JUST DO IT
HÖHERE ABTEILUNG FÜR INFORMATIONSTECHNOLOGIE

tgm
Die Schule der Technik

# Laborprotokoll

# SQL-TUNING VERGLEICH

**INSY Übung**
**4AHITM 2015/16**

**Kanyildiz Muhammedhizir & Oezsoy Ahmet**

**Version 1.0**

**Note:**

**Betreuer: Prof. Borko**

**Begonnen am 18. Mai 2016**

**Beendet am 25. Mai 2016**

# Inhaltsverzeichnis

# 1 Einführung

## 1.1 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.

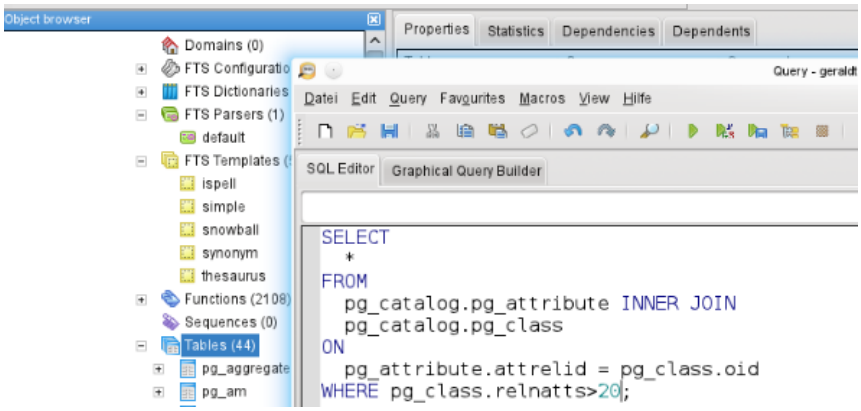
# 2 PG-Admin [3]

pgAdmin ist eine Open-Source-Softwarezur Entwicklung und Administration 3Eine graphische Benutzeroberfläche erleichtert die Administration von Datenbanken. Der Editor für SQL-Abfragen enthält ein graphisches EXPLAIN, mit dessen Hilfe sich performantere Abfragen erstellen lassen. Durch eine native Anbindung an PostgreSQL ermöglicht das GUI den Zugriff auf die gesamte PostgreSQL-Funktionalität. Die Software wird in Versionen für die Betriebssysteme Linux, Mac OS X,FreeBSD, Solaris und Windows angeboten.

**Aktuelle Version:** 1.20.0

**Programmiersprache:** C++

**Lizenz:** PostgreSQL Lizenz




# 3 Query Plan [4]

Bei einem Query Plan handelt es sich um eine Beschreibung, in welchen Einzelschritten ein relationales eine Datenbankabfrage ausführt und in welcher Reihenfolge dies geschieht. Er wird vom Anfrageoptimierer des Datenbankmanagementsystems generiert, wenn eine Datenbankabfrage gestellt wird.

# 4 Aufgaben

## 4.1 Nummer 1 [1]

### Erklärung

Die SQL Abfrage wird schneller wenn man den eigentlichen Spaltennamen im SELECT-Statement benutz anstatt eines * (Stern).

### Nicht-optimierte Variante:

#### *Abfrage*

```
SELECT * FROM employees;
```

#### *Kosten*

| | QUERY PLAN text |
|---|---|
| 1 | Seq Scan on employees (cost=0.00..810.33 rows=29833 width=105) (actual time=0.008..2.398 rows=29833 loops=1) |

### Optimierte Variante:

#### *Abfrage*

```
SELECT firstname, lastname FROM employees;
```

#### *Kosten:*

| | QUERY PLAN text |
|---|---|
| 1 | Seq Scan on employees (cost=0.00..810.33 rows=29833 width=13) (actual time=0.007..3.826 rows=29833 loops=1) |

### Ergebnisinterpretation:

Bei der optimierten Variante werden nicht alle Spalten, sondern nur die benötigten Spalten ausgelesen, wodurch sehr viel Zeit gespart wird aufgrund der kleineren „width“. Das heißt, es muss weniger ausgelesen werden.

## 4.2 Nummer 2 [1]

### Erklärung

Die HAVING-Clause ist für die Filterung der Zeilen nach der Darstellung der Spalten da. Es ist wie ein Filter. Die HAVING-Clause sollte man nicht für andere Sachen verwendet werden

### Nicht-optimierte Variante:

#### *Abfrage:*

```
SELECT firstname, COUNT(firstname)
FROM employees
GROUP BY firstname
HAVING firstname != 'Edward'
AND firstname != ''Tad';
```

#### *Kosten:*

| | QUERY PLAN text |
|---|---|
| 1 | HashAggregate (cost=1108.33..1119.97 rows=1164 width=6) (actual time=12.362..12.481 rows=1164 loops=1) |

### Optimierte Variante:

#### *Abfrage:*

```
SELECT firstname, COUNT(firstname)
FROM employees
WHERE firstname != 'Edward'
AND firstname != ''Tad';
GROUP BY firstname;
```

#### *Kosten:*

| | QUERY PLAN text |
|---|---|
| 1 | HashAggregate (cost=1108.33..1119.97 rows=1164 width=6) (actual time=12.313..12.424 rows=1164 loops=1) |

### Ergebnisinterpretation:

HAVING benötigt mehr Zeit, da es zuerst alle Zeilen auswählt und und anschließend die benötigten Zeilen daraus filtert. Having dient also als ein Filter und sollte nur für zum Beispiel MAX, MIN, etc. verwendet werden. Ansonsten sollte man HAVING vermeiden und stattdessen WHERE verwenden.

## 4.3 Nummer 3 [1]

### Erklärung

Man sollte so wenige subqueries wie möglich haben.

### Nicht-optimierte Variante:

#### *Abfrage:*

```
SELECT * FROM employees
WHERE hiredate = (SELECT MAX(hiredate) FROM employees)
AND religion = (SELECT religion FROM employees GROUP BY religion ORDER
BY COUNT(*) DESC LIMIT 1);
```

#### *Kosten:*

### Optimierte Variante:

| | QUERY PLAN text |
|---|---|
| 1 | Seq Scan on employees (cost=1844.97..2804.47 rows=1 width=105) (actual time=25.374..26.012 rows=1 loops=1) |

#### *Abfrage:*

```
SELECT * FROM employees
WHERE (hiredate, religion) = (SELECT MAX(hiredate),religion FROM employees
employees GROUP BY religion ORDER BY COUNT (*) DESC LIMIT 1);
```

#### *Kosten:*

| | QUERY PLAN text |
|---|---|
| 1 | Seq Scan on employees (cost=1034.63..1994.13 rows=1 width=105) (actual time=19.447..20.151 rows=1 loops=1) |

### Ergebnisinterpretation:

Aufgrund der Reduzierung der Subqueries wird einiges an Zeit gespart, da sich die Anzahl der der temporären Tabellen, die für die Generierung des Ergebnisses notwendig sind, verringert.

## 4.4 Nummer 4 [1]

### Erklärung

Die Operatoren EXISTS, IN und Table JOINS in der Query angemessen verwenden.

- IN hat normalerweise die schlechteste Performance.
  IN ist effizient, wenn das meiste der Filterung im subquery ist.
  EXISTS ist effizient, wenn das meiste der Filterung in der Mainquery ist.

### Nicht-optimierte Variante:

#### *Abfrage:*

```
SELECT * FROM employees e
WHERE e.city IN (SELECT name FROM city c WHERE e.city = c.name AND
name LIKE 'G%');
```

#### *Kosten:*

| | QUERY PLAN text |
|---|---|
| 1 | Seq Scan on employees e (cost=0.00..128905.77 rows=14916 width=105) (actual time=0.096..380.631 rows=1016 loops=1) |

### Optimierte Variante:

#### *Abfrage:*

```
SELECT * FROM employees e
WHERE e.city EXISTS (SELECT name FROM city c WHERE e.city = c.name AND
name LIKE 'G%');
```

#### *Kosten:*

| | QUERY PLAN text |
|---|---|
| 1 | Hash Semi Join (cost=64.48..986.10 rows=2421 width=105) (actual time=0.970..18.130 rows=1016 loops=1) |

### Ergebnisinterpretation:

IN eignet sich, wie oben beschrieben, nur dann, wenn die meisten Filterkriterien in der Subquery sind. Wenn jedoch die meisten Filterkriterien in der Hauptquery sind, eignet sich die Nutzung von EXISTS. In den obigen Abfragen kann man eindeutig erkennen, dass EXITS die Kosten sehr stark reduziert. Mit EXISTS werden im Allgemeinen Subqueries angegeben, die testen, ob Zeilen vorhanden sind.

## 4.5 Nummer 5 [1]

### Erklärung

Nutzen Sie EXISTS anstand DISTINCT, wenn Sie Joins ihn ihere Query haben.

(One-To-Many-Relation)

### Nicht-optimierte Variante:

#### *Abfrage:*

```
SELECT DISTINCT firstname FROM employees e, city c WHERE e.city = c.name
AND c.name LIKE 'G%';
```

#### *Kosten:*

| | QUERY PLAN text |
|---|---|
| 1 | HashAggregate (cost=1077.14..1088.80 rows=1166 width=6) (actual time=20.665..20.818 rows=683 loops=1) |

### Optimierte Variante:

#### *Abfrage:*

```
SELECT firstname FROM employees e
WHERE e.city EXISTS (SELECT name FROM city c WHERE e.city = c.name AND
name LIKE 'G%');
```

#### *Kosten:*

| | QUERY PLAN text |
|---|---|
| 1 | Hash Semi Join (cost=64.48..986.10 rows=2421 width=6) (actual time=1.012..18.277 rows=1016 loops=1) |

### Ergebnisinterpretation:

Bei Tabellen mit 1:N Beziehung sollte man EXISTS anstatt DISTINCT verwenden, da EXISTS die Kosten um einiges reduziert. Man erkennt auch, dass die Anzahl der Zeilen unterschiedlich sind, da EXISTS doppelte nicht zusammenfasst.

## 4.6 Nummer 6 [1]

### Erklärung

Immer UNION ALL anstatt von UNION benutzen.

### Nicht-optimierte Variante:

#### *Abfrage:*

```
SELECT * FROM employees WHERE firstname LIKE ‘A%’ UNION SELECT *
FROM employees WHERE firstname LIKE 'B%‘;
```

#### *Kosten:*

| | QUERY PLAN text |
|---|---|
| 1 | HashAggregate (cost=1912.94..1953.83 rows=4089 width=105) (actual time=13.350..14.424 rows=4180 loops=1) |

### Optimierte Variante:

#### *Abfrage:*

```
SELECT * FROM employees WHERE firstname LIKE ‘A%’ UNION ALL SELECT *
FROM employees WHERE firstname LIKE 'B%‘;
```

#### *Kosten:*

| | QUERY PLAN text |
|---|---|
| 1 | Append (cost=0.00..1810.72 rows=4089 width=105) (actual time=0.023..7.942 rows=4180 loops=1) |

### Ergebnisinterpretation:

UNION ist langsamer als UNION ALL, da in UNION die DISTINCT-Funktion enthalten ist, welche die Abfrage eindeutig langsamer macht. UNION ALL beinhaltet diese Funktion nicht und ist aus diesem Grund viel schneller. Nachteil von UNION ALL ist also, dass doppelt vorkommende Zeilen nicht zusammengefasst werden.

## 4.7 Nummer 7 [1]

### Erklärung

Man sollte aufpassen, wenn man in WHERE-Clausen Conditions benutzt.

### Nicht-optimierte Variante:

#### *Abfrage:*

```
SELECT * FROM employees WHERE firstname != '2007-01-01';
```

#### *Kosten:*

| | QUERY PLAN text |
|---|---|
| 1 | Seq Scan on employees (cost=0.00..884.91 rows=29825 width=105) (actual time=0.009..5.846 rows=29826 loops=1) |

### Optimierte Variante:

#### *Abfrage:*

```
SELECT * FROM employees WHERE hiredate > '2007-01-01';
```

#### *Kosten:*

| | QUERY PLAN text |
|---|---|
| 1 | Seq Scan on employees (cost=0.00..884.91 rows=8984 width=105) (actual time=0.010..5.207 rows=8981 loops=1) |

### Ergebnisinterpretation:

Bei diesem Beispiel kann man eindeutig erkennen, dass die Nutzung des „>" Operators um einiges schneller ist als „!=".

## Beispiel 1 [1]

### *Erklärung*

Man sollte aufpassen, wenn man in WHERE-Clausen Conditions benutzt.

### *Nicht-optimierte Variante:*

**Abfrage:**

```
SELECT * FROM employees WHERE firstname != '2007-01-01';
```

**Kosten:**

| | QUERY PLAN text |
|---|---|
| 1 | Seq Scan on employees (cost=0.00..884.91 rows=29825 width=105) (actual time=0.009..5.846 rows=29826 loops=1) |

### *Optimierte Variante:*

**Abfrage:**

```
SELECT * FROM employees WHERE hiredate > '2007-01-01';
```

**Kosten:**

| | QUERY PLAN text |
|---|---|
| 1 | Seq Scan on employees (cost=0.00..884.91 rows=8984 width=105) (actual time=0.010..5.207 rows=8981 loops=1) |

### *Ergebnisinterpretation:*

Die LIKE-Funktion ist schneller und effizienter als die SUBSTR-Funktion, wie man am obigen Beispiel klar erkennen kann. Daher sollte man eher LIKE benutzen anstatt SUBSTR.

## Beispiel 2 [1]

### *Erklärung*

Man sollte aufpassen, wenn man in WHERE-Clausen Conditions benutzt.

### *Nicht-optimierte Variante:*

#### Abfrage

```
SELECT * FROM employees WHERE SUBSTR(firstname, 1, 4) = 'Russ';
```

#### Kosten

| | QUERY PLAN text |
|---|---|
| 1 | Seq Scan on employees (cost=0.00..959.50 rows=149 width=105) (actual time=0.018..6.920 rows=33 loops=1) |

### *Optimierte Variante*

#### Abfrage

```
SELECT * FROM employees WHERE firstname LIKE 'Russ;
```

#### Kosten

| | QUERY PLAN text |
|---|---|
| 1 | Seq Scan on employees (cost=0.00..884.91 rows=8984 width=105) (actual time=0.010..5.207 rows=8981 loops=1) |

### *Ergebnisinterpretation*

Die Nutzung der BETWEEN-Funktion erspart in erster Linie die Schreibarbeit und ist viel schneller als die Nicht-optimierte Variante.

## Beispiel 3 [1]

### *Erklärung*

Man sollte aufpassen, wenn man in WHERE-Clausen Conditions benutzt.

### *Nicht-optimierte Variante:*

#### Abfrage

```
SELECT * FROM employees WHERE hiredate >= '2000-01-01' AND hiredate
<= '2006-01-01';
```

#### Kosten

| | QUERY PLAN text |
|---|---|
| 1 | Seq Scan on employees (cost=0.00..959.50 rows=17860 width=105) (actual time=0.011..6.921 rows=17855 loops=1) |

### *Optimierte Variante*

#### Abfrage

```
SELECT * FROM employees WHERE hiredate >= '2000-01-01' AND
 <= '2006-01-01';
```

#### Kosten

| | QUERY PLAN text |
|---|---|
| 1 | Seq Scan on employees (cost=0.00..959.50 rows=17860 width=105) (actual time=0.009..6.947 rows=17855 loops=1) |

### *Ergebnisinterpretation*

Das getrennte Abfragen von firstname und lastname benötigt wesentlich weniger Zeilen als wie das gemeinsame Abfragen. Es ist somit viel effektiver und schneller.

## Beispiel 4 [1]

### *Erklärung*

Man sollte aufpassen, wenn man in WHERE-Clausen Conditions benutzt.

### *Nicht-optimierte Variante:*

#### Abfrage

```
SELECT * FROM employees WHERE firstname  || lastname = 'EdwardJones';
```

#### Kosten

| | QUERY PLAN text |
|---|---|
| 1 | Seq Scan on employees  (cost=0.00..959.50 rows=149 width=105) (actual time=0.104..6.289 rows=1 loops=1) |

### *Optimierte Variante*

#### Abfrage

```
SELECT * FROM employees WHERE firstname = 'Edward' AND lastname =
'Jones';
```

Kosten

| | QUERY PLAN text |
|---|---|
| 1 | Seq Scan on employees  (cost=0.00..959.50 rows=1 width=105) (actual time=0.077..4.485 rows=1 loops=1) |

### *Ergebnisinterpretation*

Das getrennte Abfragen von firstname und lastname benötigt wesentlich weniger Zeilen als wie das gemeinsame Abfragen. Es ist somit viel effektiver und schneller.

## Beispiel 5 [1]

### *Erklärung*

Man sollte aufpassen, wenn man in WHERE-Clausen Conditions benutzt.

### *Nicht-optimierte Variante:*

#### Abfrage

```
SELECT * FROM employees WHERE area + 10000 > 40000;
```

#### Kosten

| | QUERY PLAN text |
|---|---|
| 1 | Seq Scan on desert (cost=0.00..1.94 rows=21 width=35) (actual time=0.012..0.026 rows=55 loops=1) |

### *Optimierte Variante*

#### Abfrage

```
SELECT * FROM employees WHERE area > 30000
```

Kosten

| | QUERY PLAN text |
|---|---|
| 1 | Seq Scan on desert (cost=0.00..1.79 rows=56 width=35) (actual time=0.010..0.024 rows=55 loops=1) |

### *Ergebnisinterpretation*

Das Durchführen von Zwischenrechnungen, wie man in der Nicht-optimierten Variante sieht, verlangsamt die Laufzeit und erhöht die Kosten der Query.

## 4.8 Nummer 8 [1]

### Erklärung

Use DECODE to avoid the scanning of same rows or joining the same table repetitively. DECODE can also be made used in place of GROUP BY or ORDER BY clause.

SELECT * FROM employees WHERE name LIKE 'Ramesh%' and location = 'Bangalore';

#### *Instead of*

SELECT DECODE (location, 'Bangalore', ID, NULL) id FROM employees WHERE name LIKE 'Ramesh%';

### Nicht-optimierte Variante

#### *Abfrage*

SELECT * FROM employees WHERE firstname LIKE 'Edw%' and city = 'Murmansk';

#### *Kosten*

| | QUERY PLAN text |
|---|---|
| 1 | Seq Scan on employees (cost=0.00..959.50 rows=1 width=105) (actual time=0.020..6.145 rows=1 loops=1) |

### Optimierte Variante

#### *Abfrage*

SELECT DECODE (city, 'Murmansk', 'M') FROM employees WHERE firstname LIKE 'Edw%';

#### *Kosten:*

```
ERROR:  function decode(character varying, unknown, unknown) does not exist
LINE 1: SELECT DECODE(city, 'Murmansk', 'M')
               ^
HINT:  No function matches the given name and argument types. You might need to add explicit type casts.
********** Error **********

ERROR: function decode(character varying, unknown, unknown) does not exist
SQL state: 42883
Hint: No function matches the given name and argument types. You might need to add explicit type casts.
Character: 8
```

### Ergebnisinterpretation

Aufgrund eines Syntax-Fehlers, wie oben ersichtlich, konnte ich die DECODE-Funktion nicht anwenden. Mittels DECODE sollte das Scannen von gleichen Zeilen oder der Eintritt in die gleiche Tabelle vermieden werden, was ich in diesem Beispiel nicht beweisen konnte.

## 4.9 Nummer 9 [1]

### *Erklärung*

To store large binary objects, first place them in the file system and add the file path in the database.

### *Ergebnisinterpretation:*

Da ich nicht wusste, wie ich diesen Tipp anwenden soll, musste ich diesen Punkt auslassen.

# 4.10 Nummer 10 [1]

## Erklärung

To write queries which provide efficient performance follow the general SQL standard rules.

- Use single case for all SQL verbs
- Begin all SQL verbs on a new line
- Separate all words with a single space
- Right or left aligning verbs within the initial SQL verb

## Ergebnisinterpretation:

Mit Hilfe dieser SQL-Standardregeln kann man beim Implementieren der Abfragen die Übersicht beibehalten. Hierzu ist kein Vergleich notwendig, da es um die Schreibweise von SQL-Queries unter Beachtung der Standardregeln geht.

## 4.11 Nummer 11 [2]

### Erklärung

SQL SELECT only the columns needed, avoid using SELECT *. First, for each column that you do not need every SQL Server performs additional work to retrieve and return them to the client, and the second volume of data exchanged between the client and SQL Server increases unnecessary.

### Nicht-optimierte Variante:

#### *Abfrage:*

```
SELECT * FROM employees;
```

#### *Kosten:*

| | QUERY PLAN text |
|---|---|
| 1 | Seq Scan on employees (cost=0.00..810.33 rows=29833 width=105) (actual time=0.008..2.398 rows=29833 loops=1) |

### Optimierte Variante:

#### *Abfrage:*

```
SELECT firstname, lastname FROM employees;
```

#### *Kosten:*

| | QUERY PLAN text |
|---|---|
| 1 | Seq Scan on employees (cost=0.00..810.33 rows=29833 width=13) (actual time=0.007..3.826 rows=29833 loops=1) |

### Ergebnisinterpretation:

Bei der optimierten Variante werden nicht alle Spalten, sondern nur die benötigten Spalten ausgelesen, wodurch sehr viel Zeit gespart wird aufgrund der kleineren „width“. Das heißt, es muss weniger ausgelesen werden. (Identisch zu Tipp 1 vom 1.Tutorial)

## 4.12 Nummer 12 [2]

### Erklärung

SQL SELECT only the rows needed. The less rows retrieved, the faster the SQL query will run.

### Nicht-optimierte Variante:

#### *Abfrage:*

```
SELECT firstname, lastname FROM employees;
```

#### *Kosten:*

| | QUERY PLAN text |
|---|---|
| 1 | Seq Scan on employees (cost=0.00..810.33 rows=29833 width=13) (actual time=0.012..3.664 rows=29833 loops=1) |

### Optimierte Variante:

#### *Abfrage:*

```
SELECT firstname, lastname FROM employees WHERE firstname LIKE 'Edw%';
```

#### *Kosten:*

| | QUERY PLAN text |
|---|---|
| 1 | Seq Scan on employees (cost=0.00..884.91 rows=3 width=13) (actual time=0.012..3.656 rows=26 loops=1) |

### Ergebnisinterpretation:

Durch das Reduzieren des Ergebnisses auf nur benötigte Zeilen wurde die Laufzeit der Query verkürzt. Je weniger Zeilen abgerufen werden, umso schneller die SQL-Query ausgeführt.

## 4.13 Nummer 13 [2]

### Erklärung

Prune SQL SELECT lists. Every column that is SELECTed consumes resources for processing. There are several areas that can be examined to determine if column selection is really necessary.

### Nicht-optimierte Variante:

#### *Abfrage*

```
WHERE (COL = 'X')

SELECT firstname, lastname FROM employees WHERE firstname = 'Tad';
```

#### *Kosten:*

| | QUERY PLAN text |
|---|---|
| 1 | Seq Scan on employees (cost=0.00..810.33 rows=29833 width=13) (actual time=0.012..3.664 rows=29833 loops=1) |

### Optimierte Variante:

#### *Abfrage*

```
SELECT lastname FROM employees WHERE firstname = 'Tad';
```

#### *Kosten:*

| | QUERY PLAN text |
|---|---|
| 1 | Seq Scan on employees (cost=0.00..884.91 rows=3 width=13) (actual time=0.012..3.656 rows=26 loops=1) |

### Ergebnisinterpretation:

Bei einer Filterung nach einem bestimmten Wert in einer WHERE-Klausel muss nach dem SELECT nicht zusätzlich der Wert, nach dem es gefiltert wird, ausgegeben werden, da der gefilterte Wert logischerweise bei allen Zeilen gleich ist. Hiermit erspart man viel Zeit.

## 4.14 Nummer 14 [2]

### Erklärung

When you create a new table always create a unique clustered index belong to it, possibly it is a numeric type.

### Ergebnisinterpretation:

Das Beifügen eines Indexes bei der Generierung einer Tabelle, beschleunigt das Abfragen dieser Tabelle sehr stark, da der Zugriff auf Indizes schneller erfolgt als wenn man nach bestimmten Spalten abfragt.

## 4.15 Nummer 15 [2]

### Erklärung

Use SQL JOIN instead of subqueries. As a programmer, subqueries are something that you can be tempted to use and abuse. Subqueries, as show below, can be very useful:

SELECT a.id,(SELECT MAX(created) FROM posts

WHERE author_id = a.id) AS latest_post FROM authors a;

Although subqueries are useful, they often can be replaced by a join, which is definitely faster to execute.

SELECT a.id, MAX(p.created) AS latest_post FROM authors a

INNER JOIN posts p ON (a.id = p.author_id) GROUP BY a.id;

### Nicht-optimierte Variante:

#### *Abfrage*

SELECT name FROM city

WHERE country = (SELECT code FROM country WHERE name = 'Greece');

#### *Kosten:*

| | QUERY PLAN text |
|---|---|
| 1 | Seq Scan on city (cost=5.97..68.86 rows=13 width=9) (actual time=0.041..0.401 rows=16 loops=1) |

### Optimierte Variante:

#### *Abfrage*

SELECT c.name FROM city c INNER JOIN country co on c.country = co.code

WHERE co.name = 'Greece';

#### *Kosten:*

| | QUERY PLAN text |
|---|---|
| 1 | Hash Join (cost=5.99..72.89 rows=13 width=9) (actual time=0.046..0.821 rows=16 loops=1) |

### Ergebnisinterpretation:

Die Verwendung von INNER JOIN sollte die Laufzeit normalerweise verkürzen, was in diesem Fall genau umgekehrt ist. Nach meiner Annahme, werden JOINS bei größeren Datenmengen schneller und effizienter sein als SUBSELECTS.

## 4.16 Nummer 16 [2]

### Erklärung

The following example use the OR statement to get the result:

SELECT * FROM a,b WHERE a.p = b.q OR a.x = b.y;

The UNION statement allows you to combine the result sets of 2 or more select queries. The following example will return the same result that the above query gets, but it will be faster.

SELECT * FROM a,b WHERE a.p = b.q UNION SELECT * FROM a, b WHERE a.x = b.y;

#### *Abfrage*

SELECT * FROM country, city WHERE country.code = city.country OR city.name = country.province;

#### *Kosten:*

| | QUERY PLAN text |
|---|---|
| 1 | Nested Loop (cost=24.65..6906.55 rows=3351 width=84) (actual time=0.208..34.545 rows=3115 loops=1) |

### Optimierte Variante:

#### *Abfrage*

SELECT * FROM country, city WHERE country.code = city.country

UNION SELECT * FROM country, city WHERE city.name = country.province;

#### *Kosten*

| | QUERY PLAN text |
|---|---|
| 1 | HashAggregate (cost=329.53..363.05 rows=3352 width=84) (actual time=4.973..5.672 rows=3115 loops=1) |

### Ergebnisinterpretation:

Man sollte UNION anstatt einer WHERE-Klausel mit OR bevorzugen, da er performanter in WORST-Case Fällen ist. Nachteil von UNION ist, dass es viel langsamer ist, wie man es oben eindeutig erkennen kann.

## 4.17 Nummer 17 [2]

### Erklärung

Keep your clustered index small. One thing you need to consider when determining where to put your clustered index is how big the key for that index will be. The problem here is that the key to the clustered index is also used as the key for every non-clustered index in the table. So if you have a large clustered index on a table with a decent number of rows, the size could blow out significantly. In the case where there is no clustered index on a table, this could be just as bad, because it will use the row pointer, which is 8 bytes per row.

### Ergebnisinterpretation:

Die Performance der Tabellen hängt von der Größe des Indexes ab. Daher sollte man den Index so klein wie möglich halten, da je größer der Index, desto langsamer ist die Tabelle.

## 4.18 Nummer 18 [2]

### Erklärung

Avoid cursors. A bit of a no-brainer. Cursors are less performing because every FETCH statement executed is equivalent to another SQL SELECT Statement execution that returns a single row. The optimizer can’t optimize a CURSOR statement, instead optimizing the queries within each execution of the cursor loop, which is undesirable. Given that most CURSOR statements can be re-written using set logic, they should generally be avoided.

### Ergebnisinterpretation:

Die Nutzung von Cursor sollte man wenn möglich immer vermeiden, da Cursor sehr langsam sind. Mittels angepassten SQL-Anweisungen können Cursor umgegangen werden, was performanter ist.

## 4.19 Nummer 19 [2]

### Erklärung

Use computed columns Computed columns are derived from other columns in a table. By creating and indexing a computed column, you can turn what would otherwise be a scan into a seek. For example, if you needed to calculate SalesPrice and you had a Quantity and UnitPrice column, multiplying them in the SQL inline would cause a table scan as it multiplied the two columns together for every single row. Create a computed column called SalesPrice, then index it, and the query optimizer will no longer need to retrieve the UnitPrice and Quantity data and do a calculation – it’s already done.

### Ergebnisinterpretation:

Da „Regular expressions“ von PostgreSQL unterstützt werden, muss man nicht immer die LIKE- oder NOT LIKE-Funktion benutzen.

## 4.20 Nummer 20 [2]

### Erklärung

Pattern matching: there is always the nasty LIKE and NOT LIKE, but Postgres supports regular expressions too.

### Ergebnisinterpretation:

Da „Regular expressions" von PostgreSQL unterstützt werden, muss man nicht immer die LIKE- oder NOT LIKE-Funktion benutzen.

# 5 Aufwandsabschätzung

| Arbeit | Stundenabschätzung |
|---|---|
| Implementieren der in der Aufgabenstellung als Link zur Verfügung gestellten Testdaten in die neu angelegte Datenbank „World" in PostgreSQL | 1h |
| SQL Tuning/SQL Optimization Techniques (Link 1 von Beginner SQL Tutorial) durchführen und dokumentieren | 3h |
| SQL Tutorial Tips (Link 2 von Beginner SQL Tutorial) durchführen und dokumentieren | 4h |
| Fertigstellen der restlichen Dokumentation | 2h |

# 6 Technologiebeschreibung

Für diese Aufgabe wird eine PostgreSQL-Datenbank verwendet. In PostgreSQL befindet sich die Datenbank "worldfactbook". Nach der Implementierung der Testdaten in die Datenbank, können die SQL-Queries ausgeführt und getestet werden.

# 7 Arbeitsdurchführung

SQL-Anweisungen werden verwendet, um Daten aus der Datenbank abzurufen. Wir können dieselben Ergebnisse mit verschiedenen SQL-Abfragen erzielen. Der Einsatz der besten Abfrage ist aber wichtig, wenn die Leistung betrachtet wird. Man benötigt also die Optimierung von SQL-Abfragen basierend auf die Anforderungen.

# 8 Quellen


| INDEX | Quelltitel + Information |
|---|---|
| [1] | Kapitel: 1<br>Quelle: http://beginner-sql-tutorial.com/sql-query-tuning.htm<br>zuletzt abgerufen am: 18.05.2016 |
| [2] | Kapitel: 2<br>Quelle: http://beginner-sql-tutorial.com/sql-tutorial-tips.htm<br>zuletzt abgerufen am: 18.05.2016 |
| [3] | Kapitel: 3<br>Quelle: https://www.pgadmin.org/docs/1.4/query.html<br>zuletzt abgerufen am: 18.05.2016 |
| [4] | Kapitel: 4<br>Quelle: https://en.wikipedia.org/wiki/Query_plan<br>zuletzt abgerufen am: 18.05.2016 |