Assignment 2

Query Tuning

Database Tuning

Gruppe K
Brezovic Ivica, 11702570
Demir Cansu, 11700525
Mrazovic Mirna, 11700383

March 26, 2020

Creating Tables and Indexes

SQL statements used to create the tables `Employee`, `Student`, and `Techdept`, and the indexes on the tables:

```
DROP TABLE IF EXISTS "Employee";
CREATE TABLE "Employee" (
    ssnum       INT             NOT NULL    UNIQUE, +
    name        VARCHAR(40)     NOT NULL    UNIQUE, +
    manager     VARCHAR(40),                        +
    dept        VARCHAR(40),                        +
    salary      NUMERIC(8,2)    NOT NULL,           +
    numfriends  SMALLINT        NOT NULL,           +
    PRIMARY KEY (ssnum, name)
);

DROP TABLE IF EXISTS "Student";
CREATE TABLE "Student" (
    ssnum       INT             NOT NULL    UNIQUE, +
    name        VARCHAR(40)     NOT NULL    UNIQUE, +
    course      VARCHAR(40)     NOT NULL,           +
    grade       SMALLINT        NOT NULL,           +
    PRIMARY KEY (ssnum, name)
);

DROP TABLE IF EXISTS "Techdept";
CREATE TABLE "Techdept" (
    dept        VARCHAR(40)     NOT NULL    UNIQUE, +
    manager     VARCHAR(40)     NOT NULL,           +
    location    VARCHAR(40)     NOT NULL,           +
    PRIMARY KEY (dept)
);

CREATE UNIQUE INDEX employee_ssnum_unique_index ON "Employee" (ssnum);
CREATE UNIQUE INDEX employee_name_unique_index ON "Employee" (name);
CREATE INDEX employee_dept_index ON "Employee" (dept);
```

```
CREATE UNIQUE INDEX student_ssnum_unique_index ON "Student" (ssnum);
CREATE UNIQUE INDEX student_name_unique_index ON "Student" (name);

CREATE UNIQUE INDEX techdept_dept_unique_index ON "Techdept" (dept);
```

Populating the Tables

How did you fill the tables? What values did you use? Give a short description of your program.

10 Techdepartments erstellen und in eine ArrayList speichern

- ManagerPool mit 10 random Personen füllen

  ```
  String[] managerPool = {"Mia Wagner", "David Gruber", [...]}
  ```

- Dasselbe mit einem LocationPool: 10 verschiedene Standorte erzeugen

  ```
  String[] locationPool = {"Salzburg", "Wien", [...]}
  ```

- 10 verschiedene Techdepartments erstellen, Manager und Location aus dem Pools zufällig zuordnen und in Arraylist speichern

  ```
  departments.add(new Techdept("Development",
      managerPool[gen.nextInt(10)], locationPool[gen.nextInt(10)]));
  ```

Tabelle: Techdept (10 departments)

- wie bereits oben beschrieben: 10 eindeutige Departments, Locations und Manager erstellen
- und diese dann an das jeweilige Department zuweisen
- Location und Manager kann und wird Duplikate enthalten

  ```
  wr.write(dept.getDept() + "\t" + dept.getManager() +
      "\t" + dept.getLocation() + "\n");

  cm.copyIn("COPY \"Techdept\" FROM stdin",
      new FileInputStream("Techdept.tsv"));
  ```

Tabelle: Student (100k students)

- eindeutige ssnum und name erzeugen zwischen 0 und 99 999
- zufällige Kurse erzeugen
- zufällige Noten zwischen 1 bis 5 festlegen

  ```
  wr_student.write(i + "\t" + UUID.randomUUID().toString() + "\t" +
      UUID.randomUUID().toString() + "\t" +
      ThreadLocalRandom.current().nextInt(1,6) + "\n");

  cm.copyIn("COPY \"Student\" FROM stdin",
      new FileInputStream("Student.tsv"));
  ```

Tabelle: Employee (100k employees)

- ssnum und name: Index von 100 000 bis 199 999, damit die Eindeutigkeit gewährleistet wird

- in 10 Prozent der Fälle wird dem Employee ein Manager und Department zugewiesen
- ansonsten null
- random salary zwischen 2000 und 8000 Eur [1] [randomUUID]
- random numfriends zwischen 2 und 12

```
wr_employee.write(j + "\t" + UUID.randomUUID().toString() + "\t" +
    manager + "\t" + dept "\t" +
    ThreadLocalRandom.current().nextInt(2000,8000) + "\t" +
    ThreadLocalRandom.current().nextInt(2,12) +"\n");

cm.copyIn("COPY \"Employee\" FROM stdin",
    new FileInputStream("Employee.tsv"));
```

Queries

Query 1

Original Query Give the first type of query that might be hard for your database to optimize.
Query type: don't use HAVING where WHERE is enough

```
SELECT AVG(salary) AS avgsalary, dept FROM "Employee"
GROUP BY dept HAVING dept = 'Databases' ;
```

Rewritten Query Give the rewritten query.

```
SELECT AVG(salary) AS avgsalary, dept FROM "Employee"
WHERE dept = 'Databases' GROUP BY dept ;
```

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

```
GroupAggregate  (cost=9.31..389.09 rows=97 width=130)
                (actual time=0.906..0.906 rows=1 loops=1)
  Group Key: dept
  ->  Bitmap Heap Scan on "Employee"  (cost=9.31..387.22 rows=132 width=112)
                                      (actual time=0.155..0.710 rows=1001 loops=1)
        Recheck Cond: ((dept)::text = 'Databases'::text)
        Heap Blocks: exact=675
        ->  Bitmap Index Scan on employee_dept_index
                                      (cost=0.00..9.28 rows=132 width=0)
                                      (actual time=0.090..0.090 rows=1001 loops=1)
              Index Cond: ((dept)::text = 'Databases'::text)
```

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

- Mithilfe von GroupAggregate wird das Ergebnis entsprechend der group-by Klausel gruppiert und sortiert.
- Der Gruppierungsschlüssel ist dept
- Ein einfacher Index Scan lädt einen Tupel-Zeiger nach dem anderen aus dem Index und greift sofort auf das entsprechende Tupel in der Tabelle zu. Ein Bitmap Scan lädt alle Tupel-Zeiger auf einmal aus dem Index, benutzt eine Bitmap-Struktur,

um sie im Hauptspeicher zu sortieren, und lädt die Tabellentupel entsprechend der physischen Speicherreihenfolge. Ein Bitmap Heap Scan liest die Tabellenblöcke in sequentieller Reihenfolge, sodass kein zufälliger Tabellenzugriff/Overhead erzeugt wird (wie beim Index Scan).

- Wenn zu viele Zeilen gespeichert werden, werden nicht alle Blöcke gemerkt. Daher werden die Ergebnisse nochmals getestet, ob sie alle die gewünschte Bedingung erfüllen.
- der Heap Blöcke
- Bitmap Index Scan auf das Ergebnis durchführen
- Index Bedingung: ((dept)::text = 'Databases'::text)

```
GroupAggregate  (cost=9.31..389.09 rows=97 width=130)
                (actual time=0.777..0.777 rows=1 loops=1)
  Group Key: dept
  ->  Bitmap Heap Scan on "Employee"  (cost=9.31..387.22 rows=132 width=112)
                                      (actual time=0.154..0.608 rows=1001 loops=1)
        Recheck Cond: ((dept)::text = 'Databases'::text)
        Heap Blocks: exact=675
        ->  Bitmap Index Scan on employee_dept_index
                                      (cost=0.00..9.28 rows=132 width=0)
                                      (actual time=0.090..0.090 rows=1001 loops=1)
              Index Cond: ((dept)::text = 'Databases'::text)
```

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

- Mithilfe von GroupAggregate wird das Ergebnis entsprechend der group-by Klausel gruppiert und sortiert.
- Der Gruppierungsschlüssel ist dept
- Ein einfacher Index Scan lädt einen Tupel-Zeiger nach dem anderen aus dem Index und greift sofort auf das entsprechende Tupel in der Tabelle zu. Ein Bitmap Scan lädt alle Tupel-Zeiger auf einmal aus dem Index, benutzt eine Bitmap-Struktur, um sie im Hauptspeicher zu sortieren, und lädt die Tabellentupel entsprechend der physischen Speicherreihenfolge. Ein Bitmap Heap Scan liest die Tabellenblöcke in sequentieller Reihenfolge, sodass kein zufälliger Tabellenzugriff/Overhead erzeugt wird (wie beim Index Scan).
- Wenn zu viele Zeilen gespeichert werden, werden nicht alle Blöcke gemerkt. Daher werden die Ergebnisse nochmals getestet, ob sie alle die gewünschte Bedingung erfüllen.
- der Heap Blöcke
- Bitmap Index Scan auf das Ergebnis durchführen
- Index Bedingung: ((dept)::text = 'Databases'::text)

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.

Prinzipiell gibt es bei beiden Anfragen der Queries keinen Unterschied. Auch die Laufzeiten unterscheiden sich kaum. In diesem Fall sieht man, dass die Queries mit HAVING und WHERE ziemlich gleich sind.

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

	Runtime [sec]
Original query	0.967 ms
Rewritten query	0.800 ms

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

Die Queries wurden am Localhost Server ausgeführt. Aufgrund der Tatsache, dass sich die Laufzeiten beider Queries nur minimal unterscheiden, kann man davon ausgehen der Optimizer von PostgreSQL arbeitet so gut, dass in diesem Fall zwischen HAVING und WHERE kaum Unterschied besteht. Dennoch ist die Performance mit der Rewritten-Query etwas besser als die bestehende.

Query 2

Original Query Give the second type of query that might be hard for your database to optimize.

```
SELECT name, grade FROM "Student"
WHERE grade = (SELECT MAX(grade) FROM "Student") ;
```

Rewritten Query Give the rewritten query.

```
SELECT name, grade FROM  "Student"
WHERE grade = '5' ;
```

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

```
Seq Scan on "Student"  (cost=2584.01..5168.01 rows=20000 width=39)
                       (actual time=18.532..30.177 rows=20100 loops=1)
  Filter: (grade = $0)
  Rows Removed by Filter: 79900
  InitPlan 1 (returns $0)
    ->  Aggregate  (cost=2584.00..2584.01 rows=1 width=2)
                   (actual time=18.519..18.520 rows=1 loops=1)
          ->  Seq Scan on "Student" "Student_1"
                      (cost=0.00..2334.00 rows=100000 width=2)
                      (actual time=0.003..9.955 rows=100000 loops=1)
```

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

- Sequentieller Scan auf Student um die Werte zu filtern
- Anwendung von Filter MAX(grade)
- Sequentieller Scan um die SELECT name, grade auszuführen

Give the execution plan of the rewritten query.

```
Seq Scan on "Student"  (cost=0.00..2584.00 rows=19943 width=39)
                       (actual time=0.007..11.962 rows=20100 loops=1)
  Filter: (grade = '5'::smallint)
  Rows Removed by Filter: 79900
```

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

- Anwendung von Filter grade=5
- Sequentieller Scan um die SELECT name, grade auszuführen

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.

In der Rewritten-Query fällt der erste sequentielle Scan um die MAX(grade) zu finden weg, da wir sie bereits auf grade = "5" festgelegt haben. Wir ersparen uns dadurch den Scan über die ganze Tabelle.

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

	Runtime [sec]
Original query	30.982 ms
Rewritten query	12.768 ms

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

Durch das Festlegen der MAX(grade) auf 5 ersparen wir uns den Sequentiellen Scan. Dadurch fällt einiges an Laufzeit weg und die Query wird insgesamt schneller.

Time Spent on this Assignment

Time in hours per person: **13**

References

References

[1] https://www.tutorialspoint.com/