JUST DO IT
HÖHERE ABTEILUNG FÜR INFORMATIONSTECHNOLOGIE

tgm
Die Schule der Technik

Protokoll

SQL - Tuning

INSY
4AHITM 2015/16

Benedikt Berger
Matthias Mischek

Version 1.0

Note:

Betreuer:

Begonnen am 1. Juni 2016

Beendet am 8. Juni 2016

Inhaltsverzeichnis

1. Einführung

Mithilfe von SQL-Statements können Daten von einer Datenbank abgerufen und bearbeitet werden. Mit verschiedenen SQL-Statements kann jedoch das gleiche Ergebnis erzielt werden. Der einzige Unterschied kann hierbei dir Performance sein. Aus diesem Grund beschäftigt sich SQL-Tuning genau mit diesem Thema, nämlich der Optimierung von SQL-Statements. In den nachfolgenden Kapiteln werden Optimierungstechniken beschreiben und anhand von praktischen Beispielen gezeigt.

Als Erweiterung wird jeder Tipp auch noch mit der Schokofabrik-Datenbank getestet, da sich in dieser mehr Datensätze befinden und somit das Ergebnis deutlicher zu erkennen ist.

2. Aufgabenstellung

Dokumentieren Sie alle Tipps aus den vorgestellten Quellen [1,2] mit ausgeführten Queries aus den zur Verfügung gestellten Testdaten in einem PDF-Dokument. Zeigen Sie jeweils die Kosten der optimierten und nicht-optimierten Variante und diskutieren Sie das Ergebnis.
Die Übung ist als Gruppenarbeit (2er) zu absolvieren.
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.

3. SQL Tuning Beispiele

3.1. Spaltennamen angeben

Das SQL-Statement kann schneller ausgeführt werden, wenn die tatsächlichen Spaltennamen angegeben werden, anstatt beim Auslesen aller Spalten nur ein ‚*' einzutragen. [01]

3.1.1. Testung mit World Fact Book Datenbank

```
worldfactbook=> EXPLAIN SELECT * FROM city;
                        QUERY PLAN
-----------------------------------------------------------
 Seq Scan on city  (cost=0.00..54.11 rows=3111 width=42)
(1 row)

worldfactbook=> EXPLAIN ANALYZE SELECT * FROM city;
                                                  QUERY PLAN
-------------------------------------------------------------------------------------------------------------
 Seq Scan on city  (cost=0.00..54.11 rows=3111 width=42) (actual time=0.013..0.911 rows=3111 loops=1)
 Planning time: 0.050 ms
 Execution time: 1.576 ms
(3 rows)
```

```
worldfactbook=> EXPLAIN SELECT name, country, province, population, longitude, latitude FROM city;
                        QUERY PLAN
-----------------------------------------------------------
 Seq Scan on city  (cost=0.00..54.11 rows=3111 width=42)
(1 row)

worldfactbook=> EXPLAIN ANALYZE SELECT name, country, province, population, longitude, latitude FROM city;
                                                  QUERY PLAN
-------------------------------------------------------------------------------------------------------------
 Seq Scan on city  (cost=0.00..54.11 rows=3111 width=42) (actual time=0.018..0.867 rows=3111 loops=1)
 Planning time: 0.520 ms
 Execution time: 1.549 ms
(3 rows)
```

Wie man in diesem Beispiel sehen kann, verringert sich die „Execution time", also die Zeit die es dauert, um das Statement auszuführen.

3.1.2. Testung mit Schokodb Datenbank

```
SELECT persnr, vname, nname, geschlecht, gebdat FROM person;
```

```
                                                 QUERY PLAN
------------------------------------------------------------------------------------------------------------------
 Seq Scan on person  (cost=0.00..10177.00 rows=610000 width=23) (actual time=0.009..97.111 rows=610000 loops=1)
 Planning time: 0.066 ms
 Execution time: 167.500 ms
(3 rows)
```

```
SELECT * FROM person;
```

```
                                                 QUERY PLAN
------------------------------------------------------------------------------------------------------------------
 Seq Scan on person  (cost=0.00..10177.00 rows=610000 width=23) (actual time=0.014..97.859 rows=610000 loops=1)
 Planning time: 0.071 ms
 Execution time: 165.015 ms
(3 rows)
```

3.2. Anwendung von HAVING

`HAVING` wird verwendet, um nach dem Auslesen aller Spalten praktisch einen Filter anzuwenden. `HAVING` sollte nicht für sonstige Zwecke benutzt werden. [01]

3.2.1.Testung mit World Fact Book Datenbank

```
SELECT name FROM city WHERE name != 'Wien' AND name != 'New
York' GROUP BY name;
```

```
                                                                                                  QUERY PLAN
-------------------------------------------------------------------------------------------------------------------------------------------------
 HashAggregate  (cost=77.44..107.87 rows=3043 width=9) (actual time=2.375..3.075 rows=3043 loops=1)
   Group Key: name
   ->  Seq Scan on city  (cost=0.00..69.66 rows=3109 width=9) (actual time=0.021..1.073 rows=3110 loops=1)
         Filter: (((name)::text <> 'Wien'::text) AND ((name)::text <> 'New York'::text))
         Rows Removed by Filter: 1
 Planning time: 0.190 ms
 Execution time: 3.527 ms
(7 rows)
```

```
SELECT name FROM city GROUP BY name HAVING name != 'Wien' AND
name != 'New York';
```

```
                                                                                                  QUERY PLAN
-------------------------------------------------------------------------------------------------------------------------------------------------
 HashAggregate  (cost=77.44..107.87 rows=3043 width=9) (actual time=3.322..4.121 rows=3043 loops=1)
   Group Key: name
   ->  Seq Scan on city  (cost=0.00..69.66 rows=3109 width=9) (actual time=0.022..1.565 rows=3110 loops=1)
         Filter: (((name)::text <> 'Wien'::text) AND ((name)::text <> 'New York'::text))
         Rows Removed by Filter: 1
 Planning time: 0.112 ms
 Execution time: 4.682 ms
(7 rows)
```

Wie man erneut an diesem Beispiel sieht, ist die „Execution time“ erneut deutlich höher, obwohl die costs gleich sind.

3.2.2. Testung mit Schokodb Datenbank

```
SELECT persnr FROM person WHERE vname != 'Jori' AND nname !=
'Hellberg' GROUP BY persnr;
```

```
                                                              QUERY PLAN
---------------------------------------------------------------------------------------------------------------------------------------
 Group  (cost=0.42..23157.14 rows=609885 width=4) (actual time=0.031..412.129 rows=609881 loops=1)
   Group Key: persnr
   ->  Index Scan using person_pkey on person  (cost=0.42..21632.42 rows=609885 width=4) (actual time=0.030..213.735 rows=609881 loops=1)
         Filter: (((vname)::text <> 'Jori'::text) AND ((nname)::text <> 'Hellberg'::text))
         Rows Removed by Filter: 119
 Planning time: 0.227 ms
 Execution time: 478.573 ms
(7 rows)
```

```
SELECT persnr FROM person GROUP BY persnr HAVING vname != 'Jori'
AND nname != 'Hellberg';
```

```
                                                              QUERY PLAN
---------------------------------------------------------------------------------------------------------------------------------------
 Group  (cost=0.42..23157.14 rows=609885 width=4) (actual time=0.028..398.034 rows=609881 loops=1)
   Group Key: persnr
   ->  Index Scan using person_pkey on person  (cost=0.42..21632.42 rows=609885 width=4) (actual time=0.025..205.532 rows=609881 loops=1)
         Filter: (((vname)::text <> 'Jori'::text) AND ((nname)::text <> 'Hellberg'::text))
         Rows Removed by Filter: 119
 Planning time: 0.198 ms
 Execution time: 463.227 ms
(7 rows)
```

3.3. Wenig Subqueries

Durch weniger Subqueries kann die Performance, laut dem Tutorial, genau so gesteigert werden, was man auch anhand dieses Beispiels sehen kann. [01]

3.3.1. Testung mit World Fact Book Datenbank

```
SELECT name FROM ismember WHERE (organization, type)
IN (SELECT organization, type FROM ismember WHERE organization
LIKE 'A%' AND type LIKE 'member');
```

```
                                                              QUERY PLAN
--------------------------------------------------------------------------------------------------------------------------------------
 Hash Join  (cost=168.81..385.98 rows=8008 width=15) (actual time=2.691..6.790 rows=265 loops=1)
   Hash Cond: (((ismember.organization)::text = (ismember_1.organization)::text) AND ((ismember.type)::text = (ismember_1.type)::text))
   ->  Seq Scan on ismember  (cost=0.00..126.08 rows=8008 width=15) (actual time=0.014..1.815 rows=8008 loops=1)
   ->  Hash  (cost=168.27..168.27 rows=36 width=12) (actual time=2.660..2.660 rows=13 loops=1)
         Buckets: 1024  Batches: 1  Memory Usage: 5kB
         ->  HashAggregate  (cost=167.91..168.27 rows=36 width=12) (actual time=2.642..2.647 rows=13 loops=1)
               Group Key: (ismember_1.organization)::text, (ismember_1.type)::text
               ->  Seq Scan on ismember ismember_1  (cost=0.00..166.12 rows=358 width=12) (actual time=0.011..2.465 rows=265 loops=1)
                     Filter: (((organization)::text ~~ 'A%'::text) AND ((type)::text ~~ 'member'::text))
                     Rows Removed by Filter: 7743
 Planning time: 0.526 ms
 Execution time: 6.941 ms
(12 rows)
```

```
SELECT country FROM ismember WHERE organization = (SELECT
organization FROM ismember WHERE organization LIKE 'member') AND
type = (SELECT type FROM ismember WHERE type LIKE 'A%');
```

3.4. JOINS statt Subqueries

Bei Beginner SQL Tutorial wird beschrieben, dass durch weniger Subqueries die Performance, laut dem Tutorial, genau so gesteigert werden kann, was man auch anhand dieses Beispiels sehen kann. [02]

3.4.1. Testung mit World Fact Book Datenbank

```
SELECT s.name, c.code FROM city s
INNER JOIN country c ON s.country = c.code;
```

```
                                                                QUERY PLAN
-------------------------------------------------------------------------------------------------------------------------------------------
 Hash Join  (cost=8.36..105.24 rows=3111 width=12) (actual time=0.179..2.550 rows=3111 loops=1)
   Hash Cond: ((s.country)::text = (c.code)::text)
   ->  Seq Scan on city s  (cost=0.00..54.11 rows=3111 width=12) (actual time=0.008..0.659 rows=3111 loops=1)
   ->  Hash  (cost=5.38..5.38 rows=238 width=3) (actual time=0.153..0.153 rows=238 loops=1)
         Buckets: 1024  Batches: 1  Memory Usage: 10kB
         ->  Seq Scan on country c  (cost=0.00..5.38 rows=238 width=3) (actual time=0.006..0.074 rows=238 loops=1)
 Planning time: 0.296 ms
 Execution time: 3.069 ms
(8 rows)
```

```
SELECT s.name, (SELECT c.code FROM country c
WHERE code = s.country) FROM city s;
```

```
                                                                QUERY PLAN
-------------------------------------------------------------------------------------------------------------------------------------------
 Seq Scan on city s  (cost=0.00..18642.34 rows=3111 width=12) (actual time=0.153..84.702 rows=3111 loops=1)
   SubPlan 1
     ->  Seq Scan on country c  (cost=0.00..5.98 rows=1 width=3) (actual time=0.010..0.026 rows=1 loops=3111)
           Filter: ((code)::text = (s.country)::text)
           Rows Removed by Filter: 237
 Planning time: 0.207 ms
 Execution time: 85.130 ms
(7 rows)
```

Wie man erneut an diesem Beispiel sieht, ist die „Execution time“ erneut deutlich höher, genauso wie die costs.

3.4.2.Testung mit Schokodb Datenbank

```
SELECT p.persnr, p.nname, p.vname, a.gehalt FROM angestellter a
INNER JOIN person p ON a.persnr = p.persnr;
```

```
                                                          QUERY PLAN
-------------------------------------------------------------------------------------------------------------------------------------------------------
 Merge Join  (cost=6598.02..21938.59 rows=199999 width=22) (actual time=97.068..308.044 rows=199999 loops=1)
   Merge Cond: (a.persnr = p.persnr)
   ->  Index Scan using angestellter_pkey on angestellter a  (cost=0.29..6230.28 rows=199999 width=9) (actual time=0.010..47.785 rows=199999 loops=1)
   ->  Index Scan using person_pkey on person p  (cost=0.42..18582.42 rows=610000 width=17) (actual time=0.015..108.785 rows=400000 loops=1)
 Planning time: 0.243 ms
 Execution time: 330.655 ms
(6 rows)
```

```
SELECT s.name, (SELECT c.code FROM country c
WHERE code = s.country) FROM city s;
```

```
                                                          QUERY PLAN
-------------------------------------------------------------------------------------------------------------------------------------------------------
 Seq Scan on person p  (cost=0.00..5080802.00 rows=610000 width=17) (actual time=0.023..941.131 rows=610000 loops=1)
   SubPlan 1
     ->  Index Scan using angestellter_pkey on angestellter a  (cost=0.29..8.31 rows=1 width=5) (actual time=0.001..0.001 rows=0 loops=610000)
           Index Cond: (persnr = p.persnr)
 Planning time: 0.102 ms
 Execution time: 1016.836 ms
(6 rows)
```

Bei mehreren Datenmengen, wie beispielsweise im Umfang der Schokofabrik Datenbank sieht man deutlich, dass mehr costs und Zeit benötigt werden, um das SELECT-Statement auszuführen.

3.5. UNION statt OR

Der Tipp von Beginner SQL Tutorial: Die Verwendung von `UNION` anstatt von `OR` ist wesentlich schneller. Dies kann einfach mit einem `SELECT` getestet werden. [02]

3.5.1. Testung mit World Fact Book Datenbank

```
SELECT * FROM city WHERE name LIKE 'A%'
UNION
SELECT * FROM city WHERE name LIKE 'B%';
```

```
                                                                                                                  QUERY PLAN
-----------------------------------------------------------------------------------------------------------------------------------------------------------------
 HashAggregate  (cost=133.90..137.95 rows=405 width=42) (actual time=1.581..1.700 rows=415 loops=1)
   Group Key: city.name, city.country, city.province, city.population, city.longitude, city.latitude
   ->  Append  (cost=0.00..127.83 rows=405 width=42) (actual time=0.017..1.224 rows=415 loops=1)
         ->  Seq Scan on city  (cost=0.00..61.89 rows=160 width=42) (actual time=0.016..0.554 rows=183 loops=1)
               Filter: ((name)::text ~~ 'A%'::text)
               Rows Removed by Filter: 2928
         ->  Seq Scan on city city_1  (cost=0.00..61.89 rows=245 width=42) (actual time=0.008..0.549 rows=232 loops=1)
               Filter: ((name)::text ~~ 'B%'::text)
               Rows Removed by Filter: 2879
 Planning time: 0.132 ms
 Execution time: 1.795 ms
(11 rows)
```

```
SELECT * FROM city
WHERE name LIKE 'A%' OR name LIKE 'B%';
```

```
                                                                                                  QUERY PLAN
-------------------------------------------------------------------------------------------------------------------------------------------------------------------
 Seq Scan on city  (cost=0.00..69.66 rows=393 width=42) (actual time=0.018..0.791 rows=415 loops=1)
   Filter: (((name)::text ~~ 'A%'::text) OR ((name)::text ~~ 'B%'::text))
   Rows Removed by Filter: 2696
 Planning time: 0.123 ms
 Execution time: 0.882 ms
(5 rows)
```

Wie man anhand dieses Beispiels sieht, stimmt der gegebene Tipp in diesem Fall lediglich nicht, wenn man die costs und „execution time"'s beachtet.

3.5.2.Testung mit Schokodb Datenbank

```
SELECT * FROM person WHERE vname LIKE 'A%'
UNION
SELECT * FROM person WHERE nname LIKE 'B%';
```

```
                                                       QUERY PLAN
-----------------------------------------------------------------------------------------------------------------------------------
 HashAggregate  (cost=25666.04..26671.39 rows=100535 width=23) (actual time=232.722..266.289 rows=100397 loops=1)
   Group Key: person.persnr, person.vname, person.nname, person.geschlecht, person.gebdat
   ->  Append  (cost=0.00..24409.35 rows=100535 width=23) (actual time=0.027..173.208 rows=104874 loops=1)
         ->  Seq Scan on person  (cost=0.00..11702.00 rows=45093 width=23) (actual time=0.026..79.019 rows=50185 loops=1)
               Filter: ((vname)::text ~~ 'A%'::text)
               Rows Removed by Filter: 559815
         ->  Seq Scan on person person_1  (cost=0.00..11702.00 rows=55442 width=23) (actual time=0.013..71.464 rows=54689 loops=1)
               Filter: ((nname)::text ~~ 'B%'::text)
               Rows Removed by Filter: 555311
 Planning time: 0.285 ms
 Execution time: 277.109 ms
(11 rows)
```

```
SELECT * FROM person
WHERE vname LIKE 'A%' OR nname LIKE 'B%';
```

```
                                                QUERY PLAN
-----------------------------------------------------------------------------------------------------------
 Seq Scan on person  (cost=0.00..13227.00 rows=96436 width=23) (actual time=0.020..117.921 rows=100397 loops=1)
   Filter: (((vname)::text ~~ 'A%'::text) OR ((nname)::text ~~ 'B%'::text))
   Rows Removed by Filter: 509603
 Planning time: 0.138 ms
 Execution time: 129.454 ms
(5 rows)
```

Wie man hier wieder sehen kann, steigen die costs und Dauer sehr stark an, wenn die Datensätzeanzahl erhöht wird.

3.6. Regex-Expression

In dem Tutorial der ersten Quelle wird beschrieben, dass `SELECT` Befehle mit einer Suche mit Regex-Expressions schneller ausgeführt werden als mit Substrings. Dieser Tipp wird überprüft, indem alle Städte ausgegeben werden, welche mit „Be“ beginnen. [01]

3.6.1. Testung mit World Fact Book Datenbank

```
SELECT name, country, province, population, longitude, latitude
FROM city
WHERE name LIKE 'Be%';
```

```
                                                  QUERY PLAN
-------------------------------------------------------------------------------------------------------------
 Seq Scan on city  (cost=0.00..61.89 rows=30 width=42) (actual time=0.017..0.450 rows=43 loops=1)
   Filter: ((name)::text ~~ 'Be%'::text)
   Rows Removed by Filter: 3068
 Planning time: 0.119 ms
 Execution time: 0.478 ms
(5 rows)
```

```
SELECT name, country, province, population, longitude, latitude
FROM city
WHERE SUBSTR(name,1,2) = 'Be';
```

```
                                                  QUERY PLAN
-------------------------------------------------------------------------------------------------------------
 Seq Scan on city  (cost=0.00..69.66 rows=16 width=42) (actual time=0.021..0.733 rows=43 loops=1)
   Filter: (substr((name)::text, 1, 2) = 'Be'::text)
   Rows Removed by Filter: 3068
 Planning time: 0.053 ms
 Execution time: 0.761 ms
(5 rows)
```

Wie man an diesem Beispiel sehen kann, hat dieser Tipp eine positive Auswirkung auf die Performance und die „execution time“.

3.6.2. Testung mit Schokodb Datenbank

```
SELECT persnr, vname, nname, geschlecht, gebdat
FROM person
WHERE vname LIKE 'Be%';
```

```
                                                        QUERY PLAN
-------------------------------------------------------------------------------------------------------------------
 Seq Scan on person  (cost=0.00..11702.00 rows=12288 width=23) (actual time=0.041..81.359 rows=10029 loops=1)
   Filter: ((vname)::text ~~ 'Be%'::text)
   Rows Removed by Filter: 599971
 Planning time: 0.106 ms
 Execution time: 82.686 ms
(5 rows)
```

```
SELECT persnr, vname, nname, geschlecht, gebdat
FROM person
WHERE SUBSTR(vname,1,2) = 'Be';
```

```
                                                        QUERY PLAN
-------------------------------------------------------------------------------------------------------------------
 Seq Scan on person  (cost=0.00..13227.00 rows=3050 width=23) (actual time=0.057..140.587 rows=10029 loops=1)
   Filter: (substr((vname)::text, 1, 2) = 'Be'::text)
   Rows Removed by Filter: 599971
 Planning time: 0.083 ms
 Execution time: 141.867 ms
(5 rows)
```

Wie man an diesem Beispiel sehen kann, hat dieser Tipp eine positive Auswirkung auf die Performance und die „execution time", in jedem Fall bei großen Datenbanken.

3.7. BETWEEN verwenden

Ein `SELECT`-Befehl erfolgt schneller, wenn bei einer Suche statt Größer- und Kleinervergleichen der Befehl `BETWEEN` verwendet wird. [01]

3.7.1. Testung mit World Fact Book Datenbank

```
SELECT name, population, country
FROM city
WHERE population BETWEEN 1000 AND 100000;
```

```
                                                    QUERY PLAN
-------------------------------------------------------------------------------------------------------------------
 Seq Scan on city  (cost=0.00..69.66 rows=227 width=16) (actual time=0.012..0.423 rows=220 loops=1)
   Filter: ((population >= 1000) AND (population <= 100000))
   Rows Removed by Filter: 2891
 Planning time: 0.056 ms
 Execution time: 0.462 ms
(5 rows)
```

```
SELECT name, population, country
FROM city
WHERE population >= 1000 AND population <= 100000;
```

```
                                                    QUERY PLAN
-------------------------------------------------------------------------------------------------------------------
 Seq Scan on city  (cost=0.00..69.66 rows=227 width=16) (actual time=0.013..0.488 rows=220 loops=1)
   Filter: ((population >= 1000) AND (population <= 100000))
   Rows Removed by Filter: 2891
 Planning time: 0.061 ms
 Execution time: 0.532 ms
(5 rows)
```

Wie man von diesen beiden Query plan's ablesen kann, bleiben die *costs* bei beiden Versuchen zwar gleich, allerdings sinkt die „Execution time", bei der Benutzung von `BETWEEN`.

3.7.2. Testung mit Schokodb Datenbank

```
SELECT persnr, vname, nname
FROM person
WHERE persnr BETWEEN 20000 AND 50000;
```

```
                                                         QUERY PLAN
------------------------------------------------------------------------------------------------------------------------------
 Index Scan using person_pkey on person  (cost=0.42..513.52 rows=14305 width=17) (actual time=0.014..6.572 rows=15001 loops=1)
   Index Cond: ((persnr >= 20000) AND (persnr <= 50000))
 Planning time: 0.123 ms
 Execution time: 8.956 ms
(4 rows)
```

```
SELECT persnr, vname, nname
FROM person
WHERE persnr >= 20000 AND persnr <= 50000;
```

```
                                                         QUERY PLAN
------------------------------------------------------------------------------------------------------------------------------
 Index Scan using person_pkey on person  (cost=0.42..513.52 rows=14305 width=17) (actual time=0.013..6.331 rows=15001 loops=1)
   Index Cond: ((persnr >= 20000) AND (persnr <= 50000))
 Planning time: 0.124 ms
 Execution time: 8.772 ms
(4 rows)
```

Wie man von diesen beiden Query plan’s ablesen kann, bleiben die *costs* bei beiden Versuchen zwar gleich, allerdings sinkt die Zeit um das Statement auszuführen, bei der Benutzung von `BETWEEN`.

3.8. EXISTS statt IN

Bei der Verwendung von `EXISTS` statt `IN`, kann die Performance bei der Überprüfung gesteigert werden. [01]

3.8.1.Testung mit World Fact Book Datenbank

```
SELECT * FROM country k
WHERE EXISTS (SELECT * FROM city c WHERE
c.country=k.code);
```

```
                                                    QUERY PLAN
------------------------------------------------------------------------------------------------------------------------
 Hash Join  (cost=67.24..75.89 rows=238 width=42) (actual time=2.696..2.898 rows=238 loops=1)
   Hash Cond: ((k.code)::text = (c.country)::text)
   ->  Seq Scan on country k  (cost=0.00..5.38 rows=238 width=42) (actual time=0.009..0.066 rows=238 loops=1)
   ->  Hash  (cost=64.27..64.27 rows=238 width=3) (actual time=2.677..2.677 rows=238 loops=1)
         Buckets: 1024  Batches: 1  Memory Usage: 10kB
         ->  HashAggregate  (cost=61.89..64.27 rows=238 width=3) (actual time=2.533..2.594 rows=238 loops=1)
               Group Key: (c.country)::text
               ->  Seq Scan on city c  (cost=0.00..54.11 rows=3111 width=3) (actual time=0.009..1.210 rows=3111 loops=1)
 Planning time: 0.359 ms
 Execution time: 2.973 ms
(10 rows)
```

```
SELECT * FROM country
WHERE code IN (SELECT country FROM city);
```

```
                                                    QUERY PLAN
------------------------------------------------------------------------------------------------------------------------
 Hash Join  (cost=67.24..75.89 rows=238 width=42) (actual time=2.574..2.770 rows=238 loops=1)
   Hash Cond: ((country.code)::text = (city.country)::text)
   ->  Seq Scan on country  (cost=0.00..5.38 rows=238 width=42) (actual time=0.007..0.054 rows=238 loops=1)
   ->  Hash  (cost=64.27..64.27 rows=238 width=3) (actual time=2.556..2.556 rows=238 loops=1)
         Buckets: 1024  Batches: 1  Memory Usage: 10kB
         ->  HashAggregate  (cost=61.89..64.27 rows=238 width=3) (actual time=2.411..2.472 rows=238 loops=1)
               Group Key: (city.country)::text
               ->  Seq Scan on city  (cost=0.00..54.11 rows=3111 width=3) (actual time=0.007..1.030 rows=3111 loops=1)
 Planning time: 0.361 ms
 Execution time: 2.852 ms
(10 rows)
```

Auch wenn die Unterschiede jetzt auf den ersten Blick nicht sehr groß sind, die „Execution time“ hat sich zwischen den beiden Statements verringert, auch wenn die *costs* gleich geblieben sind.

3.8.2.Testung mit Schokodb Datenbank

```
SELECT * FROM angestellter a
WHERE EXISTS (SELECT * FROM person p WHERE
p.persnr=a.persnr);
```

```
                                                                          QUERY PLAN
----------------------------------------------------------------------------------------------------------------------------------
 Merge Semi Join  (cost=6598.02..21938.59 rows=199999 width=28) (actual time=90.845..291.957 rows=199999 loops=1)
   Merge Cond: (a.persnr = p.persnr)
   ->  Index Scan using angestellter_pkey on angestellter a  (cost=0.29..6230.28 rows=199999 width=28) (actual time=0.011..44.675 rows=199999 loops=1)
   ->  Index Only Scan using person_pkey on person p  (cost=0.42..18582.42 rows=610000 width=4) (actual time=0.028..112.501 rows=399999 loops=1)
         Heap Fetches: 399999
 Planning time: 0.259 ms
 Execution time: 312.452 ms
(7 rows)
```

```
SELECT * FROM country
WHERE code IN (SELECT country FROM city);
```

```
                                                                          QUERY PLAN
----------------------------------------------------------------------------------------------------------------------------------
 Merge Semi Join  (cost=6598.02..21938.59 rows=199999 width=28) (actual time=86.406..287.111 rows=199999 loops=1)
   Merge Cond: (angestellter.persnr = person.persnr)
   ->  Index Scan using angestellter_pkey on angestellter  (cost=0.29..6230.28 rows=199999 width=28) (actual time=0.008..44.734 rows=199999 loops=1)
   ->  Index Only Scan using person_pkey on person  (cost=0.42..18582.42 rows=610000 width=4) (actual time=0.101..109.359 rows=399999 loops=1)
         Heap Fetches: 399999
 Planning time: 0.213 ms
 Execution time: 307.686 ms
(7 rows)
```

Auch wenn die Unterschiede jetzt auf den ersten Blick nicht sehr groß sind, die „Execution time“ hat sich zwischen den beiden Statements verringert, auch wenn die *costs* gleich geblieben sind.

3.9. UNION ALL statt UNION

Wie die Tutorialseite beschreibt, soll statt `UNION`, `UNION ALL` benutzt werden, um eine schnellere `SELECT` Abfrage zu erhalten. `UNION` wird eher dann verwendet, wenn mehrere `SELECT`-Statements zu einem zusammengefasst werden sollen. `UNION` hingegen filtert doppelte Datensätze hinaus. [01]

3.9.1. Testung mit World Fact Book Datenbank

```
SELECT country FROM city
UNION ALL
SELECT code FROM country;
```

```
                                                    QUERY PLAN
-------------------------------------------------------------------------------------------------------------------
 Append  (cost=0.00..59.49 rows=3349 width=3) (actual time=0.058..2.477 rows=3349 loops=1)
   ->  Seq Scan on city  (cost=0.00..54.11 rows=3111 width=3) (actual time=0.057..1.178 rows=3111 loops=1)
   ->  Seq Scan on country  (cost=0.00..5.38 rows=238 width=3) (actual time=0.033..0.096 rows=238 loops=1)
 Planning time: 0.402 ms
 Execution time: 3.082 ms
(5 rows)
```

```
SELECT country FROM city
UNION
SELECT code FROM country;
```

```
                                                    QUERY PLAN
-------------------------------------------------------------------------------------------------------------------
 HashAggregate  (cost=101.35..134.84 rows=3349 width=3) (actual time=3.542..3.608 rows=238 loops=1)
   Group Key: city.country
   ->  Append  (cost=0.00..92.98 rows=3349 width=3) (actual time=0.013..2.282 rows=3349 loops=1)
         ->  Seq Scan on city  (cost=0.00..54.11 rows=3111 width=3) (actual time=0.013..1.064 rows=3111 loops=1)
         ->  Seq Scan on country  (cost=0.00..5.38 rows=238 width=3) (actual time=0.005..0.065 rows=238 loops=1)
 Planning time: 0.123 ms
 Execution time: 3.799 ms
(7 rows)
```

In diesem Beispiel wird deutlich, dass sowohl *costs* als auch „Execution time“ eingespart werden können, wenn ein `UNION ALL` statt einem `UNION` verwendet wird.

3.9.2. Testung mit Schokodb Datenbank

```
SELECT persnr FROM person
UNION ALL
SELECT persnr FROM angestellter;
```

```
                                                     QUERY PLAN
------------------------------------------------------------------------------------------------------------------------
 Append  (cost=0.00..13647.99 rows=809999 width=4) (actual time=0.015..310.094 rows=809999 loops=1)
   ->  Seq Scan on person  (cost=0.00..10177.00 rows=610000 width=4) (actual time=0.014..108.672 rows=610000 loops=1)
   ->  Seq Scan on angestellter  (cost=0.00..3470.99 rows=199999 width=4) (actual time=0.005..34.321 rows=199999 loops=1)
 Planning time: 0.099 ms
 Execution time: 394.482 ms
(5 rows)
```

```
SELECT persnr FROM person
UNION
SELECT persnr FROM angestellter;
```

```
                                                     QUERY PLAN
------------------------------------------------------------------------------------------------------------------------
 Unique  (cost=110931.00..114981.00 rows=809999 width=4) (actual time=834.380..1262.847 rows=610000 loops=1)
   ->  Sort  (cost=110931.00..112956.00 rows=809999 width=4) (actual time=834.378..1036.948 rows=809999 loops=1)
         Sort Key: person.persnr
         Sort Method: external merge  Disk: 11072kB
         ->  Append  (cost=0.00..21747.98 rows=809999 width=4) (actual time=0.010..345.248 rows=809999 loops=1)
               ->  Seq Scan on person  (cost=0.00..10177.00 rows=610000 width=4) (actual time=0.009..124.217 rows=610000 loops=1)
               ->  Seq Scan on angestellter  (cost=0.00..3470.99 rows=199999 width=4) (actual time=0.006..39.356 rows=199999 loops=1)
 Planning time: 0.054 ms
 Execution time: 1333.967 ms
(9 rows)
```

In diesem Beispiel wird deutlich, dass sowohl *costs* als auch „Execution time" eingespart werden können, wenn ein `UNION ALL` statt einem `UNION` verwendet wird.

3.10. Weniger Zahlenwerte bei WHERE

Ein weiterer Tipp ist es, möglichst wenig Zahlenwerte (nicht-Attribute) in den `SELECT`-Abfragen zu verwenden. Dadurch kann die `SELECT`-Abfrage ebenfalls bei einer großen Anzahl an Datensätzen, beschleunigt werden. [01]

3.10.1. Testung mit World Fact Book Datenbank

```
SELECT country1, country2, length FROM borders
WHERE length < 1000;
```

```
                                                    QUERY PLAN
-------------------------------------------------------------------------------------------------------------------
 Seq Scan on borders  (cost=0.00..6.00 rows=246 width=14) (actual time=0.017..0.131 rows=246 loops=1)
   Filter: (length < '1000'::double precision)
   Rows Removed by Filter: 74
 Planning time: 0.085 ms
 Execution time: 0.203 ms
(5 rows)
```

```
SELECT country1, country2, length FROM borders
WHERE length + 100 < 1100;
```

```
                                                    QUERY PLAN
-------------------------------------------------------------------------------------------------------------------
 Seq Scan on borders  (cost=0.00..6.80 rows=107 width=14) (actual time=0.019..0.176 rows=246 loops=1)
   Filter: ((length + '100'::double precision) < '1100'::double precision)
   Rows Removed by Filter: 74
 Planning time: 0.185 ms
 Execution time: 0.311 ms
(5 rows)
```

Wie man sieht, ist bei wenigen Datensätzen kein dramatischer Unterschied zu sehen, allerdings kann dieser Tipp bei größerer Datensätzenanzahl mehr Auswirkungen haben.

3.10.2. Testung mit Schokodb Datenbank

```
SELECT persnr, vname, nname FROM person
WHERE persnr > 40000;
```

```
                                                    QUERY PLAN
--------------------------------------------------------------------------------------------------------------------
 Seq Scan on person  (cost=0.00..11702.00 rows=595697 width=17) (actual time=2.490..147.187 rows=594999 loops=1)
   Filter: (persnr > 40000)
   Rows Removed by Filter: 15001
 Planning time: 0.071 ms
 Execution time: 213.317 ms
(5 rows)
```

```
SELECT persnr, vname, nname FROM person
WHERE persnr + 1000 > 39000;
```

```
                                                    QUERY PLAN
--------------------------------------------------------------------------------------------------------------------
 Seq Scan on person  (cost=0.00..13227.00 rows=203333 width=17) (actual time=2.775..158.139 rows=595999 loops=1)
   Filter: ((persnr + 1000) > 39000)
   Rows Removed by Filter: 14001
 Planning time: 0.053 ms
 Execution time: 223.306 ms
(5 rows)
```

Wie man sieht, ist bei wenigen Datensätzen kein dramatischer Unterschied zu sehen, allerdings kann dieser Tipp bei größerer Datensätzenanzahl mehr Auswirkungen haben.

3.11. AND statt ||

Im Tutorial wird des weiteren beschrieben, dass `AND` statt `||` auch die Performance verbessern kann. Das wird mit dem folgenden Beispiel überprüft. [01]

3.11.1. Testung mit World Fact Book Datenbank

```
SELECT name, country, province
FROM city
WHERE country='AL' AND province=,Albania';
```

```
                                                  QUERY PLAN
-------------------------------------------------------------------------------------------------------------
 Seq Scan on city  (cost=0.00..69.66 rows=1 width=22) (actual time=0.017..0.630 rows=6 loops=1)
   Filter: (((country)::text = 'AL'::text) AND ((province)::text = 'Albania'::text))
   Rows Removed by Filter: 3105
 Planning time: 0.106 ms
 Execution time: 0.652 ms
(5 rows)
```

```
SELECT name, country, province
FROM city
WHERE country || province='ALAlbania';
```

```
                                                  QUERY PLAN
-------------------------------------------------------------------------------------------------------------
 Seq Scan on city  (cost=0.00..69.66 rows=16 width=22) (actual time=0.020..1.175 rows=6 loops=1)
   Filter: (((country)::text || (province)::text) = 'ALAlbania'::text)
   Rows Removed by Filter: 3105
 Planning time: 0.073 ms
 Execution time: 1.199 ms
(5 rows)
```

Obwohl die Kosten gleich geblieben sind, hat sich die „Execution time“ verringert, wenn man diesen Tipp befolgt.

3.11.2. Testung mit Schokodb Datenbank

```
SELECT persnr, vname, nname
FROM person
WHERE geschlecht='M' AND vname='Ahmet';
```

```
                                                    QUERY PLAN
-------------------------------------------------------------------------------------------------------------------
 Seq Scan on person  (cost=0.00..13227.00 rows=23 width=17) (actual time=3.430..88.539 rows=24 loops=1)
   Filter: (((geschlecht)::text = 'M'::text) AND ((vname)::text = 'Ahmet'::text))
   Rows Removed by Filter: 609976
 Planning time: 0.091 ms
 Execution time: 88.592 ms
(5 rows)
```

```
SELECT persnr, vname, nname
FROM person
WHERE geschlecht || vname='MAhmet';
```

```
                                                    QUERY PLAN
-------------------------------------------------------------------------------------------------------------------
 Seq Scan on person  (cost=0.00..13227.00 rows=3050 width=17) (actual time=3.801..119.718 rows=24 loops=1)
   Filter: (((geschlecht)::text || (vname)::text) = 'MAhmet'::text)
   Rows Removed by Filter: 609976
 Planning time: 0.061 ms
 Execution time: 119.747 ms
(5 rows)
```

Obwohl die Kosten gleich geblieben sind, hat sich die „Execution time“ verringert, wenn man diesen Tipp befolgt.

4. Quellen

[01] SQL QUERY TUNING [ONLINE] AVAILABLE AT:
HTTP://BEGINNER-SQL-TUTORIAL.COM/SQL-QUERY-TUNING.HTM
[ABGERUFEN AM 4. JUNI 2016]

[02] SQL QUERY TIPS [ONLINE] AVAILABLE AT:
HTTP://BEGINNER-SQL-TUTORIAL.COM/SQL-TUTORIAL-TIPS.HTM
[ABGERUFEN AM 4. JUNI 2016]