Assignment 2

Query Tuning

Database Tuning

New Group 8
Frauenschuh Florian, 12109584
Lindner Peter, 12101607
Weilert Alexander, 12119653

April 22, 2024

Creating Tables and Indexes

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

```
CREATE TABLE IF NOT EXISTS Employee (
    ssnum INTEGER PRIMARY KEY,
    name VARCHAR(64) UNIQUE NOT NULL,
    manager VARCHAR(64),
    dept VARCHAR(64),
    salary INTEGER,
    numfriends INTEGER)

CREATE TABLE IF NOT EXISTS Student (
    ssnum INTEGER PRIMARY KEY,
    name VARCHAR(64) UNIQUE NOT NULL,
    course VARCHAR(64),
    grade INTEGER)

CREATE TABLE IF NOT EXISTS Techdept (
    dept VARCHAR(64) PRIMARY KEY,
    manager VARCHAR(64),
    location VARCHAR(64))

CREATE UNIQUE INDEX IF NOT EXISTS idx_ssnum ON Employee(ssnum)
CREATE UNIQUE INDEX IF NOT EXISTS idx_name ON Employee(name)
CREATE INDEX IF NOT EXISTS idx_dept ON Employee(dept)

CREATE UNIQUE INDEX IF NOT EXISTS idx_ssnum ON Student(ssnum)
CREATE UNIQUE INDEX IF NOT EXISTS idx_name ON Student(name)

CREATE UNIQUE INDEX IF NOT EXISTS idx_dept ON Techdept(dept)
```

Populating the Tables

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

We populated the tables by using batch statements, starting with the **Techdept** table. The departments were named `Department_1`, `Department_2`, ..., `Department_10`. Both manager and location were generated by appending a random number between 1 and 10 to either `Name_` or `Location_`. The combinations of department and manager are stored in a HashMap, which we make use of when generating the values for the **Employee** table.

For the **Employee** table the ssnum is an incremental number, and the name is generated the same way as the department names. For every employee, there is a 10% chance that we choose a random combination of manager and department from the HashTable we previously filled to have 10% of employees in a technical department. Both salary and number of friends are randomized numbers in a given domain.

Lastly, the ssnums in the **Student** table were generated by using an incrementing number starting from 80.000, resulting in 20.000 Students also being Employees. The names generated as follows: `Name_80000`, `Name_80001`, ..., `Name_179999`. Finally, both course and grade are also randomized in a given domain.

Queries

Query 1

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

```
SELECT 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 Give the rewritten query.

```
CREATE TEMPORARY TABLE Temp AS (
    SELECT AVG(salary) as avsalary, e2.dept
    FROM Employee e2, Techdept
    WHERE e2.dept = Techdept.dept
    GROUP BY e2.dept)

SELECT ssnum FROM Employee e1, Temp
WHERE salary > avsalary AND e1.dept = Temp.dept
```

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

For this and all following execution plans we consulted the documentation of Postgres [1].

```
Seq Scan on employee e1  (cost=0.00..102386331.93 rows=33333 width=4) (actual
time=102.691..4409.692 rows=4889 loops=1)
  Filter: ((salary)::numeric > (SubPlan 1))
  Rows Removed by Filter: 95111
```

```
  SubPlan 1
    ->  Aggregate  (cost=1023.83..1023.84 rows=1 width=32)
    (actual time=0.043..0.043 rows=1
    loops=100000)
          ->  Nested Loop  (cost=16.19..1021.33 rows=1000 width=4)
          (actual time=0.010..0.040
          rows=100 loops=100000)
                ->  Index Only Scan using techdept_pkey on techdept
                (cost=0.14..8.16 rows=1
                width=146) (actual time=0.001..0.001 rows=0 loops=100000)
                      Index Cond: (dept = (e1.dept)::text)
                      Heap Fetches: 9978
                ->  Bitmap Heap Scan on employee e2
                (cost=16.04..1003.16 rows=1000 width=17)
                (actual time=0.087..0.330 rows=998 loops=9978)
                      Recheck Cond: ((dept)::text = (e1.dept)::text)
                      Heap Blocks: exact=6060425
                      ->  Bitmap Index Scan on idx_dept
                      (cost=0.00..15.79 rows=1000 width=0)
                      (actual time=0.038..0.038 rows=998 loops=9978)
                            Index Cond: ((dept)::text = (e1.dept)::text)
Planning Time: 0.451 ms
JIT:
  Functions: 15
"  Options: Inlining true, Optimization true, Expressions true, Deforming true"
"  Timing: Generation 0.688 ms, Inlining 13.301 ms, Optimization 53.221 ms, Emission 33.917 ms,
Total 101.127 ms"
Execution Time: 4410.725 ms
```

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

For the outer query the **Employee** table (`e1`) is scanned sequentially. Each tupel of this scan gets filtered by the >-operator, based on the result of `SubPlan1`. The subplan itself represents the nested query. To evaluate this subplan, both the **Employee** table (`e2`) and the **Techdept** table are needed. For the **Techdept** table we utilize an index only scan which uses an index on the primary key that Postgres created automatically. Hence, we do not access the data directly since we are able to check the condition (`e1.dept = dept`) by using only the index. The **Employee** table (`e2`) is accessed by using a combination of a bitmap heap scan and a bitmap index scan, on the index **idx_dept** that we created earlier, with respect to the condition `dept = e1.dept`. Here, the bitmap index scan marks the rows that fulfill the condition and are later needed for the bitmap heap scan by providing an efficient bitmap representation of the rows to be retrieved. The results of the scan of both tables are then combined by using a nested loop join. Finally, the aggregate function `average` is applied, which gives us the result of the `SubPlan1`.

Give the execution plan of the rewritten query.

```
HashAggregate  (cost=2717.87..2719.12 rows=100 width=45)
               (actual time=15.557..15.561 rows=10 loops=1)
  Group Key: e2.dept
  Batches: 1  Memory Usage: 24kB
  ->  Hash Join  (cost=13.82..2217.87 rows=100000 width=17)
                 (actual time=0.021..14.252 rows=9978 loops=1)
        Hash Cond: ((e2.dept)::text = (techdept.dept)::text)
```

```
        ->  Seq Scan on employee e2  (cost=0.00..1935.00 rows=100000 width=17)
                                     (actual time=0.004..5.089 rows=100000 loops=1)
        ->  Hash  (cost=11.70..11.70 rows=170 width=146)
                  (actual time=0.008..0.009 rows=10 loops=1)
              Buckets: 1024  Batches: 1  Memory Usage: 9kB
              ->  Seq Scan on techdept  (cost=0.00..11.70 rows=170 width=146)
                                        (actual time=0.003..0.004 rows=10 loops=1)
Planning Time: 0.190 ms
Execution Time: 15.593 ms
```

```
Hash Join  (cost=3771.00..7858.00 rows=65000 width=4)
           (actual time=38.731..42.704 rows=4889 loops=1)
  Hash Cond: ((temp.dept)::text = (e1.dept)::text)
  Join Filter: ((e1.salary)::numeric > temp.avsalary)
  Rows Removed by Join Filter: 5089
  ->  Seq Scan on temp  (cost=0.00..13.90 rows=390 width=178)
                        (actual time=0.013..0.015 rows=10 loops=1)
  ->  Hash  (cost=1935.00..1935.00 rows=100000 width=21)
            (actual time=38.169..38.170 rows=100000 loops=1)
        Buckets: 65536  Batches: 2  Memory Usage: 2942kB
        ->  Seq Scan on employee e1  (cost=0.00..1935.00 rows=100000 width=21)
                                     (actual time=0.013..12.844 rows=100000 loops=1)
Planning Time: 0.686 ms
Execution Time: 42.992 ms
```

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

The first execution plan handles the creation of the temporary table which is needed to efficiently check the average condition later on. As we can see, it starts with an aggregation function which takes the result of a hash join as its input. To check the condition of the hash join, we first sequentially read the **Employee** table (`e2`) and then read the **Techdept** table as well, with a hashing function applied to it, before handing both over to the hash join.

The second execution plan mainly consists of a hash join, which checks the average salary condition. To accomplish this check, we need to sequentially read the Temp table and the **Employee** table (`e1`), where `e1` gets hashed before forwarding it into the hash join.

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.

Instead of using costly nested loop joins and subqueries, the rewritten query makes use of hash joins, which are generally more efficient. Additionally, since we are using a temporary table, we get two query plans.

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

	Runtime PG [sec]	Runtime MDB [sec]
Original query	3.424	0.068
Rewritten query	0.030	0.025

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

The original query is less efficient than the rewritten query. This is because the original query uses a nested loop join, i.e. for each existing tuple in **Employee** `e1`, an additional selection is made for **Employee** `e2` and **Techdept** and only then compared to see if the values also match. The newly written query uses a hash join, i.e. we search for the matching values once in a query, which saves us a lot of time, as one can see from the test results.
Interestingly, the original query is much faster when executed on MariaDB than on Postgres. This leads to a less significant runtime improvement after rewriting the query. This is probably due to the fact that MariaDB uses a more efficient optimization algorithm and therefore, for example, does not repeat every query on `e2` in the nested loop connection, but possibly stores it in the cache and searches for matches on `e1`.

Query 2

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

```
SELECT ssnum FROM Employee
WHERE dept IN (SELECT dept FROM Techdept)
```

Rewritten Query Give the rewritten query.

```
SELECT ssnum FROM Employee, Techdept
WHERE Employee.dept = Techdept.dept
```

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

```
Hash Join  (cost=13.82..2217.87 rows=100000 width=4)
           (actual time=0.046..37.056 rows=9978 loops=1)
  Hash Cond: ((employee.dept)::text = (techdept.dept)::text)
  ->  Seq Scan on employee  (cost=0.00..1935.00 rows=100000 width=17)
                            (actual time=0.009..12.933
  rows=100000 loops=1)
  ->  Hash  (cost=11.70..11.70 rows=170 width=146)
            (actual time=0.018..0.021 rows=10 loops=1)
        Buckets: 1024  Batches: 1  Memory Usage: 9kB
        ->  Seq Scan on techdept  (cost=0.00..11.70 rows=170 width=146)
                                  (actual time=0.006..0.009 rows=10 loops=1)
Planning Time: 0.393 ms
Execution Time: 37.775 ms
```

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

In this execution plan we have a hash join based on the condition `Employee.dept = Techdept.dept`. For its input we read the **Employee** and **Techdept** table sequentially, where we hash the latter beforehand. As we discussed in class, the execution plan does not make use of the `idx_dept` of the **Employee** table that we created earlier [2].

Give the execution plan of the rewritten query.

```
Hash Join  (cost=13.82..2217.87 rows=100000 width=4)
           (actual time=0.031..30.029 rows=9978
```

```
loops=1)
  Hash Cond: ((employee.dept)::text = (techdept.dept)::text)
  ->  Seq Scan on employee  (cost=0.00..1935.00 rows=100000 width=17)
                            (actual time=0.007..10.490 rows=100000 loops=1)
  ->  Hash  (cost=11.70..11.70 rows=170 width=146)
            (actual time=0.012..0.014 rows=10 loops=1)
        Buckets: 1024  Batches: 1  Memory Usage: 9kB
        ->  Seq Scan on techdept  (cost=0.00..11.70 rows=170 width=146)
                                  (actual time=0.004..0.006 rows=10 loops=1)
Planning Time: 0.242 ms
Execution Time: 30.583 ms
```

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

Interestingly, the execution plan of the rewritten query stays the same as the one for the original query. Therefore, the index `idx_dept` which we created on the **Employee** table is not used. Hence, the evaluation of the rewritten query works exactly the same.

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.

As already mentioned, the execution plan did not change through rewriting the query. To our surprise, we still noticed an improvement in the run time while executing the rewritten query. This is most likely due to external factors.

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

	Runtime PG [sec]	Runtime MDB [sec]
Original query	0.011	0.004
Rewritten query	0.009	0.004

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

Since the execution plan stays the same for both the original and rewritten queries, we would expect the runtime to be exactly the same on average.

Time Spent on this Assignment

Time in hours per person:

- Florian Frauenschuh: **7**
- Peter Lindner: **7.5**
- Alexander Weilert: **6.5**

References

[1] PostgreSQL. *Using EXPLAIN*. Accessed: 2023-04-19. 2023. URL: https://www.postgresql.org/docs/current/using-explain.html#USING-EXPLAIN-ANALYZE.

[2] Martin Schäler. *Database Tuning: Query Tuning*. 2024.