ANFRAGEOPTIMIERUNG IN ORACLE

Enrico Genauck
37327 IN04

ANFRAGEOPTIMIERUNG IN ORACLE

Enrico Genauck
37323 IN04

Einleitung

Die Optimierung einer Anfrage an eine relationale Datenbank ist von großer Wichtigkeit und kann die Geschwindigkeit der Abarbeitung erheblich beeinflussen, insbesondere für komplexe SQL-Ausdrücke.

Aufgabe der Optimierung ist es, die beste Strategie zur Ausführung einer Abfrage zu bestimmen, dies schließt beispielsweise Entscheidungen mit ein, ob Indizes genutzt werden sollen oder welche Join-Verfahren beim Verbinden mehrerer Tabellen anzuwenden sind. Da Anwender oder Anwendungen im Allgemeinen auch sehr komplexe Anfragen stellen können, ist die Robustheit und Ausgereiftheit der Optimierung wichtig und notwendig.

Anfrageoptimierer arbeiten meist kostenorientiert, dass heißt sie generieren mehrere Ausführungspläne und berechnen die jeweiligen Kosten. Kosten sind hier ein Maß für die Aufwändigkeit einer Anfrage und berücksichtigen den Berechnungsaufwand, die Transaktionskosten der Datenein- und ausgabe eines SQL-Ausdrucks, die Speicherungskosten und die Kommunikationskosten. Anhand der Kosten für jeden Ausführungsplan wird der günstigste ausgeführt.

Allgemeiner Ablauf

Die Optimierung einer SQL-Anfrage lässt sich im Orcacle-System in vier große Bereiche einteilen:

1. SQL-Transformation
1. Ausführungsplan-Selektion
2. Kosten und Statistiken
3. Dynamische Laufzeitoptimierung

Während der **SQL-Transformation** wird der geparste SQL-Term, welcher aus verschachtelten und untereinander abhängigen Anfrageblöcken besteht, in eine semantisch äquivalente Form umgewandelt, die jedoch während der weiteren Verarbeitung durch die Optimierung bessere Ausführungspläne liefert. Grundlegende Transformationen umfassen:

View Merging

Jede View der Ursprungsabfrage wird in eine separate Query expandiert. Für diese werden dann Ausführungspläne erstellt. Meist resultiert dies in teuren Plänen, dann wird die separate Query mit der ursprünglichen Abfrage verschmolzen.

```
CREATE VIEW TEST_VIEW AS
SELECT ENAME, DNAME, SAL FROM EMP E, DEPT D
WHERE E.DEPTNO = D.DEPTNO;
SELECT ENAME, DNAME FROM TEST_VIEW WHERE SAL > 10000;
```

Eine einfache View und Query

```
SELECT ENAME, DNAME FROM EMP E, DEPT D
WHERE E.DEPTNO = D.DEPAND E.SAL > 10000;
```

Query nach dem View Merging

Predicate Pushing

Gibt es View-Queries, welche nicht verschmolzen wurden, so werden relevante Prädikate aus der Ursprungsabfrage in die View-Query verschoben. Dieses Vorgehen verbessert die Ausführungspläne der Unterabfrage, da die Prädikate als Index oder Filter agieren können.

```
CREATE VIEW EMP_AGG AS
  SELECT
    DEPTNO,
    AVG(SAL) AVG_SAL,
  FROM EMP
  GROUP BY DEPTNO;
```

View Definition

```
SELECT DEPTNO, AVG_SAL FROM EMP_AGG WHERE DEPTNO = 10;
```

Anfrage

```
SELECT DEPTNO, AVG(SAL)5 FROM EMP WHERE DEPTNO = 10
GROUP BY DEPTNO;
```

Optimierte Anfrage durch Predicate Push

Subquery Unnesting

Oft können verschachtelte Unterabfragen mit einem Join aufgelöst werden. Ist dies nicht möglich, so wird versucht, die Abfragen auf effiziente Weise neu anzuordnen um über die Reihenfolge Geschwindigkeitsvorteile zu erzielen.

Query Rewrite with Materialized Views

Wenn in der zu optimierenden Anfrage Teile gefunden werden, welche mit der Definition einer materialisierten Sicht übereinstimmen, so wird diese zur Abfrage hinzugezogen und eine aufwändige Neuberechnung entfällt.

```
CREATE MATERIALIZED VIEW SALES_SUMMARY
AS SELECT SALES.CUST_ID, TIME.MONTH, SUM(SALES_AMOUNT) AMT
FROM SALES, TIME
WHERE SALES.TIME_ID = TIME.TIME_ID
GROUP BY SALES.CUST_ID, TIME.MONTH;
```

Materialized View

```
SELECT CUSTOMER.CUST_NAME, TIME.MONTH, SUM(SALES.SALES_AMOUNT)
FROM SALES, CUSTOMER, TIME
WHERE SALES.CUST_ID = CUST.CUST_ID
AND SALES.TIME_ID = TIME.TIME_ID
GROUP BY CUSTOMER.CUST_NAME, TIME.MONTH;
```

Nicht optimierte Anfrage

```
SELECT CUSTOMER.CUST_NAME, SALES_SUMMARY.MONTH, SALES_SUMMARY.AMT
FROM CUSTOMER, SALES_SUMMARY
WHERE CUSTOMER.CUST_ID = SALES_SUMMARY.CUST_ID;
```

Optimierte Anfrage

OR-expansion

Enthält die WHERE-Klausel einer Anfrage OR-Verknüpfungen, so wird die Klausel in kleinere Unterabfragen mit UNION ALL aufgeteilt, welche jeweils keine OR-Verknüpfung mehr enthalten.

In der Phase der **Ausführungsplan-Selektion** wählt der Optimierer einen Plan, welcher die Abarbeitung der Anfrage beschreibt. Dazu gehört die Reihenfolge, in welcher auf die Tabellen zugegriffen wird, wie die Tabellen per JOIN verbunden werden und ob die Tabellen über ihren Index angesprochen werden. Es werden mehrere Pläne betrachtet und der beste ausgewählt.

Die Bewertung jedes Ausführungsplans erfolgt anhand von **Kosten und Statistiken**. Hierzu werden Schätzungen für alle Arbeitschritte herangezogen, welche auf dem Wissen über die Hardwareressourcen des Systems beruhen. Solches Wissen umfasst Informationen über die CPU, den Speicher und Eingabe-, Ausgabekanäle. Außerdem werden auch statistische Daten über die Datenbank (Tabellen, Indizes, materialisierte Sichten) genutzt.

Informationen, welche erst zur Laufzeit der Abfrage, wie zum Beispiel die Systemauslastung, fließen während der **dynamischen Laufzeitoptimierung** in die Abfrageabarbeitung ein.

Ausführungspläne für Joins

Bei der Auswahl eines Ausführungsplanes betrachtet der Optimierer ob ein Join-Statement von zwei oder mehr Tabellen mindestens eine Ergebniszeile zurück liefert. Solche Bedingung wird anhand von UNIQUE- und PRIMARY KEY-Constraints der Tabellen entschieden.

Ist solch eine Situation gegeben, so werden diese Tabellen nach vorne in der Join-Reihenfolge gestellt. Danach wird der Join der restlichen Tabellen optimiert.

Nested Loop Join

Nested Loop Joins werden verwendet, wenn kleinere Datenmengen verbunden werden sollen und die Join-Bedingung eine effiziente Art und Weise ist, auf die zweite Tabelle zuzugreifen.

Der Optimierer erklärt eine der beiden Tabellen zur äußeren, die andere zur inneren Tabelle. Nun werden für jede Zeile der äußeren Tabelle alle Zeilen der inneren Tabelle gelesen. Im Falle einer erfüllten Bedingung werden die verbundenen Zeilen der Ausgabemenge hinzugefügt.

Hash Join

Hash Joins kommen bei großen Datensätzen zur Anwendung. Hier wird die kleinere der beiden Tabellen genutzt, um eine Hash Table auf dem Join-Schlüssel zu erstellen. Danach wird die größere der beiden Tabellen gegen die Hash Table gescannt um die vereinigten Zeilen zu finden.

Diese Methode eignet sich sehr gut, wenn die kleinere Tabelle in den freien Arbeitsspeicher passt. Dann beschränken sich die Kosten auf eine einfache Leseoperation über die Daten beider Tabellen.

Sort Merge Join

Sort Merge Joins kommen in ähnlichen Fällen zur Anwendung, wie Hash Joins. Sie kosten weniger als ein Hash Join, wenn die Tabellen vorsortiert sind und für die Join-Operation keine weitere Sortierung notwendig ist.

Ein Sort Merge Join besteht aus zwei Schritten:

- Beide Tabellen werden anhand des Join-Schlüssels sortiert
- Die sortierten Listen werden verschmolzen

Cartesian Join

Der Optimierer entscheidet sich für einen Cartesian Join, wenn keine Bedingung an den Join geknüpft ist. Hier wird jede Zeile der einen Tabelle mit jeder Zeile der anderen Tabelle verknüpft. Das Ergebnis ist das kartesische Produkt beider Datenmengen.

Hints

Jeder der genannten Join-Methoden kann mit einem Hint vom Autor einer SQL-Anfrage erzwungen werden.

Hints geben dem Autor die Möglichkeit in die Hand, explizit auf den Ausführungsplan Einfluss zu nehmen. In seltenen Fällen können damit Queries gezielt verbessert werden, da der Autor über Wissen verfügt, welches der Optimierer im System nicht nutzen kann, das aber entscheidend für die Wahl des Ausführungsplans ist.

In den meisten Fällen jedoch können Hints genutzt werden, um gezielt die Leistung einer optimierten Anfrage mit einer suboptimalen Anfrage zu vergleichen.

Hints unterteilen sich in verschiedene Arten, abhängig vom Gebiet, welches sie beeinflussen.

Hints für Zugriffsmethoden

Mit diesen Hints kann eine bestimmte Art des Tabellenzugriffs erzwungen werden, sofern die Methode anwendbar ist. Sollte sie nicht anwendbar sein, so wird der entsprechende Hint ignoriert.

FULL, CLUSTER, HASH, INDEX, NO_INDEX, INDEX_ASC, INDEX_COMBINE, INDEX_JOIN, INDEX_DESC, INDEX_FFS, NO_INDEX_FFS, INDEX_SS, INDEX_SS_ASC, INDEX_SS_DESC, NO_INDEX_SS

Hints für Join Methoden

USE_NL, NO_USE_NL, USE_NL_WITH_INDEX schalten das Nutzen von Nested Loops ein oder aus.

USE_MERGE, NO_USE_MERGE erzwingen oder verbieten das Nutzen von Sort Merge Joins

USE_HASH, NO_USE_HASH bestimmen, ob Hash Joins verwendet werden sollen.

Statistiken

Die gesammelten Daten des Optimierers umfassen Angaben, wie:

- Tabellenstatistiken

  Anzahl der Zeilen, Anzahl der Blöcke, Durchschnittliche Zeilenlänge

- Spaltenstatistiken

  Anzahl verschiedener Werte, Anzahl von NULL-Werten Datenverteilung als Histogram

- Indexstatistik

  Anzahl von Blatt-Blöcken, Tiefe des Indexes

- Systemstatistik

  Ein- und Ausgabeperformanz, CPU Auslastung

Diese Daten werden genutzt um die Kosten eines Ausführungsplans zu berechnen. Gespeichert sind sie im „Data-Dictionary“ und können eingesehen werden.

```
BEGIN
  DBMS_STATS.GATHER_table_STATS (OWNNAME => 'OE', TABNAME => 'INVENTORIES',
  METHOD_OPT => 'FOR COLUMNS SIZE 10 quantity_on_hand');
END;
```

```
SELECT column_name, num_distinct, num_buckets, histogram
  FROM USER_TAB_COL_STATISTICS
 WHERE table_name = 'INVENTORIES' AND column_name = 'QUANTITY_ON_HAND';
```

```
COLUMN_NAME                    NUM_DISTINCT NUM_BUCKETS HISTOGRAM
------------------------------ ------------ ----------- ---------------
QUANTITY_ON_HAND                        237          10 HEIGHT BALANCED
```

```
SELECT endpoint_number, endpoint_value
  FROM USER_HISTOGRAMS
 WHERE table_name = 'INVENTORIES' and column_name = 'QUANTITY_ON_HAND'
  ORDER BY endpoint_number;
```

```
ENDPOINT_NUMBER ENDPOINT_VALUE
--------------- --------------
              0              0
              1             27
              2             42
              3             57
              4             74
              5             98
              6            123
              7            149
              8            175
              9            202
             10            353
```

Histogram

Empfohlen wird, das Sammeln der Statistiken dem System selbst zu überlassen. Dann werden die Daten regelmäßig zusammengestellt und aktualisiert. Da die automatische Aktualisierung immer zu einem fixen Zeitpunkt geschieht, können sich Fälle ergeben, bei denen eine manuelle Statistikaktualisierung nötig ist. Zum Beispiel erfolgt die automatische Aktualisierung während der Nacht, eine große und wichtige Tabelle jedoch wird häufig tagsüber stark verändert. Dadurch ist die Statistik über diese Tabelle meist veraltet.

Eine Lösungsmöglichkeit ist es, die Statistik der Tabelle auf NULL zu setzen und zu sperren. Dadurch wird der Optimierer gezwungen, bei jeder Abfrage als Teil der Optimierung die Statistik neu zu berechnen.

```
BEGIN
  DBMS_STATS.DELETE_TABLE_STATS('OE','ORDERS');
  DBMS_STATS.LOCK_TABLE_STATS('OE','ORDERS');
END;
```

Löschen und sperren einer Statistik

Eine manuelle Statistikerstelllung ist mit dem PL/SQL-Paket DBMS_STATS möglich.

Mit den umfangreichen Analysemöglichkeiten der Statistiken, Hints und dem Kommando EXPLAIN PLAN zur Anzeige des gewählten Ausführungsplanes bietet Oracle umfassende Werkzeuge zur Optimierung von Anfragen und zum Nachvollziehen dieser.

Quellen

Gunter Saake, Andreas Heuer, Kai-Uwe Sattler; Datenbanken - Implementierungstechniken,

2. Auflage, mitp, 2005

Query Optimization in Oracle Database10g Release 2, An Oracle White Paper, Juni 2005

www.oracle.com/technology/products/bi/db/10g/pdf/twp_general_query_optimization_10gr2_0605.pdf

Oracle Database Performance Tuning Guide 10g Release 2 (10.2), März 2008

http://www.oracle.com/pls/db102/homepage