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

3 Database Management WS 20/21: Exercise 03 – Tuning and TXs

Published: December 01, 2020 (last update Dec 10, simplified 3.3 invocation)
Deadline: December 22, 2020, 11:59pm

This exercise on tuning and transactions aims to provide practical experience with physical design tuning (such as indexing, materialized views), query, as well as transaction processing. The expected result is a zip archive named `DBExercise03_<studentID>.zip` containing all partial result files/folders, submitted in TeachCenter.

3.1 Query Processing, Materialized Views and Indexes (6/25 points)

In order to obtain a better understanding of query processing, optimization, the use of index structures and materialized views, this task aims to compare resulting plans before and after manual tuning. You can obtain the plans using `EXPLAIN`. Please check the `EXPLAIN` PostgreSQL documentation, to get information on how to output the plans in different formats.

(a) **Query Processing:** Write a query **Q09** that computes the average rating score and the number of ratings for the movie `Interstellar`. (return movie english title, the average rating, and the number of ratings).

Partial Result: SQL script `Q09.sql` and `Q09.json` for the plan.

(b) **Materialized Views:** Create a materialized view that speeds up the computation of the average rating score and the number of ratings for an **arbitrary** movie.

Partial Result: SQL script `MatView.sql` for materialized view creation and `Q09WithMatView.json` for the plan of Q09 using the materialized view.

(c) **Indexing:** The following query returns all movies that generated a revenue of more than 100,000,000. Obtain the plan for **Q10**, create a secondary (non-clustered) index on the attribute 'Revenue' of table Movies, and obtain the plan for **Q10** again.

```
Q10: SELECT Engtitle FROM Movies WHERE Revenue > 100000000;
```

Partial Result: Plan of Q10 without index `Q10WithoutIndex.json`, and the plan of Q10 `Q10WithIndex.json` using the index.

3.2 B-Tree Insertion and Deletion (5/25 points)

Given the sequence of numbers {11, 4, 2, 5, 8, 10, 6, 15, 9, 14, 16, 1, 13, 3, 7, 12}, sequentially insert this sequence—in the defined order—into an empty B-tree, and draw the resulting B-tree with $k = 2$ (i.e., max $2k = 4$ keys, $2k + 1 = 5$ pointers per node). Subsequently, delete all keys in the range [8, 14) (lower inclusive, upper exclusive) in the order of keys (i.e., del 8, del 9, ..., del 13), and draw again the resulting B-tree.

Partial Results: PDF files `BTreeInitial.pdf` and `BTreeAfterDelete.pdf`.

3.3 Iterator Model and Operator Implementation (10/25 points)

For a deeper understanding of the iterator model, individual operators, and query processing, implement the following operators in the `open()`, `next()`, `close()` iterator model (e.g., via an iterator base class and derived operators classes) in a programming language of your choosing (e.g. Python, Java, C# or C++). You can assume integer types for all attributes.

- `TblScan(filename)`: A table scan operator that takes a string `filename` of a csv file, and returns its rows in the sequence they appear in the file.
- `EqSelect(input, attr, value)`: A selection operator that takes an iterator `input` (i.e., a sub query), an attribute position `attr`, an integer `value`, and returns only rows `t` where `t[attr] == value`.
- `Sort(input, attr)`: A sort operator that takes an iterator `input` (i.e., a sub query), and an attribute position `attr` and returns rows in sorted order, ascending by `t[attr]`.
- `MergeJoin(input1, input2, attr1, attr2)`: A join operator that takes two iterators `input1` and `input2`, as well as attribute positions `attr1` and `attr2`, and performs a merge join with condition `t1[attr1] == t2[attr2]` (expecting `input2[attr2]` to be unique).

For testing, implement the query **Q11:** $\sigma_{c=7}(R) \bowtie_{\text{R.SID=S.SID}} S$ with the help of these operators. R and S are read from files `./R.csv` and `./S.csv` using `TblScan` operators, and S is assumed to be sorted ascending by sID. The code for reading input files can be re-used from Exercise 2. Please provide a script—to compile and run the program—that can be invoked as follows:

```
./runQuery11.sh ./R.csv ./S.csv ./output.csv
```

Partial Results: A folder `IteratorModel` including the source code of all required operators, test query, and the script `./runQuery.sh` to compile and run the program.

3.4 Transaction Processing (4/25 points)

(a) For a basic understanding of transaction processing, create two tables `R(a INT, b INT)`, `S(a INT, b INT)`, and insert the following tuples in one atomic transaction:

```
R := {(36, 90) (71, 18) (63, 84) (4, 10)}
S := {(29, 21) (67, 41) (36, 75)}
```

Partial Result: SQL script `Transactions.sql` with the insert transaction.

(b) Having the tuples inserted into the database, simulate a **Deadlock** via two transactions, and explain (in comments) how the operations should be interleaved to create the deadlock.

Partial Result: SQL script `Deadlock.sql` with the transactions and a brief explanation.