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 SS19: Exercise 03 – Tuning and Transactions

Published: May 13, 2019
Deadline: Jun 04, 2019, 11.59pm CET

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.

3.1 Indexing, Partitioning, 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 all sub tasks, 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 distinct club names of players with jersey numbers less than or equal 3. Now, create a secondary index on attribute jersey number and compare the new resulting plans.
- **Materialized Views:** Recall **Q10** from Task 2.3 and create a materialized view that could speed up the computation of final tables for arbitrary groups. Compare the plans.

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 = 0.0<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 structure. Subsequently, delete all keys in the range $[8, 14)$ (lower inclusive, upper exclusive) in order, and draw the resulting B-tree again.

Partial Results: PDF `B-Tree.pdf` with the two B-trees.

3.3 Join Implementations (10/25 points)

To understand query processing and important join implementations, the task is to implement a table scan and two join operators: a nested loop join and a hash join (in your favorite programming language such as C, C++, Java, or Python). All three operators should implement the `open()`, `next()`, `close()` iterator model. The table scan should be created with a collection of type `Collection<Tuple>` as input, where a `Tuple` has an `ID` and a list of other attributes. In contrast, the join operators 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), and be able to handle multisets (i.e., collections where the same `ID` appears multiple times). In your own interest, you should test your operators with synthetic data, but it is unnecessary to submit the tests.

Partial Results: All source code files for the three operators, including custom `Tuple` implementations (no build/run scripts necessary).

3.4 Transaction Processing (4/25 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)`. Second, write a SQL transaction that atomically inserts two tuples into R and three tuples into S. Third, create two SQL transactions that can be executed interactively (annotate in comments in which order the transactions should be interleaved) to create a deadlock and explain the reason of the deadlock.

Partial Results: SQL script `Transactions.sql` for the three sub tasks, including the necessary explanations as comments.