Assignment 2

Query Tuning

Database Tuning

Group Name 6
Pape David, 01634454
Vecek Filip, 11700962
Paulitsch Matthias, 01652394

April 4, 2019

Creating Tables and Indexes

```
DROP TABLE IF EXISTS Auth;
DROP TABLE IF EXISTS Publ;
DROP TABLE IF EXISTS Students;
DROP TABLE IF EXISTS Techdept;
DROP TABLE IF EXISTS Persons;
DROP TABLE IF EXISTS Employees;

CREATE TABLE Employees(
    ssnum char(50)
    name char(50)
    manager char(50)
    dept char(50)
    salary float8
    numfriends int
    PRIMARY KEY(ssnum, name))

CREATE TABLE Students(
    ssnum char(50)
    name char(50)
    course char(50),
    grade int,
    PRIMARY KEY(ssnum, name))

CREATE TABLE Techdept(
    dept char(50) PRIMARY KEY,
    manager char(50),
    location char(50))
```

Nach dem Befüllen der Tabellen haben wir mit folgenden Befehlen Indizes erstellt:

```
CREATE INDEX employees_ssnum ON employees (ssnum);
CREATE INDEX students_ssnum ON students (ssnum);
CREATE INDEX techdept_dept ON techdept (dept);
CLUSTER students USING students_ssnum;
CLUSTER employees USING employees_ssnum;
```

```
CLUSTER techdept USING techdept_dept;
ANALYZE;
```

Populating the Tables

Zum Füllen der Tables haben wir eine JavaScript Library namens faker.js verwendet, welche die Generierung von Zufallsdaten vereinfacht. Damit haben wir die Dateien employees.tsv, students.tsv und techdept.tsv erstellt und mit den oben genannten Typen entsprechenden Zufallsdaten gefüllt. Wir haben ausserdem sichergestellt, dass die Daten sinnvolle Zusammenhänge haben, z.B. dass eine Überlappung zwischen `Students` und `Employees` besteht. Weiters haben wir eindeutige Werte für Primärschlüssel erzeugt. Schlussendlich haben wir die Relationen via copy-Statement befüllt. [1]

Queries

Query 1

Original Query Für diese Query haben wir Postgres mit folgendem Kommando gezwungen, Merge-Join zu nutzen:

```
set enable_hashjoin = off;
```

Ohne diese Flag würde Postgres sowohl für die originale als auch die umgeschriebene Query einen Hash-Join einsetzen, durch den die originale Query schneller wäre.

Die originale Query ist ein Join:

```
SELECT Employees.ssnum
FROM Employees, Students
WHERE Employees.name = Students.name
```

Rewritten Query Give the rewritten query.

```
SELECT Employees.ssnum
FROM Employees, Students
WHERE Employees.ssnum = Students.ssnum
```

Evaluation of the Execution Plans Give the execution plan of the original query.

```
                                QUERY PLAN
 Merge Join  (cost=32946.63..35196.11 rows=99970 width=51) (actual
     time=4177.745..5548.078 rows=6058 loops=1)
   Merge Cond: (employees.name = students.name)
   ->  Sort  (cost=17900.32..18150.32 rows=100000 width=102) (
      actual time=2140.678..2635.763 rows=99997 loops=1)
         Sort Key: employees.name
         Sort Method: external merge  Disk: 10976kB
         ->  Seq Scan on employees  (cost=0.00..4125.00 rows
            =100000 width=102) (actual time=0.021..23.613 rows
            =100000 loops=1)
   ->  Materialize  (cost=15046.31..15546.16 rows=99970 width=51)
      (actual time=2036.734..2565.539 rows=99970 loops=1)
```

[1] https://github.com/Marak/faker.js

```
         ->  Sort  (cost=15046.31..15296.24 rows=99970 width=51) (
             actual time=2036.730..2536.496 rows=99970 loops=1)
                Sort Key: students.name
                Sort Method: external merge  Disk: 5992kB
                ->  Seq Scan on students  (cost=0.00..3324.70 rows
                   =99970 width=51) (actual time=0.016..19.640 rows
                   =99970 loops=1)
 Planning Time: 0.667 ms
 Execution Time: 5563.180 ms
(13 rows)
```

Give an interpretation of the execution plan, i.e., describe how the original query is evaluated.

Die Datenbank verwendet einen Merge Join. Zuvor sortiert sie die Daten mittels eines externen Merge Sortes nach dem key `name` in `Employees`, damit man überhaupt einen Merge Join anwenden kann.

Give the execution plan of the rewritten query.

```
                          QUERY PLAN
 Merge Join  (cost=79.51..17548.01 rows=99970 width=51) (actual
    time=0.158..482.966 rows=99970 loops=1)
   Merge Cond: (employees.ssnum = students.ssnum)
   ->  Index Only Scan using employees_ssnum on employees  (cost
      =0.42..8424.42 rows=100000 width=51) (actual time
      =0.079..98.079 rows=100000 loops=1)
         Heap Fetches: 100000
   ->  Index Only Scan using students_ssnum on students  (cost
      =0.42..7623.97 rows=99970 width=51) (actual time
      =0.071..81.086 rows=99970 loops=1)
         Heap Fetches: 99970
 Planning Time: 2.370 ms
 Execution Time: 492.798 ms
(8 rows)
```

Give an interpretation of the execution plan, i.e., describe how the rewritten query is evaluated.

Hierbei wird ebenso ein Merge Join auf die Merge Condition `employees.ssnum` = `students.ssnum` durchgeführt, wobei es jedoch nicht nötig ist die Relation vorher zu sortieren.

Discuss, how the execution plan changed between the original and the rewritten query. In both the interpretation of the query plans and the discussion focus on the crucial parts, i.e., the parts of the query plans that cause major runtime differences.

Der grosse Unterschied zwischen den beiden Ausführungsplänen liegt darin, dass bei der zweiten Query nicht vor dem Merge Join sortiert werden muss, da bereits ein clustered Index auf `ssnum` besteht.

Experiment Give the runtimes of the original and the rewritten query.

	Runtime [sec]
Original query	Planning Time: 0.000667 Execution Time: 5.563180
Rewritten query	Planning Time: 0.002370 m Execution Time: 0.492798

Discuss, why the rewritten query is (or is not) faster than the original query.

Die umgeschrieben Query ist deswegen schneller, da der clustered Index auf `ssnum` verwendet wird und daher kein externer Merge Sort durchgeführt werden muss.

Query 2

Original Query Die originale Query soll die SSN des Mitarbeiters mit dem höchsten Gehalt pro Department ausgeben.

```
SELECT ssnum
FROM employees e1
WHERE salary = (
    SELECT MAX(salary)
    FROM employees e2
    WHERE e2.dept = e1.dept
);
```

Rewritten Query Wir nutzen eine temporäre Tabelle:

```
SELECT MAX(salary) as bigsalary, dept INTO temp
FROM employees
GROUP BY dept;

SELECT ssnum
FROM employees, temp
WHERE salary = bigsalary AND employees.dept = temp.dept;
```

Evaluation of the Execution Plans Originaler Query-Plan:

```
                                    QUERY PLAN
 Seq Scan on employees e1  (cost=0.00..437755375.00 rows=500 width
    =51)
   Filter: (salary = (SubPlan 1))
   SubPlan 1
     ->  Aggregate  (cost=4377.50..4377.51 rows=1 width=8)
           ->  Seq Scan on employees e2  (cost=0.00..4375.00 rows
              =1000 width=8)
                 Filter: (dept = e1.dept)
(6 rows)
```

Give an interpretation of the execution plan, i.e., describe how the original query is evaluated.

Es wird ein sequentieller Scan auf alle Daten von `Employees` ausgeführt. Dabei wird bei jedem einzelnen employee die `salary` durch eine Subquery verglichen. Die Subquery läuft erneut durch alle Einträge von `Employees` und nimmt die Daten mit der höchsten `salary`, die das gleiche `dept` hat wie der Eintrag der äusseren Query. Somit wird für jeden Eintrag von `Employees` die ganze Relation erneut durchsucht(Laufzeit von n^2).

Give the execution plan of the rewritten query.

1)

```
                                QUERY PLAN
 HashAggregate  (cost=4625.00..4626.00 rows=100 width=59)
   Group Key: dept
   ->  Seq Scan on employees  (cost=0.00..4125.00 rows=100000
      width=59)
(3 rows)
```

2)

```
                                        QUERY PLAN

 Hash Join  (cost=4.50..4879.52 rows=2 width=51)
   Hash Cond: ((employees.salary = temp.bigsalary) AND (employees.
      dept = temp.dept))
   ->  Seq Scan on employees  (cost=0.00..4125.00 rows=100000
      width=110)
   ->  Hash  (cost=3.00..3.00 rows=100 width=59)
         ->  Seq Scan on temp  (cost=0.00..3.00 rows=100 width=59)
(5 rows)
```

Give an interpretation of the execution plan, i.e., describe how the rewritten query is evaluated.

Zunächst erstellen wir eine temporäre Table mit den maximalen `salary` Werten und dem dazugehörigen `dept` durch einen sequentiellen Scan. Dann wird mit einem Hash-Join die `ssnum` Werte von `Employees` durch Vergleichen mit der temporären Table herausgesucht. Dabei werden die `salary` und `dept` Werte miteinander verglichen.

Discuss, how the execution plan changed between the original and the rewritten query. In both the interpretation of the query plans and the discussion focus on the crucial parts, i.e., the parts of the query plans that cause major runtime differences.

Der grösste Unterschied liegt in der Verwendung der Subquery in der originalen Query, die zur quadratischen Laufzeit fhrt. In der verbesserten Query erstellen wir somit eine deutlich kleinere Relation für den Vergleich. Auch ist die mögliche Verwendung des Hash-Joins in der verbesserten Query ein Vorteil.

Experiment Give the runtimes of the original and the rewritten query.

	Runtime [sec]
Original query	8011090.635 ms (02:13:31.091)
Rewritten query	107.031 ms + 149.462 ms

Discuss, why the rewritten query is (or is not) faster than the original query.

Die neuen Queries sind weitaus schneller, da in der alten Query die Aggregator-Subquery für jeden einzelnen Wert von `Employees` einmal ausgeführt wird, und diese scannt wiederum die gesamte Relation sequentiell. Die Laufzeit ist daher quadratisch in der Grösse von `Employees`, was natürlich leicht zu schlagen ist; z.B. indem man in einem Durchlauf die höchsten Gehälter findet und die entstehende (kleine) Relation dann mit `Employees` joint, um die zugehörigen `ssnum` Werte zu finden.

Time Spent on this Assignment

Time in hours per person: **4**

References

Tuning von Datenbanksystemen Folien
Database Tuning Principles, Experiments and Troubleshooting Techniques (p.145)