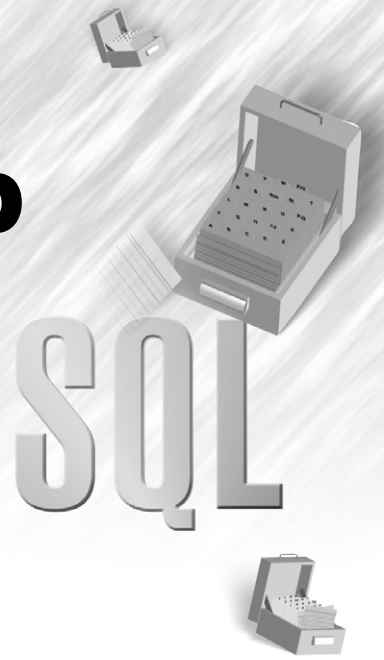

Performanceschub für PostgreSQL

VON SUSANNE EBRECHT, DIPLOM-INFORMATIKERIN (FH)

Das bekannte objektrelationale Datenbankmanagementsystem PostgreSQL ist sehr leistungsstark, sicher und zuverlässig. Dennoch tauchen ab und zu Geschwindigkeitsprobleme auf, denen man aber effizient mit dem EXPLAIN-Statement zu Leibe rücken kann.

Zwei der häufigsten Anwendungsfehler in PostgreSQL, die zu Performanceproblemen führen, sind falsch oder gar nicht gesetzte Indexe sowie eine zu seltene Ausführung des *VACUUM*-Befehls. Seit PostgreSQL 8.1 funktioniert *AUTOVACUUM* zwar sehr zuverlässig, dennoch wird für spezielle Fälle weiterhin empfohlen, regelmäßig, beispielsweise in einem Cronjob, *VACUUM ANALYZE* auszuführen. Sinnvoll ist das auch nach größeren Datenveränderungen in der Datenbank.

Um Performanceprobleme analysieren zu können, ist *EXPLAIN* sehr hilfreich. Das Statement gibt einige Einblicke in die Arbeitsweise des Planers/Optimierers bei der Ausführung einer Abfrage. Es ist anwendbar für die Anweisungen *SELECT, INSERT, UPDATE, DELETE* und *DECLARE ... CURSOR*. Die Syntax:

```
EXPLAIN [ANALYZE] query;
```

EXPLAIN selbst führt keine Query aus, sondern zeigt lediglich den erforderlichen Weg auf. EXPLAIN ANALYZE jedoch führt die Query aus, wodurch weitere Informationen sichtbar werden.

Wie wichtig es ist, den Zeitpunkt der Indexerstellung richtig zu bestimmen, verdeutlichen die folgenden Beispiele. Es wurde hier eine Datenbank erzeugt, die drei Tabellen enthält. Alle Tabellen besitzen zwei Integer-Spalten (id, number). Die erste Tabelle wurde ohne Index angelegt, die zweite enthält einen Primary Key(id), durch den automatisch ein Index angelegt wird, und bei der dritten Tabelle wurde neben dem PRIMARY KEY(id) noch die Spalte *number* mit einem Index belegt:

```
$ createdb testdb
$ psql testdb
testdb=# create table ohneindex(id \
                integer not null, \
                number integer \
                not null);
testdb=# create table einindex(id \
                integer not null, \
                number integer not \
                null, primary key(id));
testdb=# create table zweiindex(id \
                integer not null, \
                number integer not \
                null, primary key(id));
testdb=# create index i_number on
zweiindex(number);
```

Mit Hilfe von *EXPLAIN ANALZE* werden jeweils eine Million Datensätze eingefügt. Die Spalte *id* enthält fortlaufend die Zahlen 1 bis 1000000. Die Spalte *number* enthält je einen Zufallswert zwischen 0 und 100 (siehe Listing 1).

Auf den ersten Blick läßt sich in Listing 1 erkennen, daß die Ausführungszeit (Total runtime) für das Einfügen neuer Datensätze bei Tabellen mit Indexen wesentlich höher ist als bei Tabellen ohne Index. Der Hintergrund ist, daß bei Tabellen mit Indexen zusätzlich zum Einfügen des Datensatz zeitgleich die Informationen der Indexe angepaßt werden. Des-

```
testdb=# explain analyze insert into ohneindex \
                values(generate_series(1,1000000), random()*100);
QUERY PLAN
------------------------------------------------------------------------------
Result  (cost=0.00..0.02 rows=1 width=0) (actual time=0.030..2477.223 rows=1000
000 loops=1)
Total runtime: 7317.660 ms

testdb=# explain analyze insert into einindex values (generate_series(1,1000000
), random()*100);
QUERY PLAN
------------------------------------------------------------------------------
Result  (cost=0.00..0.02 rows=1 width=0) (actual time=0.016..2738.146 rows=1000
000 loops=1)
Total runtime: 15862.169 ms

testdb=# explain analyze insert into zweiindex values (generate_series(1,100000
0), random()*100);
QUERY PLAN
------------------------------------------------------------------------------
Result  (cost=0.00..0.02 rows=1 width=0) (actual time=0.016..2857.797 rows=1000
000 loops=1)
Total runtime: 29090.368 ms
```

Listing 1: Das Einfügen von einer Million Datensätzen

```
testdb=# explain analyze select * from ohneindex where number=42;
QUERY PLAN
------------------------------------------------------------------------------
 Seq Scan on ohneindex  (cost=0.00..17402.00 rows=9548 width=8) (actual time=2.2
41..477.509 rows=10046 loops=1)
   Filter: (number = 42)
 Total runtime: 492.325 ms

testdb=# explain analyze select * from einindex where number=42;
QUERY PLAN
------------------------------------------------------------------------------
 Seq Scan on einindex  (cost=0.00..17402.00 rows=9535 width=8) (actual time=2.44
3..476.107 rows=9956 loops=1)
   Filter: (number = 42)
 Total runtime: 490.718 ms

testdb=# explain analyze select * from zweiindex where number=42;
QUERY PLAN
------------------------------------------------------------------------------
 Bitmap Heap Scan on zweiindex  (cost=62.32..5118.15 rows=9520 width=8) (actual
time=7.405..120.099 rows=9980 loops=1)
   Recheck Cond: (number = 42)
   ->  Bitmap Index Scan on i_number  (cost=0.00..62.32 rows=9520 width=0) (actu
al time=4.045..4.045 rows=9980 loops=1)
         Index Cond: (number = 42)
 Total runtime: 135.301 ms
```

Listing 2: Index und SELECT

halb kommt es bei Tabellen mit Indexen nicht nur bei der Verwendung von *INSERT* zu Perfomanceverlusten, sondern auch bei *DELETE* und *UPDATE*, sofern das Update die Spalten betrifft, die mit dem Index versehen wurden. Ansonsten verhält sich *UPDATE* im Bezug auf Indexe wie *SELECT*.

Kosten

Die Kosten (cost=0.00..0.02) sind eine Schätzung, wie teuer die Operation wird, gemessen in Zeit der Lesezugriffe. Die erste Zahl zeigt an, wie schnell die erste Zeile der Ergebnismenge zurückgegeben werden kann. Mit der zweiten Zahl wird die Dauer der Ausführung der gesamten Operation geschätzt. Die zu erwartende Anzahl der Rückgabezeilen läßt sich im Ausdruck *rows=1* finden. Die anzunehmende Breite einer durchschnittlichen Zeile der Ergebnismenge in Byte wird durch *width=0* ersichtlich. Bei der Anwendung von *EXPLAIN ANALYZE* werden zusätzlich die aktuellen Ausführungskosten angezeigt. Im Beispiel ist unter anderem zu erkennen, daß beim *INSERT*-Statement nur eine Zeile als Ergebnis vom System erwartet wurde, jedoch das Ergebnis aus insgesamt einer Million Zeilen besteht. Der Unterschied ist unbedenklich, da bei einem *INSERT* standardmäßig nur ein Datensatz erwartet wird.

Wie wichtig Indexe bei einem *SELECT* sind, lassen die Beispiele in Listing 2 erkennen.

Da im Beispiel nach der zweiten Spalte *number* gesucht wird und auf dieser Spalte nur in der dritten Tabelle (zweiindex) ein Index liegt, ist das Ergebnis der andern beiden Tabellen gleichwertig. Ohne Index wird die Tabelle sequentiell abgefragt (Seq Scan). Bei der dritten Tabelle ist eindeutig am Index-Scan zu erkennen, daß der Index verwendet wird. Die Ausführungszeit (Total runtime) ist hier um ein Vielfaches niedriger als bei den anderen beiden Tabellen.

Um genauere Analysen zu bekommen, lassen sich die einzelnen Scans abschalten und bestimmte Scans erzwingen. Es ist dabei zu beachten: Wenn alle Scans abgeschaltet sind, wird automatisch mit dem Seq-Scan gearbeitet. Bei kleineren Datenmengen nutzt der Planer/Optimierer häufiger einen Seq-Scan, obwohl ein Index vorliegt (Listing 3).

Das Beispiel zeigt, daß bei nur fünf Datensätze der Planer/Optimierer eine sequentielle Suche bevorzugt. Wird der Seq-Scan abgeschaltet, kommt der Index-Scan zum Einsatz. Jedoch ist zu erkennen, daß der Index-Scan mehr Zeit benötigt als der Seq-Scan. Die Verwendung von passenden Indexen ist aber für *SELECT*-Anfragen generell zu empfehlen, denn auch Tabellen mit kleinen Datenmengen können wachsen. Auch die Arbeitsweise des Planers/Optimierers bei komplexeren *SELECT*-Abfragen läßt sich mit Hilfe von EXPLAIN nachvollziehen wie Listing 4 zeigt. Nicht nur die unterschiedlichen Scans, son-

```
testdb=# create table wenigdaten (id integer not null, \
         number integer not null, primary key(id));
testdb=# create index i_wenigdaten_number on wenigdaten(number);
testdb=# insert into wenigdaten values (generate_series(1,5), random()*2);

testdb=# explain analyze select * from wenigdaten where number=2;
QUERY PLAN
------------------------------------------------------------------------------
 Seq Scan on wenigdaten  (cost=0.00..1.06 rows=2 width=8) (actual time=0.013..0
.018 rows=2 loops=1)
   Filter: (number = 2)
 Total runtime: 0.057 ms

testdb=# set enable_seqscan=off;

testdb=# explain analyze select * from wenigdaten where number=2;
QUERY PLAN
------------------------------------------------------------------------------
 Index Scan using i_wenigdaten_number on wenigdaten  (cost=0.00..4.56 rows=2 wi
dth=8) (actual time=0.022..0.029 rows=2 loops=1)
   Index Cond: (number = 2)
 Total runtime: 0.073 ms
```

Listing 3: Das Verhalten des Planers bei einfachen Abfragen

dern auch die Joins lassen sich abstellen. Sind alle Joins ausgeschaltet, wird Nested Loop angewandt. Vorausgesetzt, es werden im Ergebnis keine NULL-Werte benötigt, empfiehlt es sich, bei der Verknüpfung von Tabellen in einem *SELECT* nicht mit der JOIN-Syntax zu arbeiten, sondern, wie im Beispiel gezeigt, die Tabellen aufzulisten und die Verknüpfungen in der *WHERE*-Bedingungen zu beschreiben (ohne das Wort *JOIN* in der Syntax). Der Planer/Optimierer findet bei dieser Syntax meistens den performancegünstigsten Join.

Sind bei der Ausführung von *EXPLAIN ANALYZE* die Kosten extrem hoch oder die geschätzten Zeilen weichen stark von den tatsächlichen Zeilen ab, ist das in den meisten Fällen ein Zeichen für unaufgeräumte Statistiken. Durch die Ausführung von *VACUUM ANALYZE* ist anschließend das Ergebnis des *EXPLAIN ANALYZE* häufig erheblich besser.

Wie die Beispiele zeigen, lassen sich anhand von *EXPLAIN* Performanceprobleme schnell eingrenzen. Die häufigsten Ursachen sind bei *SELECT* und *UPDATE* fehlende oder falsch gesetzte Indexe sowie nicht optimal performante, manuell zusammengestellte Joins. Wird ein Index nicht erkannt, obwohl er vorhanden ist, hilft das Ausstellen der verwendeten Scans, um zu prüfen, ob der Index erkannt wird. Defekte Indexe sind selten, lassen sich aber nicht ausschließen, hier hilft die Neuerstellung des Index. *VACUUM ANALYZE* sollte regelmäßig ausgeführt werden, um Performanceprobleme zu dämpfen. ◆

```
testdb=# explain analyze select a.id, a.number, b.number, c.number \
                  from ohneindex as a, einindex as b, zweiindex as c \
                    where b.id=a.id and c.id=b.id and c.number=23;
 QUERY PLAN
------------------------------------------------------------------------------
 Hash Join  (cost=27365.84..47363.04 rows=9520 width=16) (actual time=3795.067.
.8574.018 rows=10246 loops=1)
 Hash Cond: ("outer".id = "inner".id)
   ->  Seq Scan on ohneindex a  (cost=0.00..14902.00 rows=1000000 width=8) (act
ual time=2.506..1801.943 rows=1000000 loops=1)
   ->  Hash  (cost=27342.04..27342.04 rows=9520 width=16) (actual time=3791.880
..3791.880 rows=10246 loops=1)
         ->  Merge Join  (cost=5747.27..27342.04 rows=9520 width=16) (actual ti
me=143.373..3768.375 rows=10246 loops=1)
               Merge Cond: ("outer".id = "inner".id)
               ->  Index Scan using einindex_pkey on einindex b  (cost=0.00..18
952.00 rows=1000000 width=8) (actual time=3.771..2013.297 rows=999932 loops=1)
               ->  Sort  (cost=5747.27..5771.07 rows=9520 width=8) (actual
time=138.811..153.718 rows=10246 loops=1)
                     Sort Key: c.id
                     ->  Bitmap Heap Scan on zweiindex c  (cost=62.32..5118.15
rows=9520 width=8) (actual time=6.545..115.086 rows=10246 loops=1)
                           Recheck Cond: (number = 23)
                           ->  Bitmap Index Scan on i_number  (cost=0.00..62.32
 rows=9520 width=0) (actual time=3.796..3.796 rows=10246 loops=1)
                                 Index Cond: (number = 23)
 Total runtime: 8590.739 ms
```

Listing 4: Das Verhalten des Planers bei komplexeren Abfragen