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);
```

PostgreSQL automatically creates an index as soon as you put a UNIQUE contraint on a column. Therefore the indizes on ssnum and name would theoretically not be necessary, but we included them anyway for completness sake.

Populating the Tables

The tables are filled using a Python script. We used CSV files as target files to store the table contents. The underlying data (employees, students, departments) was randomly generated, taking into account certain correlations and conditions - e.g. overlaps in people, manager assignments and department memberships.

TechDept

For each technical department, we selected a randomly generated manager from the existing set of employees.

```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

100,000 entries were generated for the student table. Each person received:

- a unique social security number
- a unique name
- a random course and grade

```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 for creation of 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: Excerpt from Students

Employee

100,000 entries were generated for the employee table. Each person received:

- a unique social security number
- a unique name
- a random manager assignment
- with a 10 percent chance: an assignment to a technical departement

- random salary and amount of friends

```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 for creation of 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: The tuples in Employee are read sequentially. This is restricted to the results of the two sub-queries SubPlan1 and SubPlan2. The result is then compared with TechDept and joined

SubPlan1/2: The index in TechDept on techdept is used to find all tech departments that correspond to the TechDept in Employee. The index on techdept in Employee is used for the comparison. This process is carried out twice, as the average earnings are used twice.

```
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: Calculates a value based on all data records, in this case AVG(salary), based on a hash join.

Hash Inner Join: Employee is joined with TechDept. This is done with a hash join, so the values in both tables are read, then a hash table is created based on TechDept and compared with Employee.

Nested loop inner join: The values in Employee are read sequentially and only values with the corresponding salary are selected. These values are compared with TechDept, and the index on TechDept is also used and saved so that the index does not have to be run through several times for the same department.

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 the naive query, two joins with one aggregate each are executed for each tuple in Employee, while after optimization the average value is calculated once and then reused.

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

Experiment The improved query is significantly faster, as in the original query two joins and an aggregate have to be calculated for each tuple in Employee, which are very time-consuming and computationally intensive operations. The improved query calculates the value for the restriction in Employee once and then uses it again and again.

Query 2

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

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

Rewritten Query Give the rewritten query.

```sql
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 the naive query, a query is executed in TechDept for each tuple in Employee. The optimized query uses the existing indices efficiently and merges and filters the results in advance.

Differences between PostgreSQL and SQLite What differences did you observe between the postgres dbms and your alternative dbms?

Query	Runtime SQLite (initial)	Runtime SQLite (full)
Naive 1	0.897 seconds	76.247 seconds
Tuned 1	0.077 seconds	0.146 seconds
Naive 2	0.054 seconds	0.104 seconds
Tuned 2	0.076 seconds	0.074 seconds

SQLite uses a different execution model than PostgreSQL. Instead of storing the execution plan as a tree, it generates a sequential bytecode representation which is directly executed. The execution plan is a high-level description of this bytecode.

For query 1, significant runtime differences could be observed between the naive versions: SQLite managed to complete it in a tenth of the time when compared to PostgreSQL. However, further investigation revealed that this was only the initial execution time for the first row; retrieving all rows is in fact many times slower than in PostgreSQL.

To provide detailed query plans, we complied SQLite using the SQLITE_ENABLE_STMT_SCANSTATUS option. The plan for the naive query:

```
QUERY PLAN (cycles=2427167766 [100%])
|--CORRELATED SCALAR SUBQUERY 1                                    (cycles=1682897569 [69%] loops
|  |--SEARCH T USING COVERING INDEX idx_techdept_dept (dept=?)     (cycles=1006677 [0%] loops=115
|  `--SEARCH E2 USING INDEX idx_employee_dept (dept=?)             (cycles=1663633623 [69%] loops
|--CORRELATED SCALAR SUBQUERY 2                                    (cycles=739479704 [30%] loops=
|  |--SEARCH T USING COVERING INDEX idx_techdept_dept (dept=?)     (cycles=254716 [0%] loops=46 r
|  `--SEARCH E2 USING INDEX idx_employee_dept (dept=?)             (cycles=731309980 [30%] loops=
|--SCAN E1 USING INDEX idx_employee_ssnum                          (cycles=9360875 [0%] loops=1 r
`--SCAN T USING COVERING INDEX idx_techdept_dept                   (cycles=2448 [0%] loops=1 rows
```

The plan itself largely resembles that of PostgreSQL. A majority of the time is spent in looking up employees in an index for the dept column of employee.

The runtime difference is explained by the fact that we only retrieved the first row from the result, and SQLite's bytecode design allows for efficient evaluation of partial results. Attempting to retrieve all rows inflates the execution time to 82 seconds, almost ten times *slower* than PostgreSQL.

The tuned query reduces to two sequential scans over E, with a nested loop to find the corresponding tech department (if any):

```
QUERY PLAN (cycles=105487554 [100%])
|--SCAN E                                                       (cycles=55791135 [53%] loops=1 ro
`--SEARCH T USING COVERING INDEX idx_techdept_dept (dept=?)     (cycles=38080482 [36%] loops=1000

QUERY PLAN (cycles=138855928 [100%])
|--SCAN E                                                       (cycles=76062419 [55%] loops=1 ro
`--SEARCH T USING COVERING INDEX idx_techdept_dept (dept=?)     (cycles=31517151 [23%] loops=1000
```

For query 2, the runtime difference is less significant. However, it yields another interesting result: the tuned version is *slower* than the naive version.

Naive plan:

```
QUERY PLAN (cycles=243576876 [100%])
|--USING INDEX idx_techdept_dept FOR IN-OPERATOR
`--SEARCH employee USING INDEX idx_employee_dept (dept=?)     (cycles=241322241 [99%] loops=1 row
```

Here, SQLite traverses an index on the techdept table's dept field, and searches for employees with according departments using an index on employee.

In contrast, the tuned plan looks like:

```
QUERY PLAN
|--SCAN employee                                                    (loops=0 rows=0 rpl=NaN es
`--SEARCH techdept USING COVERING INDEX idx_techdept_dept (dept=?)     (loops=0 rows=0 rpl=NaN es
```

SQLite has decided to flip the looping order, doing a scan on the employee table and looking up the corresponding tech department for each employee. It is interesting to contrast this with PostgreSQL's approach, which goes with a sort-merge join. The reason for this discrepancy is that SQLite implements all joins as nested loops; it has no merge join strategy.

This time, we loaded the first 100 results, which the first query completes faster than the second query. However, when loading *all* results, the tuned query runs faster. (105ms vs 75ms).

A conclusion we can draw from this is that in SQLite, there may be a significant difference between the latency until the first row is returned and all rows are returned.

Time Spent on this Assignment

Time in hours per person: **4**

References

https://www.postgresql.org/docs/current/planner-optimizer.html
https://sqlite.org/draft/whybytecode.html
https://www.sqlite.org/optoverview.html#joins