SQL TUNING VERGLEICH

INSY

22. MÄRZ 2015

STEFAN ERCEG
4AHITT

Inhalt

1. Aufgabenstellung 1
2. Problemfälle 2
2.1 SELECT Attributenangabe statt * 2
2.2 Kein HAVING verwenden 4
2.3 Anzahl der Subqueries reduzieren 6
2.4 JOINS statt Subqueries verwenden 8
2.5 UNION statt OR verwenden 9
2.6 Regex-Expression statt SUBSTRING verwenden 10
2.7 BETWEEN statt Größer-/Kleiner-Vergleichen verwenden 12
2.8 EXISTS statt IN verwenden 14
2.9 UNION ALL statt UNION verwenden 16
2.10 Wenig Nicht-Attribute in der WHERE-Klausel verwenden 18
2.11 AND statt || verwenden 19
2.12 Nur die notwendigen Datensätze selektieren 20
3. Quellenangaben 21

1. Aufgabenstellung

Dokumentieren Sie alle Tipps aus den vorgestellten Quellen [1,2] mit ausgeführten Queries aus den zur Verfügung gestellten Testdaten [3] in einem PDF-Dokument. Zeigen Sie jeweils die Kosten der optimierten und nicht-optimierten Variante und diskutieren Sie das Ergebnis.

Eine Herausforderung wäre die Schokofabrik-Datenbank mit den generierten 10000 Datensätzen pro Tabelle zu verwenden, dies ist aber nicht Pflicht, ersetzt aber den Einsatz der oben genannten Testdaten.

[1] http://beginner-sql-tutorial.com/sql-query-tuning.htm
[2] http://beginner-sql-tutorial.com/sql-tutorial-tips.htm
[3] https://elearning.tgm.ac.at/mod/resource/view.php?id=33104
[4] http://www.borko.at/~mike/Testdaten.zip

2. Problemfälle

Die gesamten Problemfälle wurden mit den vom Herr Professor zur Verfügung gestellten Testdaten für die Schokoladefabrik [1] durchgeführt. Dort wurden 10 000 Datensätze pro Tabelle hinzugefügt. Ebenfalls wurden für größere Mengen die von mir generierten Testdaten für die Schokoladefabrik verwendet.

2.1 SELECT Attributenangabe statt *

Auf der einen von Herr Professor Borko vorgestellten Tutorial-Seite [2] wurde beschrieben, dass der SELECT-Befehl schneller erfolgt, wenn die jeweiligen Attribute statt dem * angegeben werden. Dies wird nun überprüft:

2.1.1 kleinere Menge

Die beiden Befehle werden zuerst mit 10 000 Datensätzen verglichen.

1)

```
SELECT firmenname, nummer, datum, status FROM auftrag;
```

→

```
Seq Scan on auftrag  (cost=0.00..188.00 rows=10000 width=36) (actual
time=0.010..1.338 rows=10000 loops=1)
Total runtime: 1.913 ms
Total query runtime: 939 ms.
10000 rows retrieved.
```

2)

```
SELECT * FROM auftrag;
```

→

```
Seq Scan on auftrag  (cost=0.00..188.00 rows=10000 width=36) (actual
time=0.009..1.035 rows=10000 loops=1)
Total runtime: 1.520 ms
Total query runtime: 901 ms.
10000 rows retrieved.
```

Die Ergebnisse bei den 10 000 Datensätze zeigen, dass der SELECT-Befehl mit den angegebenen Attributennamen ca. 0,4 ms länger gedauert hat als der SELECT * - Befehl. Die Endkosten beim SELECT * - Befehl betragen im Gegensatz zum anderen SELECT-Befehl ca. 0,3 weniger.

2.1.2 größere Menge

Nun werden die beiden SELECT-Befehle mit 62 400 Datensätzen verglichen.

1)

```
SELECT fname, auftragsnr, auftragsdtm, aktstatus FROM
produktionsauftrag;
```

→

```
Seq Scan on produktionsauftrag  (cost=0.00..1409.00 rows=62400 width=28)
(actual time=0.006..5.722 rows=62400 loops=1)
Total runtime: 12.229 ms
Total query runtime: 5089 ms.
62400 rows retrieved.
```

2)

```
SELECT * FROM produktionsauftrag;
```

→

```
Seq Scan on produktionsauftrag  (cost=0.00..1409.00 rows=62400 width=28)
(actual time=0.006..13.906 rows=62400 loops=1)
Total runtime: 16.725 ms
Total query runtime: 5216 ms.
62400 rows retrieved.
```

Wie man hier erkennen kann, war der SELECT-Befehl mit den angegebenen Attributennamen um ganze 4,5 ms schneller als der SELECT * - Befehl. Die Endkosten beim SELECT * - Befehl betragen im Gegensatz zum anderen SELECT-Befehl ca. 8 mehr.

➔ Die Aussage, dass der SELECT-Befehl schneller erfolgt, wenn die jeweiligen Attribute statt dem * angegeben werden, stimmt nur bei größeren Mengen! Bei kleineren Mengen merkt man nur einen kleinen Unterschied.

2.2 Kein HAVING verwenden

Auf der Tutorial-Seite [2] wird ebenfalls beschrieben, dass kein HAVING zum Filtern der Datensätze verwendet werden soll, da der SELECT-Befehl beim Einsetzen von HAVING länger dauert. Dies wird nun überprüft, indem ein SELECT-Befehl erstellt wird, bei dem alle Künstler ausgegeben werden, die eine Bekanntheit größer 6 besitzen.

2.2.1 kleinere Menge

Die beiden Befehle werden zuerst mit 10 000 Datensätzen verglichen.

1)

```
SELECT nummer FROM kuenstler
WHERE bekanntheit > 6
GROUP BY nummer;
```

→

```
HashAggregate  (cost=179.88..219.42 rows=3954 width=4) (actual
time=2.675..3.213 rows=3954 loops=1)
   ->  Seq Scan on kuenstler  (cost=0.00..170.00 rows=3954 width=4) (actual
   time=0.015..1.520 rows=3954 loops=1)
         Filter: (bekanntheit > 6)
         Rows Removed by Filter: 6046
Total runtime: 3.458 ms
Total query runtime: 89 ms.
3954 rows retrieved.
```

2)

```
SELECT nummer FROM kuenstler
GROUP BY nummer
HAVING bekanntheit > 6;
```

→

```
HashAggregate  (cost=179.88..219.42 rows=3954 width=4) (actual
time=2.983..3.563 rows=3954 loops=1)
      ->  Seq Scan on kuenstler  (cost=0.00..170.00 rows=3954 width=4)
      (actual time=0.014..1.695 rows=3954 loops=1
        Filter: (bekanntheit > 6)
        Rows Removed by Filter: 6046
Total runtime: 3.831 ms
Total query runtime: 89 ms.
3954 rows retrieved.
```

Die Ergebnisse bei den 10 000 Datensätze zeigen, dass der SELECT-Befehl mit der WHERE-Klausel ca. 0,4 ms kürzer gedauert hat als der SELECT- Befehl mit der Having-Abfrage. Bei den Kosten merkt man, dass die Start- und Endkosten beim SELECT-Befehl mit der WHERE-Klausel weniger betragen als bei der Having-Abfrage.

2.2.2 größere Menge

Dies wird nun ebenfalls mit 100 000 Datensätzen verglichen.

1)

```
SELECT persid FROM kuenstler
WHERE bekanntheitsgrad > 6
GROUP BY persid;
```

→

```
Group  (cost=0.29..3389.86 rows=35827 width=4) (actual time=0.023..71.645
rows=36275 loops=1)
->  Index Scan using kuenstler_pkey on kuenstler  (cost=0.29..3300.29
rows=35827 width=4) (actual time=0.021..48.497 rows=36275 loops=1)
        Filter: (bekanntheitsgrad > 6)
Total runtime: 73.422 ms
        Rows Removed by Filter: 63725
Total query runtime: 809 ms.
36275 rows retrieved.
```

2)

```
SELECT persid FROM kuenstler
GROUP BY persid
HAVING bekanntheitsgrad > 6;
```

→

```
Group  (cost=0.29..3389.86 rows=35827 width=4) (actual time=0.043..69.839
rows=36275 loops=1)
->  Index Scan using kuenstler_pkey on kuenstler  (cost=0.29..3300.29
rows=35827 width=4) (actual time=0.039..47.907 rows=36275 loops=1)
        Filter: (bekanntheitsgrad > 6)
        Rows Removed by Filter: 63725
Total runtime: 75.862 ms
Total query runtime: 823 ms.
36275 rows retrieved.
```

Bei der Analyse von diesen beiden Befehlen kann man nun erkennen, dass die Endkosten bei dem SELECT-Befehl mit der WHERE-Klausel ca. 2 mehr betragen als bei der Having-Abfrage. Trotzdem war der Befehl mit der WHERE-Klausel um ganze 2 ms schneller.

➔ Die Aussage, dass der Befehl schneller ist, wenn kein HAVING zum Filtern der Datensätze verwendet wird, stimmt!

2.3 Anzahl der Subqueries reduzieren

Weiteres wird auf der Tutorial-Seite [2] beschrieben, dass man die Anzahl der Subqueries beim SELECT-Befehl reduzieren sollte. Dies wird nun überprüft, indem ein SELECT-Befehl erstellt wird, bei dem jede Kunstschau, die in einem Ort bzw. Land auftritt, dessen Name mit 3 endet, ausgegeben wird.

Die beiden Befehle werden mit 10 000 Datensätzen verglichen.

1)

```
SELECT * FROM kunstschau
WHERE (ort, land) IN (SELECT ort, land FROM kunstschau WHERE
ort LIKE '%3' AND land LIKE '%3');
```

→

```
Hash Semi Join  (cost=244.31..489.60 rows=29 width=41) (actual time=3.604..15.908
rows=73 loops=1)
  Hash Cond: ((kunstschau.ort = kunstschau_1.ort) AND (kunstschau.land =
kunstschau_1.land))
  ->  Seq Scan on kunstschau  (cost=0.00..192.00 rows=10000 width=41) (actual
time=0.009..1.286 rows=10000 loops=1)
  ->  Hash  (cost=242.00..242.00 rows=154 width=24) (actual time=3.524..3.524
rows=73 loops=1)
        Buckets: 1024  Batches: 1  Memory Usage: 5kB
        ->  Seq Scan on kunstschau kunstschau_1  (cost=0.00..242.00 rows=154
width=24) (actual time=0.035..3.492 rows=73 loops=1)
              Filter: ((ort ~~ '%3'::text) AND (land ~~ '%3'::text))
              Rows Removed by Filter: 9927
Total runtime: 15.949 ms
Total query runtime: 24 ms.
73 rows retrieved.
```

2)

```
SELECT * FROM kunstschau
WHERE ort IN (SELECT ort FROM kunstschau WHERE ort LIKE '%3')
AND land IN (SELECT land FROM kunstschau WHERE land LIKE
'%3');
```

→

```
Hash Semi Join  (cost=470.00..727.94 rows=502 width=41) (actual time=12.067..18.764
rows=73 loops=1)
  Hash Cond: (kunstschau.land = kunstschau_2.land)
  ->  Hash Join  (cost=237.05..479.32 rows=2167 width=41) (actual time=2.552..9.126
rows=678 loops=1)
        Hash Cond: (kunstschau.ort = kunstschau_1.ort)
        ->  Seq Scan on kunstschau  (cost=0.00..192.00 rows=10000 width=41) (actual
time=0.004..1.100 rows=10000 loops=1)
        ->  Hash  (cost=227.59..227.59 rows=757 width=12) (actual time=2.512..2.512
rows=427 loops=1)
              Buckets: 1024  Batches: 1  Memory Usage: 19kB
              ->  HashAggregate  (cost=220.02..227.59 rows=757 width=12) (actual
time=2.328..2.392 rows=427 loops=1)
                    ->  Seq Scan on kunstschau kunstschau_1  (cost=0.00..217.00
rows=1208 width=12) (actual time=0.015..2.117 rows=678 loops=1)
                          Filter: (ort ~~ '%3'::text)
                          Rows Removed by Filter: 9322
  ->  Hash  (cost=217.00..217.00 rows=1276 width=12) (actual time=9.470..9.470
rows=809 loops=1)
        Buckets: 1024  Batches: 1  Memory Usage: 36kB
        ->  Seq Scan on kunstschau kunstschau_2  (cost=0.00..217.00 rows=1276
width=12) (actual time=0.015..9.228 rows=809 loops=1)
              Filter: (land ~~ '%3'::text)
              Rows Removed by Filter: 9191
Total runtime: 18.830 ms
Total query runtime: 24 ms.
73 rows retrieved.
```

Wie man hier erkennen kann, ist der Befehl mit den reduzierten Subqueries um ca. 3 ms schneller. Vor allem an den Startkosten merkt man, dass der Befehl mit den reduzierten Subqueries schneller war (um ca. 8,5 weniger).

➔ Die Aussage, dass der Befehl schneller ist, wenn man die Anzahl der Subqueries beim SELECT-Befehl reduziert, stimmt!

2.4 JOINS statt Subqueries verwenden

Auf der zweiten Tutorial-Seite [3] wird beschrieben, dass die Verwendung von JOINS anstatt von Subqueries die Ausführung wesentlich schneller macht. Dies wird nun überprüft, indem ein SELECT-Befehl erstellt wird, bei dem zusätzlich zu der ID eines Produkts der maximale Preis von diesem ausgegeben wird.

Die beiden Befehle werden mit 10 000 Datensätzen verglichen.

1)

```
SELECT p.nummer, MAX(s.preis) AS maxPreis
FROM produkt p
INNER JOIN standardsortiment s ON p.nummer = s.nummer
GROUP BY p.nummer;
```

→

```
GroupAggregate  (cost=0.57..991.57 rows=10000 width=8) (actual time=0.043..31.621 rows=10000
loops=1)
  ->  Merge Join  (cost=0.57..841.57 rows=10000 width=8) (actual time=0.029..18.544 rows=10000
loops=1)
        Merge Cond: (p.nummer = s.nummer)
        ->  Index Only Scan using produkt_pkey on produkt p  (cost=0.29..347.29 rows=10000
width=4) (actual time=0.017..7.099 rows=10000 loops=1)
              Heap Fetches: 10000
        ->  Index Scan using standardsortiment_pkey on standardsortiment s  (cost=0.29..344.29
rows=10000 width=8) (actual time=0.008..6.785 rows=10000 loops=1)
Total runtime: 32.176 ms
Total query runtime: 307 ms.
10000 rows retrieved.
```

2)

```
SELECT p.nummer, (SELECT MAX(s.preis) FROM standardsortiment s
WHERE nummer = p.nummer) AS maxPreis
FROM produkt p;
```

→

```
Seq Scan on produkt p  (cost=0.00..83324.00 rows=10000 width=4) (actual time=0.031..66.915
rows=10000 loops=1)
  SubPlan 1
    ->  Aggregate  (cost=8.30..8.31 rows=1 width=4) (actual time=0.004..0.005 rows=1
loops=10000)
          ->  Index Scan using standardsortiment_pkey on standardsortiment s  (cost=0.29..8.30
rows=1 width=4) (actual time=0.003..0.003 rows=1 loops=10000)
                Index Cond: (nummer = p.nummer)
Total runtime: 67.713 ms
Total query runtime: 313 ms.
10000 rows retrieved.
```

Die Ergebnisse zeigen, dass der Befehl mit dem INNER JOIN um ca. 35 ms schneller war als der Befehl mit der Subquery. Die Endkosten beim Befehl mit dem INNER JOIN betragen ebenfalls ca. 35 weniger.

➔ Die Aussage, dass der Befehl schneller ist, wenn man beim SELECT-Befehl JOINS statt Subqueries verwendet, stimmt!

2.5 UNION statt OR verwenden

Auf der zweiten Tutorial-Seite [3] wird ebenfalls beschrieben, dass die Verwendung von UNION anstatt von OR die Ausführung wesentlich schneller macht. Dies wird nun überprüft, indem ein SELECT-Befehl erstellt wird, bei dem alle Personen ausgegeben werden, deren Vorname mit „Nan“ oder deren Nachname mit „Con“ beginnt.

Die beiden Befehle werden mit 10 000 Datensätzen verglichen.

1)

```
SELECT * FROM person WHERE vorname LIKE 'Nan%'
UNION
SELECT * FROM person WHERE nachname LIKE 'Con%';
```

→

```
HashAggregate  (cost=377.70..378.67 rows=97 width=17) (actual time=3.226..3.238
rows=51 loops=1)
  ->  Append  (cost=0.00..376.97 rows=97 width=17) (actual time=0.014..3.187
rows=52 loops=1)
              Filter: (vorname ~~ 'Nan%'::text)
        ->  Seq Scan on person  (cost=0.00..188.00 rows=96 width=17) (actual
time=0.013..1.502 rows=27 loops=1)
              Rows Removed by Filter: 9973
        ->  Seq Scan on person person_1  (cost=0.00..188.00 rows=1 width=17)
(actual time=0.005..1.673 rows=25 loops=1)
              Filter: (nachname ~~ 'Con%'::text)
              Rows Removed by Filter: 9975
Total runtime: 3.291 ms
Total query runtime: 13 ms.
51 rows retrieved.
```

2)

```
SELECT * FROM person
WHERE vorname LIKE 'Nan%' OR nachname LIKE 'Con%';
```

→

```
Seq Scan on person  (cost=0.00..213.00 rows=97 width=17) (actual time=0.015..2.365
rows=51 loops=1)
  Filter: ((vorname ~~ 'Nan%'::text) OR (nachname ~~ 'Con%'::text))
  Rows Removed by Filter: 9949
Total runtime: 2.389 ms
Total query runtime: 13 ms.
51 rows retrieved.
```

Bei diesem Vergleich merkt man, dass der SELECT-Befehl mit OR um ca. 1,1 ms schneller war als der SELECT-Befehl mit UNION. Die Startkosten betragen beim SELECT-Befehl mit UNION ca. 3,2 mehr als beim Befehl mit OR, die Endkosten ca. 0,9 mehr.

➔ Die Aussage, dass der Befehl schneller ist, wenn man beim SELECT-Befehl UNION statt OR verwendet, stimmt nicht! Der Befehl mit OR war um einiges schneller.

2.6 Regex-Expression statt SUBSTRING verwenden

Es wird auf der Tutorial-Seite [2] ebenfalls beschrieben, dass der SELECT-Befehl schneller erfolgt, wenn man für eine Suche statt einem Substring eine Regex-Expression verwendet. Dies wird überprüft, indem alle Personen ausgegeben werden, dessen Vorname mit ‚Al' beginnt.

2.6.1 kleinere Menge

Die beiden Befehle werden zuerst mit 10 000 Datensätzen verglichen.

1)

```
SELECT nummer, vorname, nachname
FROM person
WHERE vorname LIKE 'Al%';
```

→

```
Seq Scan on person  (cost=0.00..188.00 rows=211 width=17) (actual
time=0.018..1.553 rows=141 loops=1)
  Filter: (vorname ~~ 'Al%'::text)
  Rows Removed by Filter: 9859
Total runtime: 1.586 ms
Total query runtime: 13 ms.
141 rows retrieved.
```

2)

```
SELECT nummer, vorname, nachname
FROM person
WHERE SUBSTR(vorname,1,2) = 'Al';
```

→

```
Seq Scan on person  (cost=0.00..213.00 rows=50 width=17) (actual
time=0.023..2.930 rows=141 loops=1)
  Filter: (substr(vorname, 1, 2) = 'Al'::text)
  Rows Removed by Filter: 9859
Total runtime: 2.962 ms
Total query runtime: 27 ms.
141 rows retrieved.
```

Die Ergebnisse bei den 10 000 Datensätze zeigen, dass der SELECT-Befehl mit der Regex-Expression ca. 1,4 ms kürzer gedauert hat als der SELECT-Befehl, wo ein Substring verwendet wird. Bei den Kosten merkt man, dass die Start- und Endkosten beim SELECT-Befehl mit der Regex-Expression (bei den Endkosten sogar ca. 1,4) weniger betragen als bei der Substring-Abfrage.

2.6.2 größere Menge

Die beiden Befehle werden jetzt ebenfalls mit der größeren Datenbank verglichen. Die Tabelle Person besitzt 220 000 Datensätze.

1)

```
SELECT persid, vname, nname
FROM person
WHERE vname LIKE 'Al%';
```

→

```
Seq Scan on person  (cost=0.00..4098.00 rows=4721 width=17) (actual
time=0.018..71.223 rows=4480 loops=1)
  Filter: ((vname)::text ~~ 'Al%'::text)
  Rows Removed by Filter: 215520
Total runtime: 71.491 ms
Total query runtime: 217 ms.
4480 rows retrieved.
```

2)

```
SELECT persid, vname, nname
FROM person
WHERE SUBSTR(vname,1,2) = 'Al';
```

→

```
Seq Scan on person  (cost=0.00..4648.00 rows=1100 width=17) (actual
time=0.026..137.117 rows=4480 loops=1)
  Filter: (substr((vname)::text, 1, 2) = 'Al'::text)
  Rows Removed by Filter: 215520
Total runtime: 137.413 ms
Total query runtime: 221 ms.
4480 rows retrieved.
```

Bei dieser Analyse merkt man, dass der SELECT-Befehl mit der Regex-Expression um ganze 66 ms schneller war als der SELECT-Befehl mit dem SUBSTRING. Die Endkosten vom SELECT-Befehl mit der Regex-Expression betragen dementsprechend ebenfalls um 66 weniger.

➔ Die Aussage, dass der Befehl schneller ist, wenn eine Regex-Expression statt einem Substring verwendet wird, stimmt!

2.7 BETWEEN statt Größer-/Kleiner-Vergleichen verwenden

Es wird auf der Tutorial-Seite [2] ebenfalls beschrieben, dass der SELECT-Befehl schneller erfolgt, wenn man für eine Suche statt mehreren Größer-/Kleiner-Vergleichen den Befehl BETWEEN verwendet.

2.7.1 kleinere Menge

Die beiden Befehle werden zuerst mit 10 000 Datensätzen verglichen. Hier werden alle Standardsortimente ausgegeben, deren Preis zwischen 0,3 und 0,6 liegt.

1)

```
SELECT nummer, preis
FROM standardsortiment
WHERE preis BETWEEN 0.3 AND 0.6;
```

→

```
Seq Scan on standardsortiment  (cost=0.00..221.00 rows=3078 width=8)
(actual time=0.017..3.639 rows=3118 loops=1)
  Filter: ((preis >= 0.3) AND (preis <= 0.6))
  Rows Removed by Filter: 6882
Total runtime: 3.870 ms
Total query runtime: 101 ms.
3118 rows retrieved.
```

2)

```
SELECT nummer, preis
FROM standardsortiment
WHERE preis >= 0.3 AND preis <= 0.6;
```

→

```
Seq Scan on standardsortiment  (cost=0.00..221.00 rows=3078 width=8)
(actual time=0.017..3.434 rows=3118 loops=1)
  Filter: ((preis >= 0.3) AND (preis <= 0.6))
  Rows Removed by Filter: 6882
Total runtime: 3.583 ms
Total query runtime: 108 ms.
3118 rows retrieved.
```

Bei dem Vergleich von diesen beiden Befehlen merkt man, dass sie von der Zeit her ziemlich gleich lang gedauert haben, allerdings war der Befehl mit den Größer-/Kleiner-Vergleichen um ca. 0,3 ms schneller. Die Endkosten beim Befehl mit den Größer-/Kleiner-Vergleichen betragen um ca. 0,2 weniger als der Befehl mit dem BETWEEN-Befehl.

2.7.2. größere Menge

Die beiden Befehle werden nun mit der größeren Tabelle vom Standardsortiment verglichen, in der sich 100 000 Datensätze befinden. Hier werden alle Standardsortimente ausgegeben, deren Preis zwischen 5 und 8 liegt.

1)

```
SELECT prid, verkaufspreis
FROM standardsortiment
WHERE verkaufspreis BETWEEN 5 AND 8;
```

→

```
Seq Scan on standardsortiment  (cost=0.00..2196.00 rows=37332 width=12)
(actual time=0.015..41.840 rows=37347 loops=1)
  Filter: ((verkaufspreis >= 5::double precision) AND (verkaufspreis <=
8::double precision))
  Rows Removed by Filter: 62653
Total runtime: 52.634 ms
Total query runtime: 1362 ms.
37347 rows retrieved.
```

2)

```
SELECT prid, verkaufspreis
FROM standardsortiment
WHERE verkaufspreis >= 5 AND verkaufspreis <= 8;
```

→

```
Seq Scan on standardsortiment  (cost=0.00..2196.00 rows=37332 width=12)
(actual time=0.018..51.891 rows=37347 loops=1)
  Filter: ((verkaufspreis >= 5::double precision) AND (verkaufspreis <=
8::double precision))
  Rows Removed by Filter: 62653
Total runtime: 53.607 ms
Total query runtime: 1289 ms.
37347 rows retrieved.
```

Auch bei dieser Analyse merkt man, dass die beiden Befehle ziemlich gleich lang dauern. Der Befehl mit dem BETWEEN-Befehl ist nur ca. 1 ms schneller, die Endkosten betragen jedoch um ca. 10 weniger als beim Befehl mit den Größer-/Kleiner-Vergleichen.

➔ Die Aussage, dass der Befehl schneller ist, wenn der BETWEEN-Befehl statt den Größer-/Kleiner-Vergleichen verwendet wird, stimmt nicht ganz! Wenn man beide Befehle vergleicht gibt es nur geringfügige Unterschiede.

2.8 EXISTS statt IN verwenden

Auf der Tutorial-Seite [2] wird beschrieben, dass ein SELECT-Befehl schneller erfolgt, wenn EXISTS statt IN verwendet wird, da IN die langsamste Performance besitzt. Dies wird überprüft, indem ein SELECT-Befehl erstellt wird, bei dem alle Personen ausgegeben werden, die Künstler sind.

2.8.1 kleinere Menge

Die beiden Befehle werden zuerst mit 10 000 Datensätzen verglichen.

1)

```
SELECT * FROM person p
WHERE EXISTS (SELECT * FROM kuenstler k WHERE
p.nummer=k.nummer);
```

→

```
Hash Semi Join  (cost=270.00..683.00 rows=10000 width=17) (actual
time=3.752..17.760 rows=10000 loops=1)
  Hash Cond: (p.nummer = k.nummer)
  ->  Seq Scan on person p  (cost=0.00..163.00 rows=10000 width=17) (actual
time=0.006..8.901 rows=10000 loops=1)
  ->  Hash  (cost=145.00..145.00 rows=10000 width=4) (actual time=3.727..3.727
rows=10000 loops=1)
        Buckets: 1024  Batches: 1  Memory Usage: 352kB
        ->  Seq Scan on kuenstler k  (cost=0.00..145.00 rows=10000 width=4) (actual
time=0.005..1.766 rows=10000 loops=1)
Total runtime: 18.349 ms
Total query runtime: 493 ms.
10000 rows retrieved.
```

2)

```
SELECT * FROM person
WHERE nummer IN (SELECT nummer FROM kuenstler);
```

→

```
Hash Semi Join  (cost=270.00..683.00 rows=10000 width=17) (actual
time=7.032..18.024 rows=10000 loops=1)
  Hash Cond: (person.nummer = kuenstler.nummer)
  ->  Seq Scan on person  (cost=0.00..163.00 rows=10000 width=17) (actual
time=0.007..1.281 rows=10000 loops=1)
  ->  Hash  (cost=145.00..145.00 rows=10000 width=4) (actual time=7.010..7.010
rows=10000 loops=1)
        Buckets: 1024  Batches: 1  Memory Usage: 352kB
        ->  Seq Scan on kuenstler  (cost=0.00..145.00 rows=10000 width=4) (actual
time=0.006..5.034 rows=10000 loops=1)
Total runtime: 18.635 ms
Total query runtime: 505 ms.
10000 rows retrieved.
```

Bei diesem Vergleich kann man vor allem erkennen, dass die Startkosten beim Befehl mit IN um ca. 3,3 mehr betragen als beim Befehl mit EXISTS. Der Befehl mit IN ist ebenfalls um ca. 0,3 ms schneller.

2.8.2 größere Menge

Bei der Personen Tabelle existieren hier 220 000 Datensätze, 100 000 davon sind Künstler.

1)

```
SELECT * FROM person p
WHERE EXISTS (SELECT * FROM kuenstler k WHERE
p.persid=k.persid);
```

→

```
Merge Semi Join  (cost=0.71..7751.74 rows=100000 width=17) (actual
time=0.034..231.261 rows=100000 loops=1)
  Merge Cond: (p.persid = k.persid)
  ->  Index Scan using person_pkey on person p  (cost=0.42..7071.42 rows=220000
width=17) (actual time=0.015..60.992 rows=100001 loops=1)
  ->  Index Only Scan using kuenstler_pkey on kuenstler k  (cost=0.29..3050.29
rows=100000 width=4) (actual time=0.011..83.275 rows=100000 loops=1)
        Heap Fetches: 100000
Total runtime: 235.954 ms
Total query runtime: 4914 ms.
100000 rows retrieved.
```

2)

```
SELECT * FROM person
WHERE persid IN (SELECT persid FROM kuenstler);
```

→

```
Merge Semi Join  (cost=0.71..7751.74 rows=100000 width=17) (actual
time=0.022..224.076 rows=100000 loops=1)
  Merge Cond: (person.persid = kuenstler.persid)
  ->  Index Scan using person_pkey on person  (cost=0.42..7071.42 rows=220000
width=17) (actual time=0.009..50.948 rows=100001 loops=1)
  ->  Index Only Scan using kuenstler_pkey on kuenstler  (cost=0.29..3050.29
rows=100000 width=4) (actual time=0.010..69.274 rows=100000 loops=1)
        Heap Fetches: 100000
Total runtime: 234.339 ms
Total query runtime: 4847 ms.
100000 rows retrieved.
```

Beide Befehle dauerten hier ziemlich genau gleich lang, der Befehl mit IN war um ca. 1,5 ms schneller. Die Endkosten beim Befehl mit IN betragen ca. 7 weniger als beim Befehl mit EXISTS.

➔ Die Aussage, dass der Befehl schneller ist, wenn der EXISTS-Befehl statt dem IN-Befehl verwendet wird, stimmt nicht ganz! Wenn man beide Befehle vergleicht gibt es nur geringfügige Unterschiede.

2.9 UNION ALL statt UNION verwenden

Auf der Tutorial-Seite [2] wird beschrieben, dass UNION ALL statt UNION für einen schnelleren SELECT-Befehl verwendet werden soll. UNION wird verwendet, um mehrere SELECT-Statements zu einer Ausgabe zusammenzufassen. Bei UNION werden im Gegensatz zu UNION ALL doppelte Datensätze herausgefiltert. Der Vergleich wird nun durch Kombinieren der Tabellen Person und Künstler erstellt.

2.9.1 kleinere Menge

Die beiden Befehle werden zuerst mit 10 000 Datensätzen verglichen.

1)

```
SELECT nummer FROM person
UNION ALL
SELECT nummer FROM kuenstler;
```

→

```
Append  (cost=0.00..308.00 rows=20000 width=4) (actual time=0.012..5.111 rows=20000
loops=1)
  ->  Seq Scan on person  (cost=0.00..163.00 rows=10000 width=4) (actual
time=0.011..1.571 rows=10000 loops=1)
  ->  Seq Scan on kuenstler  (cost=0.00..145.00 rows=10000 width=4) (actual
time=0.006..1.565 rows=10000 loops=1)
Total runtime: 5.989 ms
Total query runtime: 426 ms.
20000 rows retrieved.
```

2)

```
SELECT nummer FROM person
UNION
SELECT nummer FROM kuenstler;
```

→

```
HashAggregate  (cost=558.00..758.00 rows=20000 width=4) (actual time=30.059..40.559
rows=10000 loops=1)
  ->  Append  (cost=0.00..508.00 rows=20000 width=4) (actual time=0.011..9.526
rows=20000 loops=1)
        ->  Seq Scan on person  (cost=0.00..163.00 rows=10000 width=4) (actual
time=0.011..2.132 rows=10000 loops=1)
        ->  Seq Scan on kuenstler  (cost=0.00..145.00 rows=10000 width=4) (actual
time=0.007..5.027 rows=10000 loops=1)
Total runtime: 50.677 ms
Total query runtime: 245 ms.
10000 rows retrieved.
```

Hier kann man erkennen, dass der Befehl mit UNION ALL deutlich kürzer gedauert hat als der Befehl mit UNION (Differenz von ca. 45 ms!). Sowohl die Start-, als auch die Endkosten betragen beim Befehl mit UNION deutlich mehr (ca. 30 – 35) als beim Befehl mit UNION ALL.

2.9.2 größere Menge

Hier besitzt die Tabelle Person 220 000 Datensätze und die Tabelle Künstler 100 000 Datensätze. Bei UNION ALL werden somit 320 000 Datensätze zurückgeliefert.

1)

```
SELECT persid FROM person
UNION ALL
SELECT persid FROM kuenstler;
```

→

```
Append  (cost=0.00..4991.00 rows=320000 width=4) (actual time=0.012..181.309
rows=320000 loops=1)
  ->  Seq Scan on person  (cost=0.00..3548.00 rows=220000 width=4) (actual
time=0.011..86.618 rows=220000 loops=1)
  ->  Seq Scan on kuenstler  (cost=0.00..1443.00 rows=100000 width=4) (actual
time=0.006..36.514 rows=100000 loops=1)
Total runtime: 201.007 ms
Total query runtime: 6799 ms.
320000 rows retrieved.
```

2)

```
SELECT persid FROM person
UNION
SELECT persid FROM kuenstler;
```

→

```
Unique  (cost=41826.34..43426.34 rows=320000 width=4) (actual time=586.170..814.773
rows=220000 loops=1)
  ->  Sort  (cost=41826.34..42626.34 rows=320000 width=4) (actual
time=586.168..692.782 rows=320000 loops=1)
        Sort Key: person.persid
        Sort Method: external merge  Disk: 4368kB
        ->  Append  (cost=0.00..8191.00 rows=320000 width=4) (actual
time=0.011..165.815 rows=320000 loops=1)
              ->  Seq Scan on person  (cost=0.00..3548.00 rows=220000 width=4)
(actual time=0.011..86.282 rows=220000 loops=1)
              ->  Seq Scan on kuenstler  (cost=0.00..1443.00 rows=100000 width=4)
(actual time=0.006..23.885 rows=100000 loops=1)
Total runtime: 853.557 ms
Total query runtime: 5564 ms.
220000 rows retrieved.
```

Auch hier kann man sehr deutlich erkennen, dass der Befehl mit UNION ALL viel schneller ist als der Befehl mit UNION (ganze 652 ms!). Die Start- und Endkosten beim Befehl mit UNION betragen ca. 580 bis 650 weniger als der Befehl mit UNION ALL.

➔ Die Aussage, dass der Befehl schneller ist, wenn der UNION ALL-Befehl statt dem UNION-Befehl verwendet wird, stimmt eindeutig! Es existieren große Unterschiede von der Performance her.

2.10 Wenig Nicht-Attribute in der WHERE-Klausel verwenden

Auf der Tutorial-Seite [2] wird beschrieben, dass ein SELECT-Befehl schneller erfolgt, wenn wenig bis gar keine Nicht-Attribute, d.h. geschriebene Werte, wie z.B. 10 000, in der WHERE-Klausel verwendet werden. Hier wird dies überprüft, indem ich bei einem SELECT-Befehl alle Künstler, deren Bekanntheitsgrad unter 6 liegt, ausgebe und bei einem weiteren SELECT-Befehl dem Bekanntheitsgrad 10 hinzurechne und überprüfe, welche Künstler einen Bekanntheitsgrad unter 16 besitzen.

Die beiden Befehle werden mit 10 000 Datensätzen verglichen.

1)

```
SELECT nummer, bekanntheit FROM kuenstler
WHERE bekanntheit < 6;
```

→

```
Seq Scan on kuenstler  (cost=0.00..170.00 rows=5033 width=8) (actual
time=0.015..1.524 rows=5033 loops=1)
  Filter: (bekanntheit < 6)
  Rows Removed by Filter: 4967
Total runtime: 1.767 ms
Total query runtime: 141 ms.
5033 rows retrieved.
```

2)

```
SELECT nummer, bekanntheit FROM kuenstler
WHERE bekanntheit + 10 < 16;
```

→

```
Seq Scan on kuenstler  (cost=0.00..195.00 rows=3333 width=8) (actual
time=0.015..2.152 rows=5033 loops=1)
  Filter: ((bekanntheit + 10) < 16)
  Rows Removed by Filter: 4967
Total runtime: 2.383 ms
Total query runtime: 143 ms.
5033 rows retrieved.
```

Die Ergebnisse bei den 10 000 Datensätzen zeigen, dass der SELECT-Befehl, wo kein expliziter Wert in der WHERE-Klausel angegeben wird, um ca. 0,6 ms schneller ist. Ebenfalls betragen die Endkosten ca. 0,6 weniger.

➔ Die Aussage, dass der Befehl schneller ist, wenn wenig bis gar keine Nicht-Attribute in der WHERE-Klausel verwendet werden, stimmt!

2.11 AND statt || verwenden

Auf der Tutorial-Seite [2] wird beschrieben, dass ein SELECT-Befehl schneller erfolgt, wenn der Operator AND statt || verwendet wird. Dies wird überprüft, indem ein SELECT-Befehl erstellt wird, bei dem die Person ausgegeben wird, die mit Vornamen Willis und mit Nachnamen Eisler heißt.

Die beiden Befehle werden mit 10 000 Datensätzen verglichen.

1)

```
SELECT nummer, vorname, nachname
FROM person
WHERE vorname='Willis' AND nachname='Eisler';
```

→

```
Seq Scan on person  (cost=0.00..213.00 rows=1 width=17) (actual
time=1.422..1.423 rows=1 loops=1)
  Filter: ((vorname = 'Willis'::text) AND (nachname = 'Eisler'::text))
  Rows Removed by Filter: 9999
Total runtime: 1.454 ms
Total query runtime: 13 ms.
1 row retrieved.
```

2)

```
SELECT nummer, vorname, nachname
FROM person
WHERE vorname || nachname='WillisEisler';
```

→

```
Seq Scan on person  (cost=0.00..213.00 rows=50 width=17) (actual
time=2.639..2.642 rows=1 loops=1)
  Filter: ((vorname || nachname) = 'WillisEisler'::text)
  Rows Removed by Filter: 9999
Total runtime: 2.670 ms
Total query runtime: 12 ms.
1 row retrieved.
```

Wie man hier erkennen kann, dauert der SELECT-Befehl mit dem verwendeten AND-Operator ca. 1,2 ms kürzer als der Befehl mit dem verwendeten ||-Operator. Die Endkosten beim SELECT-Befehl mit dem verwendeten AND-Operator betragen daher ca. 1,2 weniger.

➔ Die Aussage, dass der Befehl schneller ist, wenn ein AND-Operator statt einem ||-Operator verwendet wird, stimmt!

2.12 Nur die notwendigen Datensätze selektieren

Auf der zweiten Tutorial-Seite [3] wird beschrieben, dass man sich, bevor man ein SELECT-Befehl durchführt, überlegen sollte, welche Datensätze man überhaupt als Ergebnis zurückbekommen möchte. Dies wird hier veranschaulicht, indem ich von der Tabelle „Auftrag“ bei einem Befehl nur die Aufträge ausgebe, die den Status „in Produktion“ besitzen und bei einem weiteren Befehl die Aufträge ausgebe, die den Status „in Produktion“ oder „abgeschlossen“ besitzen.

Die beiden Befehle werden mit 10 000 Datensätzen verglichen.

1)

```
SELECT * FROM auftrag
WHERE status = 'in Produktion';
```

→

```
Seq Scan on auftrag  (cost=0.00..213.00 rows=2517 width=36) (actual
time=0.013..1.986 rows=2517 loops=1)
  Filter: (status = 'in Produktion'::text)
Total runtime: 2.117 ms
  Rows Removed by Filter: 7483
Total query runtime: 239 ms.
2517 rows retrieved.
```

2)

```
SELECT * FROM auftrag
WHERE status = 'in Produktion' OR status = 'abgeschlossen';
```

→

```
Seq Scan on auftrag  (cost=0.00..238.00 rows=4409 width=36) (actual
time=0.012..2.434 rows=5046 loops=1)
  Filter: ((status = 'in Produktion'::text) OR (status =
'abgeschlossen'::text))
  Rows Removed by Filter: 4954
Total runtime: 2.686 ms
Total query runtime: 431 ms.
5046 rows retrieved.
```

Logischerweise kann man hier nun erkennen, dass der Befehl, der nur Aufträge ausgegeben hat, die als Status „in Produktion“ besitzen, um ca. 0,5 ms kürzer gedauert hat als der Befehl, der zusätzlich auch die Aufträge ausgegeben hat, die als Status „abgeschlossen“ besitzen.

➔ Die Aussage, dass der Befehl schneller ist, wenn nur die notwendigen Datensätze selektiert werden, stimmt logischerweise!

3. Quellenangaben

[1] Michael Borko (2015). Testdaten für die Schokofabrik [Online].
Available at: http://www.borko.at/~mike/Testdaten.zip
[zuletzt abgerufen am 18.03.2015]

[2] Beginner-Sql-Tutorial.com (2007, 2015). SQL Tuning or SQL Optimization [Online].
Available at: http://beginner-sql-tutorial.com/sql-query-tuning.htm
[zuletzt abgerufen am 22.03.2015]

[3] Beginner-Sql-Tutorial.com (2007, 2015). Beginner SQL Tutorial Tips [Online].
Available at: http://beginner-sql-tutorial.com/sql-tutorial-tips.htm
[zuletzt abgerufen am 22.03.2015]

[4] The PostgreSQL Global Development Group (1996, 2015).
Chapter 7.4. Combining Queries [Online].
Available at: http://www.postgresql.org/docs/9.4/static/queries-union.html
[zuletzt abgerufen am 22.03.2015]