Univ.-Prof. Dr.-Ing. Matthias Boehm
Graz University of Technology
Computer Science and Biomedical Engineering
Institute of Interactive Systems and Data Science
BMVIT endowed chair for Data Management

3 Database Systems WS19: Exercise 03 – Tuning and Transactions

Published: Nov 29, 2019 (last update: Dec 09 [changed predicate in 3.1])
Deadline: Dec 20, 2019, 11:59pm

This exercise on tuning and transactions aims to provide practical experience with physical design tuning such as indexing and materialized views, as well as aspects of query and transaction processing. The expected result is a zip archive named `DB_Exercise03_<student_ID>.zip`, submitted in TeachCenter. If all tasks (including extra credit) are submitted, it should contain `PhysicalDesign.pdf`, `QueryOptimization.sql`, `Transactions.sql`, `Tuning.sql`, and a folder called `Join` with the code of your join implementation.

3.1 Indexing and MatViews in SQL (5/25 points)

In order to understand the effects of physical design tuning, this task aims to compare query plans with and without existing data access structures that can be exploited for more efficient query processing. For this task, create or copy the SQL query, obtain the plan via text-based `EXPLAIN`, create the access structure, re-run and obtain the modified plan, and finally describe the plan differences.

- **Indexing:** Create a query that returns the airport names, where the latitude is above 70. Now, create a secondary index on the latitude and compare the new resulting plans.
- **Materialized Views:** Recall **Q5** from Task 2.3 and create a materialized view that could speed up computing the number of airports for arbitrary countries. Compare the plans for **Q5** with and without the materialized view.

Partial Results: SQL script `Tuning.sql` with queries, DDL, and plan comparison.

3.2 B-Tree Insertion and Deletion (6/25 points)

As a preparation step, let `x = 0.1<student_ID>` and generate a sequence of 16 numbers via `SET seed TO <x>; SELECT * FROM generate_series(1,16) ORDER BY random();`. Now, assume an empty B-tree with $k = 2$ (max $2k = 4$ keys, $2k + 1 = 5$ pointers), insert the sequence of numbers, and draw the resulting B-tree. Subsequently, delete all items with numbers that fall in the range $[5, 11)$ (lower inclusive, upper exclusive) in order, and draw the resulting B-tree again.

Partial Results: Add a section in `PhysicalDesign.pdf` with at least the two B-trees.

3.3 Join Implementation (9/25 points)

To understand query processing and important join implementations, the task is to implement a table scan, a projection (with bag semantics), and a Sort-Merge-Join operator (in your favorite programming language such as Python, Java, C# or C++). The three operators should implement the `open()`, `next()`, `close()` iterator model. The table scan should be created with a collection of type `Collection<Tuple>` (or similar depending on the used programming language) as input, where a `Tuple` has an `ID` and a list of other attributes. The join operator should be created with two iterators as input (left and right join input), realize a natural join (equi-join, join attribute appears just once in the output). Your code should be able to handle multisets (i.e., collections where the same `ID` appears multiple times). After that, use your operator implementations to execute a query $\pi((R \bowtie S) \bowtie T)$. In your own interest, you should test your operators with synthetic data, but it is unnecessary to submit the tests.

Partial Results: Put all source code files for the two operators, including custom `Tuple` implementations (no build/run scripts necessary) into a folder called `Join`.

3.4 Transaction Processing (5/25 points)

The final task explores key concepts of transaction processing. First, create two tables `R(a INT, b INT)` and `S(a INT, b INT)`, and write a SQL transaction that atomically inserts the following tuples (five each) into R and S, respectively.

```
R := {(11, 40), (12, 41), (13, 42), (14, 43), (15, 44)}
S := {(1, 10), (2, 30), (3, 35), (4, 10), (5, 10)}
```

Second, create two SQL transactions that can be executed interactively (e.g., in `psql` terminals) to create a deadlock and explain the reason of the deadlock. Please, annotate in comments in which order the transactions should be interleaved. Third, explain the impact of transaction isolation levels on your two SQL transactions that created the deadlock.

Partial Results: SQL script `Transactions.sql` including the explanations as comments.

3.5 Extra Credit (5 points)

Given the following suboptimal query specification, use the text or visual explain functionality to understand the resulting query execution plan and provide a query formulation that yields a plan with neither a union (append) operation nor a subquery (subplan).

```
SELECT DISTINCT C.Name
  FROM Cities C
  WHERE C.CityID IN (
    SELECT A.CityID FROM Airports A
      WHERE A.Latitude<10.9 AND A.Longtitude<2.35 AND C.CityID=A.CityID
    UNION ALL
    SELECT A.CityID FROM Airports A
      WHERE A.Latitude>48.86 AND A.Longtitude>116.6 AND C.CityID=A.CityID
  )
```

Partial Results: SQL script `QueryOptimization.sql` with the modified query.