalten zu informieren. Wird dies gemacht, kann man die zwei oben erwähnten Strategien leicht demonstrieren. Listing 2 zeigt die Abfrage. Betrachtet man die obige Datenverteilung in „T1.ATTR1" und „T1.ATTR2", wird offensichtlich, dass 900.000 Zeilen von „T1" dieses Suchkriterium erfüllen. Sofern der CBO dies erkennt, würde er wahrscheinlich die Strategie „ein großer Job" auswählen, da die „Kleine Job"-Strategie hier bedeuten würde, 900.000-mal auf „T2" per Index-Zugriff in einer Schleife zuzugreifen, was üblicherweise deutlich ineffizienter wäre. Listing 3 zeigt den vom CBO automatisch ausgewählten Ausführungsplan.

Aufgrund der Histogramme hat der CBO die Mengenabschätzung für den Filter auf „T1" im richtigen Bereich gemacht (Operation ID 3: 819.000 geschätzte (E-Rows) anstatt 900.000 tatsächliche (A-Rows)-Zeilen) und automatisch die „Ein großer Job"-Strategie gewählt. Dies kann auch anhand der „Starts"-Spalte verifiziert werden – jede Operation des Ausführungsplans ist zur Laufzeit genau einmal ausgeführt worden.

Dazu eine Anmerkung: Die normalerweise im Ausführungsplan nicht angezeigten Spalten „Starts" und „A-Rows" können über einen speziellen Modus aktiviert werden, indem man entweder den Hint „GATHER_PLAN_STATISTICS" verwendet oder den Parameter „STATISTICS_LEVEL" auf „ALL" setzt (nur auf Session-Ebe-

```
     ATTR1      ATTR2      COUNT
---------- ---------- ----------
         1          1     900000
     90001      90001         10
     90002      90002         10
     90003      90003         10
     90004      90004         10
     90005      90005         10
     90006      90006         10
     90007      90007         10
     90008      90008         10
     90009      90009         10
.
.
.
```

Listing 1

```
---------------------------------------------------------------
| Id  | Operation            | Name | Starts | E-Rows | A-Rows |
---------------------------------------------------------------
|   0 | SELECT STATEMENT     |      |      1 |        |      1 |
|   1 |  SORT AGGREGATE      |      |      1 |      1 |      1 |
|*  2 |   HASH JOIN          |      |      1 |   819K|    900K|
|*  3 |    TABLE ACCESS FULL| T1   |      1 |   819K|    900K|
|   4 |    TABLE ACCESS FULL| T2   |      1 |  1000K|   1000K|
---------------------------------------------------------------

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

   2 - access("T1"."FK"="T2"."ID")
   3 - filter(("T1"."ATTR1"=1 AND "T1"."ATTR2"=1))
```

Listing 3

```
select
        count(t2.attr2)
from
        t1
      , t2
where
/*------------------*/
        t1.attr1 = 90001
and     t1.attr2 = 90001
/*------------------*/
and     t1.fk = t2.id
;
```

Listing 4

ne zu empfehlen, da es einen deutlichen Overhead in der Ausführung mit sich bringt). Die zusätzlichen Spalten können dann über einen Aufruf von „DBMS_XPLAN.DISPLAY_CURSOR“ mit der Formatierungsoption „ALLSTATS LAST“ erzeugt werden. Wird eine ähnliche Abfrage durchgeführt, bei der nur wenige Zeilen von „T1“ die Filterbedingung erfüllen (siehe Listing 4), erhält man den in Listing 5 gezeigten Ausführungsplan.

Der CBO hat hier für die Filterung der Tabelle „T1“ (Operation ID = 4) anstatt 819.000, wie im vorherigen Beispiel, nur eine Zeile geschätzt. Das ist zwar einerseits um den Faktor „10“ falsch, da zehn Zeilen die Filterbedingung erfüllen, liegt jedoch andererseits hier grundsätzlich im richtigen Bereich. Folgerichtig hat der CBO die „Kleine Jobs“-Strategie gewählt: Auf die Tabelle „T2“ wird über den eindeutigen Index zugegriffen, und zwar zehnmal (Starts=10) in einer Schleife (NESTED LOOP), da dies effizienter erschien, als die gesamte Tabelle „T2“ in einem großen Schritt zu verarbeiten. Das Interessante an dem Beispiel ist, dass sich an dem Zugriff auf Tabelle „T1“ nichts ändert – es ist in beiden Beispielen ein sogenannter „Full Table Scan“, da kein sinnvoller alternativer Zugriffsweg zur Verfügung gestellt wurde.

Entgegen der häufig vorherrschenden Meinung kann also die Mengenabschätzung für eine Tabelle nicht nur beeinflussen, wie auf die Tabelle selbst zugegriffen wird, sie kann auch die Zugriffsart und die Reihenfolge anderer Operationen des Ausführungsplans maßgeblich prägen. Verwendet man eine einfache Abwandlung der ersten Abfrage, die den Großteil von „T1“ zurückliefert und besser mit der „Ein großer Job“-Strategie verarbeitet wird, dann wird das noch klarer (siehe Listing 6).

Durch die Verwendung der TRUNC-Funktion (Abschneiden von Nachkomma-Stellen) verändert sich das Ergebnis der Abfrage nicht, da es sich um Ganzzahlen in ATTR1 und ATTR2 handelt. Der CBO kennt dies alles jedoch nicht und sieht nur die Funktion „TRUNC(…)“. Für diesen Ausdruck liegen aber keine Statistiken vor – vor allem kann das Histogramm auf den Spalten „ATTR1“ und „ATTR2“ nicht mehr verwendet werden – und der CBO fällt zurück auf vorgegebene Standardwerte, die nichts mit den eigentlichen Daten zu tun haben (siehe Listing 7).

Wie erwartet, hat sich am Zugriff auf „T1“ nichts geändert, es ist immer noch ein Full Table Scan. Allerdings ist die Abschätzung bezüglich der resultierenden Zeilenanzahl von „T1“ grob falsch: 100 anstatt der bekannten 900.000. Dies hat dazu geführt, dass der CBO die falsche Strategie gewählt hat: Der Ausführungsplan greift 900.000-mal per „Kleine Job“-Strategie auf die Tabelle „T2“ zu, was je nach Tabellen- und Cache-Größe sowie der Verteilung der Daten in der Tabelle „T2“ zu einer sehr langen Laufzeit führen kann.

```
select
        count(t2.attr2)
from
        t1
      , t2
where
/*------------------*/
        trunc(t1.attr1) = 1
and     trunc(t1.attr2) = 1
/*------------------*/
and     t1.fk = t2.id
;
```

Listing 6

Der wichtigste Punkt hier ist also, dass auch einfache Mengenabschätzungen für Filteroperationen auf einzelnen Tabellen maßgebliche Auswirkung auf den gesamten Ausführungsplan haben können. Von daher ist es elementar wichtig, dass der CBO diese Abschätzungen im richtigen Bereich macht.

Im Laufe der verschiedenen Datenbank-Versionen sind zu diesem Zweck verschiedene Funktionalitäten hinzugefügt worden, angefangen von „Function-Based Indizes“ in der Version 8.1, um Ausdrücke zu indizieren (als Seiteneffekt der Indizierung auch Spalten-Statistiken für den Ausdruck), „Dynamic Sampling“ seit Version 9.2, um dem CBO die Möglichkeit zu geben, einen Blick auf die tatsächlichen

```
---------------------------------------------------------------------------
| Id  | Operation                     | Name    | Starts | E-Rows | A-Rows |
---------------------------------------------------------------------------
|   0 | SELECT STATEMENT              |         |      1 |        |      1 |
|   1 |  SORT AGGREGATE               |         |      1 |      1 |      1 |
|   2 |   NESTED LOOPS                |         |      1 |        |     10 |
|   3 |    NESTED LOOPS               |         |      1 |      1 |     10 |
|*  4 |     TABLE ACCESS FULL         | T1      |      1 |      1 |     10 |
|*  5 |     INDEX UNIQUE SCAN         | T2_IDX  |     10 |      1 |     10 |
|   6 |    TABLE ACCESS BY INDEX ROWID| T2      |     10 |      1 |     10 |
---------------------------------------------------------------------------

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

   4 - filter((«T1”.”ATTR1”=90001 AND «T1”.”ATTR2”=90001))
   5 - access(«T1”.”FK”=”T2”.”ID”)
```

Listing 5

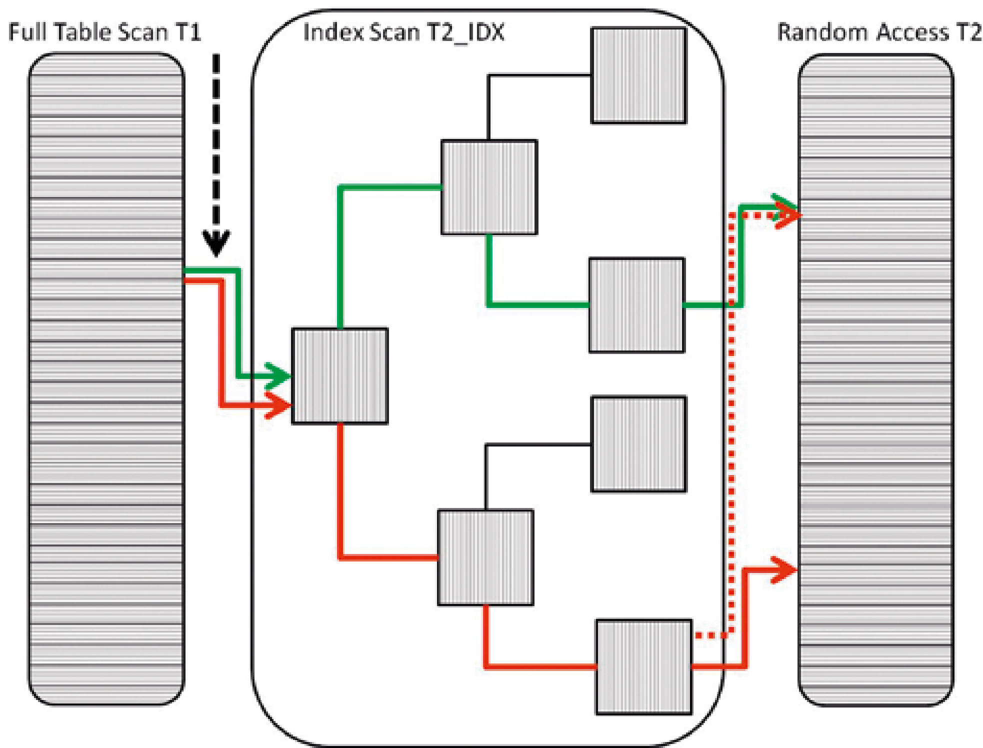

Abbildung 1: Schematische Darstellung zweier Durchläufe (erster „grün", zweiter „rot") einer Nested Loop T1->T2_IDX->T2. Die gestrichelte rote Linie zeigt an, dass der zweite Durchlauf der Nested Loop im gleichen Block der Tabelle „T2" landet wie der erste, bei der festen Linie muss auf einen anderen Block zugegriffen werden.

Daten zu werfen, bis hin zu „Virtual Columns" und „Extended Statistics" in der Version 11.1, die beide ermöglichen, auf Ausdrücken und sogar Spaltengruppen Statistiken zu erzeugen, unabhängig von eventuell erzeugten „Function-Based Indizes". Diese Mengenabschätzungen gilt es zu allererst zu überprüfen, da sie einfach nachvollzogen werden können („SELECT COUNT(*)" mit der Filter-Bedingung auf der Tabelle genügt) und es ausreichend Möglichkeiten gibt, sie zu korrigieren.

```
--------------------------------------------------------------------------
| Id  | Operation                    | Name   | Starts | E-Rows | A-Rows |
--------------------------------------------------------------------------
|   0 | SELECT STATEMENT             |        |      1 |        |      1 |
|   1 |  SORT AGGREGATE              |        |      1 |      1 |      1 |
|   2 |   NESTED LOOPS               |        |      1 |        |   900K|
|   3 |    NESTED LOOPS              |        |      1 |    100 |   900K|
|*  4 |     TABLE ACCESS FULL        | T1     |      1 |    100 |   900K|
|*  5 |     INDEX UNIQUE SCAN        | T2_IDX |   900K|      1 |   900K|
|   6 |    TABLE ACCESS BY INDEX ROWID| T2    |   900K|      1 |   900K|
--------------------------------------------------------------------------

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

   4 - filter((TRUNC("T1"."ATTR1")=1 AND TRUNC("T1"."ATTR2")=1))
   5 - access("T1"."FK"="T2"."ID")
```

Listing 7

Organisation der Daten

Für die Auswahl der richtigen Strategie ist also zum einen die Mengenabschätzung entscheidend. Um die Effizienz eines selektiven Zugriffs (zum Beispiel per Index) im Rahmen der „Kleine Job"-Strategie zu bestimmen, ist aber noch ein anderes Kriterium elementar wichtig: Wie zusammenhängend oder durcheinander sind die Daten in der zu lesenden Tabelle organisiert gemäß der Reihenfolge des jeweiligen Zugriffs?

Nehmen wir das Beispiel: Bei der „Kleine Job"-Strategie wird selektiv auf die Tabelle „T2" zugegriffen, indem für jede Zeile, die die Filterbedingung auf der Tabelle „T1" erfüllt, per Index die passende Zeile in „T2" gesucht wird. Für die Effizienz dieser Operation ist maßgeblich relevant, ob die jeweils passenden Zeilen in der Tabelle „T2" gemäß der Reihenfolge der Daten aus Tabelle „T1" zusammenhängend im gleichen Bereich (in den gleichen Blöcken) gespeichert sind oder ob zumeist für jede Zeile auf einen anderen Block zugegriffen werden muss.

Bei diesem Zugriffsmuster ist es meistens diese Organisation der Tabellendaten, die die Effizienz der Operation bestimmt, da Indizes häufig viel kleiner sind als Tabellen und daher viel mehr vom Caching profitieren können, was heißt, dass die Wahrscheinlichkeit für ein Caching der relevanten Index-Blöcke viel größer sein kann als bei den entsprechenden Tabellen-Blöcken. Abbildung 1 stellt schematisch dar, wie der Zugriff von „T1" auf „T2" im Rahmen der „Kleine Job"-Strategie aussieht.

Für jede Zeile, die im Rahmen des Full Table Scan der Tabelle „T1" dem Filterkriterium entspricht (repräsentiert im linken Teil der Grafik), wird der Index „T2_IDX" auf einen passenden Eintrag durchsucht. Dafür muss je nach Höhe des Index eine bestimmte Anzahl von sogenannten „Root-und-Branch-Blöcken" des Index durch-

sucht werden, bis der entsprechende Leaf-Block des Index gefunden wurde, in dem schließlich überprüft werden kann, ob ein passender Eintrag in „T2“ existiert oder nicht (mittlerer Teil der Grafik).

Falls ein passender Eintrag im Index identifiziert wurde, muss normalerweise der Rest der benötigten Daten für diese Zeile, die nicht im Index abgebildet sind, von der Tabelle selbst geholt werden (rechter Teil der Grafik). Obwohl der Zugriff auf die im Index referenzierte Tabellenzeile in den meisten Fällen mit einem einzigen Blockzugriff möglich ist (Ausnahmen sind zum Beispiel sogenannte „Migrated Rows“), bestimmt eben genau dieser Zugriff auf den Tabellen-Block in den meisten Fällen über die Effizienz dieser Operation.

Dies ist so, weil die Index-Blöcke aufgrund der Größe des Index und der Frequenz der Zugriffe meistens im Cache verbleiben können. Die Tabelle ist normalerweise viel größer und es kann je nach Art des Zugriffsmusters viel wahrscheinlicher sein, dass der gesuchte Block nicht im Cache ist und von Platte gelesen werden muss. Es handelt sich bei dem gezielten Lesen eines einzelnen Tabellen-Blocks um einen sogenannten „Random Access“, da für jeden Durchlauf der Schleife und damit für jede einzelne Zeile, die in „T1“ gefunden und für die eine passende Zeile in „T2“ gesucht wird, theoretisch auf einen anderen Block der Tabelle „T2“ zugegriffen werden muss. Im schlechtesten Fall kann dies sogar dazu führen, dass der gleiche Block von „T2“ mehrfach von Platte gelesen werden muss, da der vorherige Zugriff auf den gleichen Block bereits wieder von nachfolgenden Zugriffen aus dem Cache verdrängt wurde.

Wird aber beim Durchlauf der Schleife für eine bestimmte Anzahl an Iterationen immer wieder auf den gleichen Block (oder eine kleinere Anzahl von Blöcken) der Tabelle „T2“ zugegriffen, bleibt der Block nach dem ersten Zugriff im Cache und muss nicht mehr von Platte gelesen werden. Dies kann dramatische Unterschiede für die Effizienz solcher Operationen bedeuten, da ein typischer „Random Access“ auch von sehr schnellen Festplatten heutzutage immer noch zwischen drei und fünf Millisekunden dauert (Massenspeicher ohne rotierende Massen wie SSDs können hier deutlich schneller sein), während der Zugriff auf einen Block im Cache im unteren Mikrosekunden-Bereich liegt.

Legt man diese Zahlen einer einfachen Berechnung zugrunde, so benötigt der Zugriff auf 1.000 Zeilen der Tabelle „T2“ allein für das Lesen der 1.000 Blöcke von Festplatte zwischen drei und fünf Sekunden, unter Umständen auch deutlich länger, während das Zugreifen auf die gleiche Anzahl Blöcke aus dem Cache immer noch im Millisekunden-Bereich liegt.

Allgemein wird das eine Zugriffs-Muster („für jede identifizierte Zeile von „T2“ muss auf einen anderen Tabellen-Block zugegriffen werden“) als „Scattered“ bezeichnet, während das andere („für jede identifizierte Zeile von „T2“ kann wiederholt auf die gleichen Tabellen-Blöcke zugegriffen werden“) „Clustered“ genannt wird. Wichtig ist in diesem Zusammenhang zu verstehen, dass je nach Zugriffsart die gleiche Tabelle „Scattered“ für eine Art von Zugriff sein kann, aber „Clustered“ für eine andere. Genauer gesagt kann eine Tabelle normalerweise nur für genau eine Zugriffsart „Clustered“ sein und für alle anderen „Scattered“.

Proaktives Design

Mit diesem Wissen ist die Optimierung solcher Zugriffe, die als kritisch für eine Anwendung identifiziert werden, proaktiv möglich. Gerade bei OLTP-Anwendungen, die häufig gezielt kleinere Datenmengen suchen, kann eben genau diese Organisation von Daten einen großen Unterschied in Bezug auf die Effizienz beim Datenzugriff bedeuten. Daher sollten idealerweise schon während des Designs der Applikation die Daten und die wichtigsten Abfragen auf diese Daten bekannt sein.

Mithilfe dieses Wissens kann dann evaluiert werden, welche Art der Indizierung oder der Verwendung anderer Speicherungsmethoden den Zugriff auf die Daten entscheidend verbessern kann, ohne beim Modifizieren der Daten zu viel Zeit zu verlieren. Insbesondere die Verwendung von Clustern (Index Cluster oder Hash Cluster) oder Index-Organized Tables (IOTs) bietet je nach Anwendungsfall die Möglichkeit, die Organisation der Daten proaktiv zu beeinflussen und den Zugriff auf die Daten entscheidend zu beschleunigen.

Natürlich können diese alternativen Speicherungsmethoden nicht ungesehen eingesetzt werden. Die genauen Zugriffsmuster sowohl beim Lesen als auch beim Schreiben der Daten müssen bekannt sein, da ansonsten leicht das Gegenteil erreicht werden kann: Sowohl Lese- als auch Schreibzugriff können deutlich ineffizienter werden. Zudem ist ein Wissen über die spezifischen Verhaltensweisen von Clustern und IOTs erforderlich, um vorab abschätzen zu können, welche der alternativen Speicherungsmethoden welche Vor- und Nachteile mit sich bringen. Sind zum Beispiel viele sekundäre Indizes nötig, kann eine IOT zu insgesamt schlechterer Performance führen, da sekundäre Indizes für IOTs grundsätzlich anders funktionieren als für normale Tabellen.

Cluster können aus verschiedenen Gründen sehr ineffizient werden und erlauben bestimmte Operationen und Zugriffsmuster, je nach Variante, nicht. Darüber hinaus unterstützen derzeitige Versionen keine Partitionierung von Clustern. Insgesamt gesehen ist es also sehr wichtig, bereits während der Design-Phase einer Applikation Wissen bezüglich der genannten Kernfragen aufzubauen, um die Organisation der Daten in der Datenbank optimal darauf abstimmen zu können.

Randolf Geist
info@sqltools-plusplus.org