

Database Tuning – Assignment 2

Query Tuning

A2

Baumgartner Dominik, 0920177

Dafir Thomas Samy, 1331483

Schörgenhofer Kevin, 1421082

November 1, 2016

Experimental Data

Creating Tables and Indexes SQL statements used to create the tables Employee, Student, and Techdept, and the indexes on the tables:

```
CREATE TABLE Employee
(
  ssnnum integer,
  name character varying,
  manager character varying,
  dept character varying,
  salary numeric(8,2),
  numfriends smallint,
  CONSTRAINT ssnnumNameE PRIMARY KEY (ssnum, name)
);
CREATE UNIQUE INDEX ssnnumE ON Employee (ssnum);
CREATE UNIQUE INDEX nameE ON Employee (name);
CREATE INDEX deptE ON Employee (dept);
```

```
CREATE TABLE Student
(
  ssnnum integer,
  name character varying,
  course character varying,
  grade smallint,
  CONSTRAINT ssnnumNameS PRIMARY KEY (ssnum, name)
);
CREATE UNIQUE INDEX ssnnumS ON Student (ssnum);
CREATE UNIQUE INDEX nameS ON Student (name);
```

```

CREATE TABLE Techdept
(
dept character varying,
manager character varying,
location character varying,
CONSTRAINT deptT PRIMARY KEY (dept)
);
-- no explicit creation of a unique index necessary, because the primary key
creates implicitly an unique index on dept (according to postgres doc)

```

Populating the Tables The tables were populated using the *copy* command on a tsv-file containing the specified amount of data rows for each table. The files were created with a self-written java program generating values for each entry and inserting them into the corresponding tsv-file. the following values were created:

- Employee: ssnum: int(0...99999), name: random string of length 20, manager and department: randomly chosen from a 2D array containing (department,manager) pairs, salary: random int value (1000...11000), numfriends: random int (50...520)
- Student: ssnum: int (100000...199999), name: random string of length 20, course: randomly generated using a letter followed by a digit (260 subjects), grade: random int (1...5).
- Techdept: File was created by hand due to the small number of entries. Corresponds to the values in the other tables.

Query 1

Original Query

```

SELECT count(ssnum)
FROM Employee e1
WHERE salary > (SELECT AVG(e2.salary)
FROM Employee e2, Techdept
WHERE e2.dept = e1.dept
AND e2.dept = Techdept.dept);

```

Rewritten Query

```

SELECT count(ssnum)
FROM Techdept t, Employee e1,
(SELECT dept, AVG(salary) as avgsalary FROM Employee GROUP BY dept) as e2
WHERE e1.dept = t.dept AND e1.salary > e2.avgsalary AND e1.dept = e2.dept;

```

Evaluation of the Execution Plans Execution plan original query:

```
Aggregate (cost=117285824.33..117285824.34 rows=1 width=4)
-> Seq Scan on employee e1 (cost=0.00..117285741.00 rows=33333 width=4)
    Filter: (salary > (SubPlan 1))
    SubPlan 1
        -> Aggregate (cost=1172.83..1172.84 rows=1 width=5)
            -> Nested Loop (cost=117.23..1159.67 rows=5263 width=5)
                -> Index Only Scan using deptt on techdept
                    (cost=0.15..8.17 rows=1 width=32)
                    Index Cond: (dept = (e1.dept)::text)
                -> Bitmap Heap Scan on employee e2
                    (cost=117.08..1098.87 rows=5263 width=8)
                    Recheck Cond: ((dept)::text = (e1.dept)::text)
                    -> Bitmap Index Scan on depte
                        (cost=0.00..115.77 rows=5263 width=0)
                        Index Cond: ((dept)::text = (e1.dept)::text)
```

The query consists of a subquery and a main query. the main query iterates over each employee in the employee table and checks if salary is larger than the value returned by the subquery. The plan makes visible the execution of the subquery for each entry in the employee table. In the end an aggregate function is executed giving us the number of all employees earning more than the average salary of their respective department. The subquery is executed as follows: First a bitmap index scan is performed on the employee table to get all blocks where employees in the same department as the current employee in the outer query (e1). Next a Bitmap Heap Scan is performed (scan all previously marked blocks) and the selection condition is rechecked on the fly. In the next step an index nested loop join is performed where the index on Techdept.dept is scanned for the every employee in the result of the former step. Finally the average salary of all remaining employees which are all in the same department as e1 is computed. This value is then used for the comparison in the outer query.

Execution plan rewritten query:

```
Aggregate (cost=4847.34..4847.35 rows=1 width=4)
-> Hash Join (cost=2435.89..4764.01 rows=33333 width=4)
    Hash Cond: ((e1.dept)::text = (t.dept)::text)
    Join Filter: (e1.salary > e2.avgsalary)
-> Seq Scan on employee e1 (cost=0.00..1916.00 rows=100000 width=12)
-> Hash (cost=2435.66..2435.66 rows=19 width=67)
    -> Hash Join
        (cost=2416.67..2435.66 rows=19 width=67)
        Hash Cond: ((t.dept)::text = (e2.dept)::text)
        -> Seq Scan on techdept t
            (cost=0.00..16.40 rows=640 width=32)
        -> Hash (cost=2416.43..2416.43 rows=19 width=35)
            -> Subquery Scan on e2
                (cost=2416.00..2416.43 rows=19 width=35)
                -> HashAggregate
                    (cost=2416.00..2416.24 rows=19 width=8)
                    Group Key: employee.dept
                    -> Seq Scan on employee
                        (cost=0.00..1916.00 rows=100000 width=8)
```

Starting from the bottom of the execution plan:

First of all the inner query is executed. For this, a Seq Scan on employee and a grouping function is used, so we get the employees grouped by their department. Afterwards an aggregate function (the average of the salary) gets executed on the grouped employees, which gives us the average salary for every department. Now a Hash Join is used to join the techdepts (outer query) with their average salary from the inner query. The next step is another Hash Join between employee (outer query; Seq Scan for getting the data) and the results of the previous Hash Join, which ensures that every employee is matched with his department + a Join Filter ensures that the salary of the employee (outer query) is higher than the average salary of his department (inner query). Finally the Count aggregate function is applied to the result of the last Hash Join, which returns our requested result.

Runtime The biggest difference between the two queries is the fact that in the original the inner query is executed for every employee whereas in the rewritten one the inner query is executed exactly once and the result is used to in a hash join. This saves huge amounts of time as can be seen in runtime evaluation. Furthermore the rewritten query enables the use of Hash Joins. Apart from that there is no real difference concerning

the efficiency of the techniques uses. Both use some linear heap scanning and both also use the index structures to conduct joins. So the main gain concerning execution time is really achieved through unnesting of the correlated inner query.

	Runtime [ms]
Original query	181654
Rewritten query	95

Query 2

Original Query

```
SELECT DISTINCT ssnnum FROM employee;
```

Rewritten Query

```
SELECT ssnnum FROM employee;
```

Evaluation of the Execution Plans

 Execution plan original query:

```
Unique (cost=0.29..3799.50 rows=100983 width=4)
-> Index Only Scan using ssnnum on employee (cost=0.29..3547.04 rows=100983)
```

The optimizer notices that employee has a unique index on ssnnum, so all unique ssnnums can be obtained simply by scanning only the index since it contains every unique ssnnum in the table.

Execution plan rewritten query:

```
Seq Scan on employee (cost=0.00..1934.83 rows=100983 width=4)
```

There is not much to optimize in this query. A simple sequential scan is conducted on the employee table.

Runtime Both queries return the exact same result since ssnnums are unique. The difference in runtime is due to the fact that conducting an index only scan on the unique index simply requires more block accesses than a sequential scan of the table.

	Runtime [ms]
Original query	103
Rewritten query	72

Time in hours per person: **6 hours**