Assignment 2

Query Tuning

Database Tuning

Group Name (e.g. A1, B5, B3)
Peter Balint, 12213073
David Ottino, 51841010
Lukas Günter, 12125639

April 7, 2025

Creating Tables and Indexes

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

```
CREATE TABLE Employee (
    ssnum SERIAL PRIMARY KEY,
    name TEXT NOT NULL UNIQUE,
    manager INTEGER,
    dept TEXT,
    salary NUMERIC(10,2),
    numfriends INTEGER,
    FOREIGN KEY (manager) REFERENCES Employee(ssnum) ON DELETE SET NULL
);

CREATE UNIQUE INDEX idx_employee_ssnum ON Employee(ssnum);
CREATE UNIQUE INDEX idx_employee_name ON Employee(name);
CREATE INDEX idx_employee_dept ON Employee(dept);

CREATE TABLE Student (
    ssnum SERIAL PRIMARY KEY,
    name TEXT NOT NULL UNIQUE,
    course TEXT NOT NULL,
    grade CHAR(2)
);

CREATE UNIQUE INDEX idx_student_ssnum ON Student(ssnum);
CREATE UNIQUE INDEX idx_student_name ON Student(name);

CREATE TABLE Techdept (
    dept TEXT PRIMARY KEY,
    manager INTEGER,
    location TEXT,
    FOREIGN KEY (manager) REFERENCES Employee(ssnum) ON DELETE SET NULL
);

CREATE UNIQUE INDEX idx_techdept_dept ON Techdept(dept);
```

In PostgreSQL wird automatisch ein Index erzeugt, sobald eine Spalte mit einem UNIQUE constraint belegt wird. Daher wären die Indizes für ssnum und name nicht nötig, der Vollständigkeit halber werden sie hier trotzdem angegeben.

Populating the Tables

Die Tabellen werden über ein Python Script gefüllt. Wir haben CSV-Dateien als Zieldateien verwendet, um die Tabelleninhalte zu speichern. Die zugrunde liegenden Daten (Mitarbeitende, Studierende, Abteilungen) wurden zufällig generiert, wobei bestimmte Zusammenhänge und Bedingungen berücksichtigt wurden - z.B. Überschneidungen bei Personen, Manager-Zuweisungen und Abteilungsmitgliedschaften.

TechDept

Für jedes technische Department habenb wir einen zufällig generierten Manager aus der bestehenden Mitarbeitermenge ausgewählt.

```python
def generate_tech_departments(filename="techdepartments.csv"):
    with open(filename, "w", newline="") as file:
        writer = csv.writer(file)
        writer.writerow(["dept", "manager", "location"])

        for i in range(1, TOTAL_TECHDEPTS + 1):
            dept = f"TechDept{i}"
            manager = random.randint(1, TOTAL_EMPLOYEES)                    # Managers come from Employee table
            location = random.choice(["New York", "San Francisco", "Chicago", "Austin", "Seattle"])

            writer.writerow([dept, manager, location])

    print(f"Generated {TOTAL_TECHDEPTS} technical departments in {filename}")
```

Figure 1: Script für Erstellung von TechDept

```
tuninglab2=# SELECT * FROM techdept LIMIT 10;
    dept    | manager |   location
------------+---------+---------------
 TechDept1  |   66370 | Chicago
 TechDept2  |   70608 | Chicago
 TechDept3  |   22937 | Austin
 TechDept4  |   15785 | New York
 TechDept5  |   81879 | San Francisco
 TechDept6  |   83217 | Austin
 TechDept7  |   17979 | Seattle
 TechDept8  |   88464 | Chicago
 TechDept9  |    9853 | Chicago
 TechDept10 |   12416 | Seattle
```

Figure 2: Ergebnis TechDept

Students

Für die Studententabelle wurden 100.000 Einträge erzeugt. Jede Person erhielt:

- eine eindeutige Sozialversicherungsnummer
- einen eindeutigen Namen
- einen zufälligen Kurs und zufällige Note

```python
def generate_students(people, filename="students.csv"):
    with open(filename, "w", newline="") as file:
        writer = csv.writer(file)
        writer.writerow(["ssnum", "name", "course", "grade"])

        student_pool = people[-TOTAL_STUDENTS:]                    # Ensures overlap
        for person in student_pool:
            course = random.choice(["Math", "Physics", "CS", "Biology", "History"])
            grade = random.choice(["A", "B", "C", "D", "F"])

            writer.writerow([
                person["ssnum"],
                person["name"],
                course,
                grade
            ])

    print(f"Generated {len(student_pool)} students in {filename}")
```

Figure 3: Script für Erstellung von Students

```
tuninglab2=# SELECT * FROM student LIMIT 10;
 ssnum |      name       | course  | grade
-------+-----------------+---------+-------
 94001 | Jay Perez       | Physics | A
 94002 | Deborah Beard   | Math    | C
 94003 | Arthur Thompson | CS      | F
 94004 | Heather Owen    | Biology | F
 94005 | Nicholas Duran  | CS      | A
 94006 | Anthony Patton  | History | A
 94007 | James Wang      | History | D
 94008 | Allen Alexander | Biology | C
 94009 | Daniel Stuart   | Physics | F
 94010 | Warren Carney   | CS      | C
(10 rows)
```

Figure 4: Auszug aus Students

Employee

Für die Mitarbeitertabelle wurden 100.000 Einträge erzeugt. Jede Person erhielt:

- eine eindeutige Sozialversicherungsnummer
- einen eindeutigen Namen
- eine zufällige Manager-Zuweisung
- mit 10-prozentiger Wahrscheinlichkeit: Zugehörigkeit zu einer technischen Abteilung

- zufälliges Gehalt und Anzahl an "Freunden".

```python
def generate_students(people, filename="students.csv"):
    with open(filename, "w", newline="") as file:
        writer = csv.writer(file)
        writer.writerow(["ssnum", "name", "course", "grade"])

        student_pool = people[-TOTAL_STUDENTS:]                                    # E
        for person in student_pool:
            course = random.choice(["Math", "Physics", "CS", "Biology", "History"])
            grade = random.choice(["A", "B", "C", "D", "F"])

            writer.writerow([
                person["ssnum"],
                person["name"],
                course,
                grade
            ])
```

Figure 5: Script für Erstellung von Employees

```
tuninglab2=# SELECT * FROM employee LIMIT 10;
 ssnum |          name          | manager |   dept    |  salary   | numfriends
-------+------------------------+---------+-----------+-----------+------------
     1 | Dr. Mallory Hanson DVM |         |           |  76490.09 |         50
     2 | Chad Henry             |       1 |           |  71186.76 |         37
     3 | Andrea Bell            |       2 | TechDept4 |  49135.28 |         67
     4 | Earl Norman            |       2 |           |  40673.55 |          9
     5 | Rachel Davis           |       1 | TechDept8 |  85920.77 |         36
     6 | Hannah English         |       2 |           |  92675.47 |         55
     7 | Jesus Parker           |       5 |           | 126731.09 |         76
     8 | Andrew Cole            |       2 |           | 140283.91 |         61
     9 | Jenna Chang            |       8 |           |  39204.40 |        100
    10 | Todd Kelly             |       4 |           |  91963.99 |          7
(10 rows)

tuninglab2#
```

Figure 6: Auszug aus Employee

Queries

Query 1

Original Query The first query shows the ssnum of all employees in tech-departments, that have a yearly salary within 1000 of the average salary over all tech-departments.

```sql
SELECT DISTINCT E1.ssnum
FROM Employee E1, Techdept T
WHERE E1.salary BETWEEN (
    (SELECT AVG(E2.salary)
     FROM Employee E2, Techdept T
     WHERE E2.dept = E1.dept
       AND E2.dept = T.dept) - 1000
) AND (
    (SELECT AVG(E2.salary)
     FROM Employee E2, Techdept T
     WHERE E2.dept = E1.dept
       AND E2.dept = T.dept) + 1000
);
```

Rewritten Query Give the rewritten query.

```
SELECT AVG(E.salary) as salary
FROM Employee E
JOIN Techdept T ON E.dept = T.dept;

SELECT DISTINCT E.ssnum
FROM Employee E
JOIN Techdept T ON E.dept = T.dept
WHERE E.salary BETWEEN salary - 1000 AND salary + 1000;
```

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

```
Nested Loop Inner Join
 Seq Scan on employee as e1
 Filter: ((salary >= ((SubPlan 1) - '1000'::numeric)) AND (salary <= ((SubPlan 2) + '1000'::numer
  Aggregate
   Nested Loop Inner Join
    Index Only Scan using idx_techdept_dept on techdept as t_1
    Index Cond: (dept = e1.dept)
    Bitmap Heap Scan on employee as e2
    Recheck Cond: (dept = e1.dept)
     Bitmap Index Scan using idx_employee_dept
     Index Cond: (dept = e1.dept)
  Aggregate
   Nested Loop Inner Join
    Index Only Scan using idx_techdept_dept on techdept as t_2
    Index Cond: (dept = e1.dept)
    Bitmap Heap Scan on employee as e2_1
    Recheck Cond: (dept = e1.dept)
     Bitmap Index Scan using idx_employee_dept
     Index Cond: (dept = e1.dept)
 Materialize
 Seq Scan on techdept as t
```

Nested Loop Inner Join: Die Tupel in Employee werden sequentiell gelesen. Dabei wird auf die Ergebnisse der beiden Sub-Queries SubPlan1 und SubPlan2 eingeschränkt. Anschließend wird das Ergebnis mit TechDept verglichen und gejoined

SubPlan1/2: Der Index in TechDept auf techdept wird verwendet, um alle Tech-Departements zu finden, die dem TechDept in Employee entsprechen. Für den Vergleich wird der Index auf techdept in Employee verwendet. Dieser Prozess wird 2x durchgeführt, da der durchschnittliche Verdienst 2x gebraucht wird.

```
Aggregate
 Hash Inner Join
 Hash Cond: (e.dept = t.dept)
  Seq Scan on employee as e
  Hash
   Seq Scan on techdept as t

Nested Loop Inner Join
 Seq Scan on employee as e
 Filter: ((salary >= '89008'::numeric) AND (salary <= '91008'::numeric))
 Memoize
  Index Only Scan using idx_techdept_dept on techdept as t
  Index Cond: (dept = e.dept)
```

Aggregate: Berechnet einen Wert auf Basis aller Datensätze, hier AVG(salary), auf Basis eines Hash Joins.

Hash Inner Join: Employee wird mit TechDept gejoined. Dies Erfolgt mit einen hash join, die Werte in beiden Tabellen werden also gelesen, dann wird ein Hash-Table auf Basis von TechDept erstellt ung mit Employee verglichen.

Nested Loop Inner Join: Die Werte in Employee werden sequentiell gelesen und nur Werte mit dem entsprechenden salary ausgewählt. Diese Werte werden mit TechDept verglichen, dabei wird auch der Index auf TechDept verwendet und gespeichert, damit der Index für das gleiche Department nicht mehrmals durchlaufen werden muss.

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 naiven Query werden für jedes Tupel in Employee zwei Joins mit jeweils einem Aggregate ausgeführt, während nach der Optimierung der durchschnittliche Wert 1x berechnet und dann wiederverwendet wird.

	Runtime [sec]
Original query	10.7524 seconds
Rewritten query	0.2958 seconds

Experiment Die verbesserte Query ist wesentlich schneller, da in der originalen Anfrage für jedes Tupel in Employee zwei Joins und dazu jeweils ein Aggregate berechnet werden müssen, was sehr zeit- und rechenaufwändige Operationen sind. Die verbesserte Query berechnet den Wert für die Einschränkung in Employee 1x und verwendet diesen dann immer wieder weiter.

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

```
('Nested Loop  (cost=0.15..5212.39 rows=10007 width=4) (actual time=0.041..34.670 rows=10061 loop
->  Seq Scan on employee  (cost=0.00..2723.00 rows=100000 width=14) (actual time=0.004..7.640 row
 ->  Memoize  (cost=0.15..0.16 rows=1 width=10) (actual time=0.000..0.000 rows=0       loops=1000
    ->   Index Only Scan using idx_techdept_dept on techdept       (cost=0.14..0.15 rows=1 width=
```

Nested Loop: For each employee, PostgreSQL checks if their dept exists in techdept, leading to many repeated sub-queries.

Seq Scan on employee: Scans the entire employee table row by row without using any index, which is slow for large datasets.

Memoize: Caches previous techdept lookups to avoid querying the same department multiple times, but still not optimal with many unique dept values.

Index Only Scan on techdept: Uses the index to check if the employee's department exists, but this check is done 100,000 times.

```
('Merge Join  (cost=1.56..1077.82 rows=10007 width=4) (actual time=0.070..5.647 rows=10061 loops=
 ->  Index Scan using idx_employee_dept on employee  (cost=0.29..9504.86 rows=100000 width=14) (a
 ->   Sort  (cost=1.27..1.29 rows=10 width=10) (actual time=0.040..0.042 rows=10         loops=1)
  ('        Sort Key: techdept.dept',)
  ('        Sort Method: quicksort  Memory: 25kB',)
  ->   Seq Scan on techdept  (cost=0.00..1.10 rows=10 width=10) (actual     time=0.008..0.009 row
```

Merge Join: Combines rows from employee and techdept where the dept values match, using sorted input for efficient merging.

Index Scan on employee: Quickly retrieves employee rows by scanning the index to get sorted dept values.

Sort on Techdept: Sorts values to prepare them for the merge join.

Seq Scan on Techdept: Reads all rows from the small techdept table sequentially before sorting.

Naive Query: Does a sequential scan over all employee rows.

- For each row, it checks if dept is in techdept using index lookup.
- Slower overall due to repeated subqueries.

Tuned Query: Uses Merge Join with:

- Index Scan on employee.dept
- Sorted Scan on techdept
- Optimizer uses indexes and sorting efficiently.
- Much faster due to bulk processing and join strategy.

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

	Runtime [sec]
Original query	0.1140 seconds
Rewritten query	0.0814 seconds

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

In der naiven Anfrage wird für jedes Tupel in Employee eine Anfrage in TechDept ausgeführt. Die optimierte Query verwendet die vorhandenen Indizes effizient und führt die Ergebnisse bereits vorab zusammen und filtert sie.

Time Spent on this Assignment

Time in hours per person: **4**

References

Important: Reference your information sources!
Remove this section if you use footnotes to reference your information sources.