SQL Kochbuch

Von Thomas Kestler, elevato GmbH

Einleitung

Dieses „Kochbuch“ ist eine kompakte Zusammenfassung von Themen, die einem bei SQL immer wieder begegnen. Es erhebt keinen Anspruch auf Vollständigkeit, sondern ist als kleines Nachschlagewerk (nicht zuletzt für den Autor selbst) gedacht. Die Ausführungen basieren auf Oracle und MySQL.

Tipps und Tricks

Im Folgenden ein paar Tricks aus der Praxis.

Temporäre Views

Die meisten Datenbanken unterstützen temporäre Views:

```
SELECT * FROM (
 SELECT * FROM xxx
) WHERE id > 10 AND station = 1;
```

Der gelbe Bereich ist die temporäre View. Man hätte auch folgendes tun können:

```
CREATE VIEW V_BLAH AS SELECT * FROM xxx;
```

und dann eben auf V_BLAH zugreifen können. Der Effekt ist der gleiche, nur spart man sich die View-Definition. Das Feature geht aber noch deutlich weiter: die WHERE-Bedingungen der äußeren Anweisung wirken direkt auf die temp. View. Man muss also nicht Angst haben, dass die Datenbank zuerst alle Daten aus xxx liest und dann gegen die äußeren Bedingungen filtert.

Das lässt sich beliebig oft schachteln:

```
SELECT * FROM (
 SELECT 0 as VAL1, KUNDE_ID, KUNDE_NAME FROM (
  SELECT * FROM xxx
 )
) WHERE id > 10 AND station = 1;
```

Trifft man auf Unbestimmtheiten (Ambiguitäten), weil bei einem Join zwei gleichnamige Spalten ins Spiel kommen, so kann man sich mit einer temp. View behelfen:

```
SELECT seriennummer
FROM (SELECT b.Produkt_Nr, b.Auftrags_nr, b.Seriennummer,
b.zeitpunkt_messung, f.maschinen_nr, b.Seq_nr
      FROM V_Status b, V_Fehler b
     WHERE f.snr_id = b.snr_id AND f.seq_nr = b.seq_nr AND f.maschinen_nr =
b.maschinen_nr
)
```

Im obigen Fall bestand das Problem, dass `maschinen_nr` in beiden Views enthalten ist und somit nicht eindeutig. Darum habe ich den Join zwischen den beiden Views in eine temp. View gepackt und mich mit `f.maschinen_nr` auf die

Spalte in V_Fehler festgelegt. Ach ja, dass hier Views gejoint werden und nicht Tabellen, ist – wie oben erklärt – egal.

Sub-Select == Join

Wenn man sich mal festgefahren hat mit einem komplexen SQL-Skript, dann hilft es sich daran zu erinnern, dass Subselect und Join identisch sind. Ein Beispiel:

```
SELECT cust.name, …,
 (SELECT COUNT(*)
  FROM callrecord c
  WHERE c.customer_id = cust.customer_id
  AND c.reason = 'FAILED'
 ) AS CNT_FAILED_CALLS
FROM customer cust
```

Das Entscheidende ist, dass die innere Query auf die äußere (`cust.customer_id`) zugreift. Letztlich ist auch dies ein Join, aber in dieser Form viel leichter zu lesen. Nun lassen sich später weitere Speudospalten hinzufügen:

```
SELECT cust.name, …,
 (SELECT COUNT(*)
  FROM callrecord c
  WHERE c.customer_id = cust.customer_id
  AND c.reason = 'FAILED'
 ) AS CNT_FAILED_CALLS,
 (SELECT COUNT(*)
  FROM callrecord c
  WHERE c.customer_id = cust.customer_id
  AND c.reason = 'CANCELED'
 ) AS CNT_CANCELED_CALLS
FROM customer cust
```

Tuning

Datenbank-Tuning ist eine Wissenschaft für sich. Ich habe das lange für INFORMIX und Oracle gemacht. Häufig findet man die Erwartung vor, man müsste nur an ein paar Konfigurationsparametern schrauben, dann würde es schon schneller gehen.

Vor jeder Maßnahme müssen genau Analysen stehen, um den Engpass (Bottleneck) zu identifizieren. Dazu müssen reproduzierbare Verhältnisse existieren, z. B. realistische Lastsituation, Datenbank genügend lange im Betrieb (Cache warm gefahren). Da alleine dieses Vorbereiten mehrere Stunden dauern kann, wird klar, dass die Turnuszeiten zur Überprüfung von Änderungen sehr lange sind.

Moderne Datenbanksysteme tunen sich mittlerweilen selbst, so dass man durch eigenmächtige Änderungen an der Konfiguration wohl eher nur Verschlechterungen erreicht.

SQL analysieren

Sehr oft reicht es mir, das SQL-Logging einer Anwendung anzuschalten. Wenn für eine Bildschirmseite hunderte von SQL-Statements abgefeuert werden, dann ist klar, dass das Problem in der Anwendung liegt. Meist passieren dann Lazy-Loads durch den ORM-Mapper. Durch entsprechende Join-Anweisungen in den Queries kann dies ganz einfach behoben werden.

Wenn eine einzelne Anweisung lange läuft, dann analysiere ich zunächst die Selektionskriterien (WHERE-Bedingung). Häufige Selektionskriterien sollte durch Indices unterstützt werden. Abfragen wie LIKE '%abc%' sind aber für einen Index ungeeignet (selbst für Function-Based-Index). Gut, der Optimizer stellt diese Bedingung hinten an (Filterung statt Zugriff), sofern noch weitere, selektivere Bedingungen existieren.

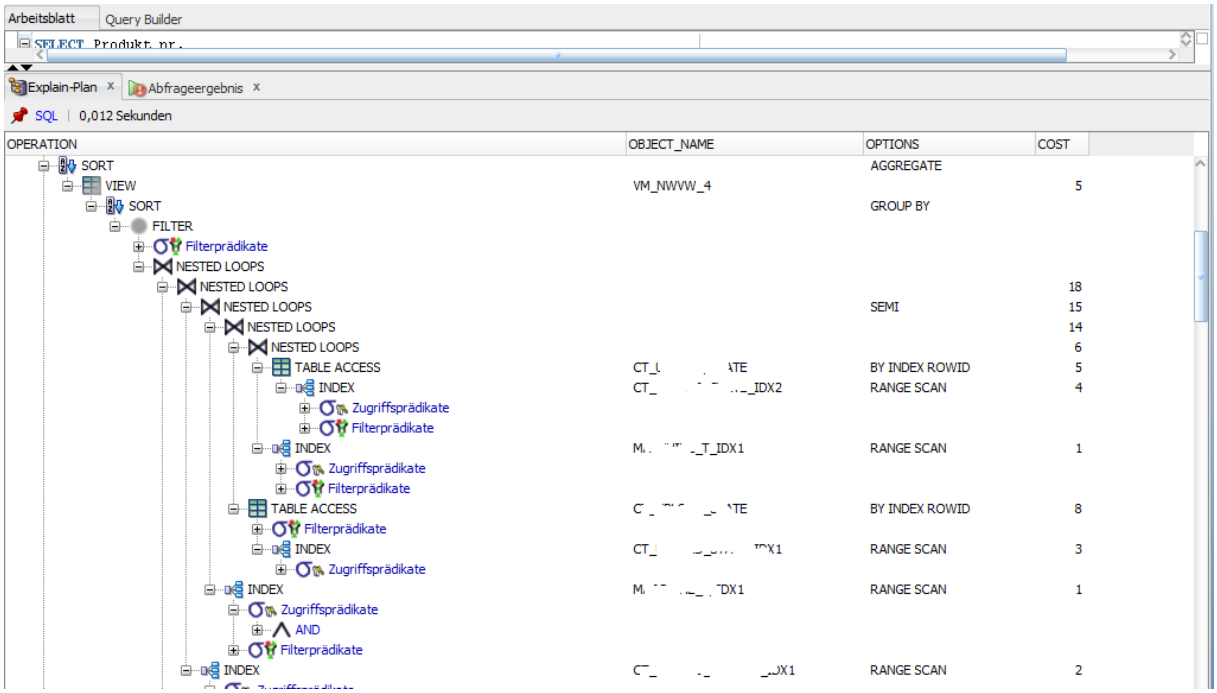
Selektivität ist relativ. Beispiel: eine Tabelle hat diese Status-Verteilung:

Status	Anzahl
COMPLETED	1.000.000
BROKEN	10
WORKING	250

BROKEN und WORKING sind sehr selektiv, COMPLETED nicht. Würde man einen Index auf das Status-Feld anlegen, so muss der Optimizer von Fall zu Fall entscheiden, ob der Zugriff über den Index Vorteile bringt oder nicht – und das ist nur bei selektiven Werten der Fall. Für diese Entscheidung benötigt der Optmizer allerdings aktuelle Statistiken.

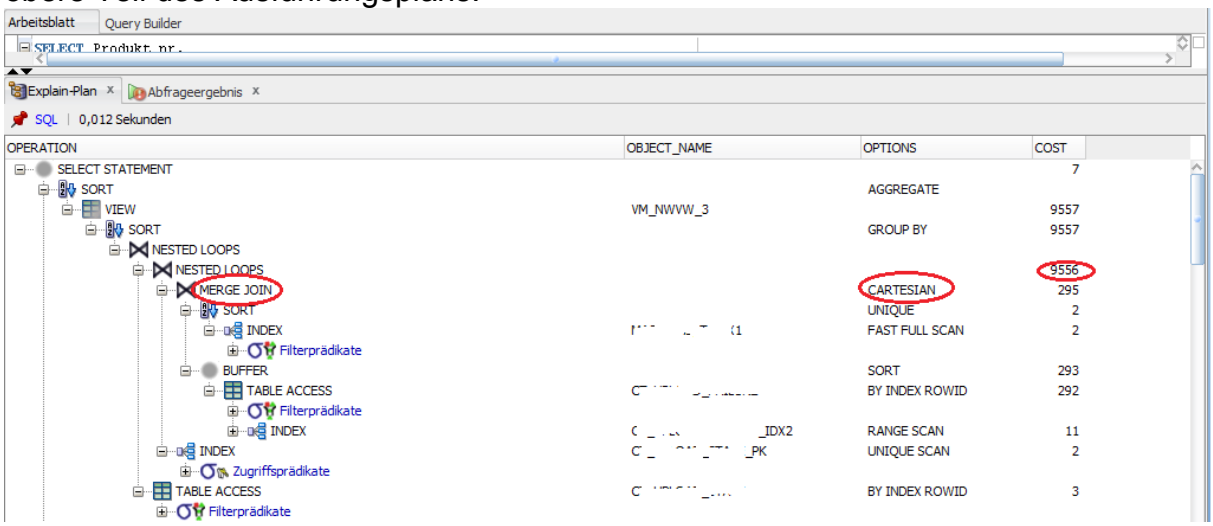
Ausführungspläne verstehen

Oracle SQLDeveloper macht die Erstellung von Ausführungsplänen sehr einfach. SQL im Worksheet ausführen und Plan ansehen. Das mit dem Verstehen ist schon schwieriger:



Der obige Ausführungsplan ist schon recht komplex, er passt gar nicht auf den Bildschirm, obwohl ich schon viele eingeklappt habe (sensible Daten habe ich absichtlich ausradiert).

Ich will hier keine Abhandlung über Oracle Tuning liefern, dazu verweise ich auf die gute Oracle-Dokumentation. Mein erster Blick gilt immer den Kosten (das teuerste ist hier ein Merge-Join und oben rausgerutsch) und FULL TABLE SCANS. Hier der obere Teil des Ausführungsplans:



Man sieht also hier den echten Brecher mit Kosten von 9556, wobei die Zahl kein absoluter Wert ist. Ein Merge Join deutet häufig auf eine ungünstige Abfrage oder das Fehlen eines Index hin.

Zugriffsprädikate steuern, welche Daten physikalisch eingelesen werden, Filterprädikate steuern, welche davon letztlich übrig bleiben. Beispiel: Die Tabelle callrecord hat Index auf den Zeitstempel (Beginn) und ich möchte wissen, wieviele Gespräche am 01.04.2012 länger als 10 Minuten gedauert haben:

```
SELECT * FROM callrecord c WHERE c.startdate BETWEEN '01.04.2012 00:00:00
AND '01.04.2012 23:59:59'AND duration > (10*60)
```

`startdate` wird dann ein Filterprädikat werden und `duration` ein Zugriffsprädikat.

Best Practices

Im Folgenden eine lose Sammlung von Best Practices, die sich bewährt haben.

UTC-Zeiten

Zeitstempel sollten entweder als UTC-Zeit oder als Datentyp mit Zeitzone definiert werden. Andernfalls gibt's Ärger wenn die erste Niederlassung in einer anderen Zeitzone eröffnet (ja ja, die Globalisierung). Eine Ad-Hoc-Umrechnung bei jedem Zugriff ist absoluter Unsinn (schon gesehen).

Versions-String

Jede Query sollte immer eine Spalte QUERYVERSION enthalten, damit man sehen kann, mit welcher Query-Version die Ergebnisse ermittelt wurden:

```
SELECT …, 'test.sql@V1.27' AS QUERYVERSION FROM …
```

Bei Reports könnte man diese in der Fußzeile ausgeben, bei Programmen (Batches) evtl. über das Logging.

Spalten nicht zu klein wählen
Immer wieder kommt es zu Data-Truncation-Fehlern, weil ein VARCHAR-Feld zu klein gewählt wurde. Selbst erlebt:

```
PHONE_NO VARCHAR(20)
```

Jetzt könnte ich mal schnell bei der ITU nachschlagen, wie lange eine Telefonnummer maximal werden kann, bin aber gerade offline. Ich wollte die Nummer in sprechender Form ablegen:

```
+49 (0) 1234 / 20304 – 199
```

Damit es in HTML dann auch schön dargestellt wird hätte ich das ganze für die URL noch URL-kodieren müssten und schwupps, zu groß :-(. Also Spalten im Zweifelsfall größer machen. Ich hatte schon oft den Fall, dass eine Auftragsnummer dann doch länger wurde als gedacht (das kommt nie vor), die Fahrzeugnummer ist ein schönes Beispiel dafür.

Konsistente Benennungen

Tabellennamen und Spaltennamen sollen möglichst konsistent benannt werden:

- Durchgängige Sprache (Deutsch, Englisch)
- Singular-Form (KUNDE statt KUNDEN)
- Gleiche Spalten sollen in verschiedenen Tabellen auch gleich heißen.
- ID-Spalte heißt xxx_id (`KUNDE.KUNDE_ID`). Referenzspalten heißen xxx_id (`AUFTRAG.KUNDE_ID`). Allerdings führt JPA implizite Namensschemata ein, so dass man sich im Zweifel daran halten sollte (im Falle von ORM gilt eh: vom Programm-Design aus denken, nicht vom der relationalen ERM aus).

Tabellenaliase schaffen Durchblick

Wenn man zwei oder mehr Tabellen joint, sollte man vernünftige Tabellenaliase verwenden:

```
SELECT cn.country_name, c.call_code, sum(c.duration)  AS duration
FROM customer cust, country cn, callrecord c
WHERE …
GROUP BY  cn.country_name, c.call_code
```

Spaltenaliase für Literale und berechnete Spalten

Führt man Literale oder Berechnungen ein (Aggregate, CASE), so sollte man den Spalten unbedingt Alias-Namen geben. So schafft man stabile Verhältnisse nach oben.

```
SELECT 0 AS LOWER_LIMIT, AVG(duration) AS AVG_DURATION FROM …
```

Reports immer auf VIEWs aufbauen

Es gibt Regeln, die bewahrheiten sich einfach immer, auch wenn man sich selbst – trotz besseren Wissens – nicht immer daran hält. Eine davon ist die, dass man für Reports immer erst die passende View erstellen sollte und dann den Report darauf erstellt. Ändert sich später die Datenbank, so muss nur die View angepasst werden. Evtl. kann die View später eine Materialized View oder Tabelle werden, die zyklisch

befüllt wird, dem Report ist das egal. Tabellen und Views werden aus SQL-Sicht (bei der Abfrage) nicht unterschieden.

Vorsicht mit Berechnungen in Views

Gerne werden Berechnungsspalten z. B. mit DECODE in Views eingebaut. Doch Vorsicht, wenn über solche Spalten selektiert werden soll:

```
CREATE VIEW V1 AS SELECT DECODE(statusbyte, 0,'FAIL', 1, 'OK') AS STATE, …
```

Auf den ersten Blick komfortabel, weil die View dann immer sprechende Werte (statt 0/1) anzeigt. Aber wehe, jemand nun V1 nach Status abfragt:

```
SELECT * FROM V1 WHERE STATE = 'OK';
```

Sofern die zugrundeliegende Tabelle viele Einträge hat, kann die Datenbank keinen Index dafür nutzen, da DECODE bei jedem Zugriff ausgeführt werden muss. Eine Alternative sind Function-Based-Indices.

Tools

Datenbank-Tools gibt es viele, je nach Geschmack. Ich persönlich setze DbVizualizer (Free) ein, wenn es um Java/JDBC geht, weil ich so das Verhalten des JDBC-Treibers gut beurteilen kann. Außerdem ist er plattformunabhängig und läuft gegen fast jede Datenbank.

Für Oracle gab es früher den TOAD (gibt es heute wohl noch), aber der kostenlose SQLDeveloper von Oracle lässt bei mir eigentlich keine Wünsche mehr offen.

Thomas Kestler ist Geschäftsführer der elevato GmbH. http://www.elevato.de