3 Database Systems SS19: Exercise 03 – Tuning and Transactions

Published: May 01, 2019
Deadline: May 28, 2019, 11.59pm CET

This exercise on tuning and transactions aims to provide practical experience with physical design tuning such as indexing, partitioning, and materialized views, as well as aspects of 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 (6/25 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, please, create the SQL query, obtain the plan, create the access structure, obtain the modified plan, and finally highlight and explain the plan differences.

- **Indexing:** Create a query that returns all players with jersey numbers greater than 11, ordered by their jersey number. Now, create a secondary index on attribute jersey number and compare the new resulting plan.

- **Partitioning:** Create a query that computes the maximum number of goals per match between Jun/12 2014 and Jun/24 2014. Now, create a new table with range partitioning of match dates, copy the data into this table, and rerun the query over this table.

- **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.

**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 $x = 0.0<\text{student_ID}>$ 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 node size $n = 3$ ($n$ keys, $(n + 1)$ 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), and draw the resulting B+-tree again.

**Partial Results:** PDF `B-Tree.pdf` with the two B+-trees.
3.3 Join Implementations (9/25 points)

To understand query processing and important join implementations, the task is to implement two join operators: a nested loop join and a hash join (in your favorite programming language such as C, C++, Java, or Python). The two inputs and one output are collections of type `Collection<Tuple>`, where a `Tuple` has an `ID` and a list of other attributes. The operator implementations should realize an equi-join, use iterators over the input collections, and be able to handle multisets (i.e., collections where the same `ID` appears multiple times). Finally, discuss the advantages and disadvantages of the two different join implementations.

**Partial Results:** All source code files for nested loop and hash join, including a comment that discussed the pros and cons.

3.4 Transaction Processing (4/25 points)

The final task explores key concepts of transaction processing. First, create two tables \( R(\text{a INT, b INT}) \) and \( S(\text{a INT, b INT}) \). Second, write a SQL transaction that atomically inserts two tuples into \( R \) and one tuple into \( S \). Second, use two interactive SQL transactions (with a consistent isolation level) to create a deadlock. Explain the cause of this deadlock, and discuss the impact of different isolation levels.

**Partial Results:** SQL script `Transactions.sql` for the two sub tasks, including the necessary explanations.