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 SS22: Exercise 03 – Tuning and TXs

Published: May 9, 2022
Deadline: May 31, 2022, 11.59pm

This exercise on tuning and transactions aims to provide practical experience with physical design tuning (such as indexing), as well as query and transaction processing. The expected result is a zip archive named `DBExercise03_<studentID>.zip` containing all partial results in folders per task (i.e., T3.1,...,T3.4), submitted in TeachCenter. Make sure to adhere to the requested formats of results, because this exercise is subject to automated grading.

3.1 Query Processing and Indexes (5/25 points)

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

(a) **Query Processing:** Write a(n) SQL query **Q09** that finds districts with more than 1500 inhabitants holding citizenships other than `'Austria'` as of '2022-01-01'. (return district name, country name, population count).

Partial Result: SQL script `Q09.sql` and `Q09.json` for the plan (after calling `ANALYZE`) including estimated costs.

(b) **Indexing:** Create one or many indexes on attributes of your choosing to reduce the estimated costs of query Q09. After creating these indexes, obtain the plan for **Q09** again.

Partial Result: SQL script `Index.sql` for creating the index, and the plan of **Q09** `Q09WithIndex.json` using the index.

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

As a preparation step, obtain a seed via the following SQL query with X set to your student-ID:

```
SELECT SETSEED(1.0/(SELECT MOD(X,8)+1));
SELECT * FROM generate_series(1,16) ORDER BY random();
```

Now, insert all numbers of the obtained sequence—in sequence order—into an empty B-tree with $k = 2$ (i.e., max $2k = 4$ keys, $2k + 1 = 5$ pointers per node) and capture the resulting B-tree. 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 again capture the resulting B-tree. Please, use the following text format to represent both B-trees.

```
node_id: (child_node_id 1) key (child_node_id 2) ... (child_node_id n)
```

For instance, the following example represents a tree of height two, where (a) is the root node pointing to child nodes (b) and (c), respectively. Append each node as a separate line (without empty lines), assign unique node IDs, and linearized the tree in a depth- or breadth-first manner.

```
a: (b) 7 (c)
b: 2 () 4
c: 8 () 9 () 12
```

Partial Results: Input sequence `Input.txt` (copied from the PostgreSQL output), and the textual representation of the two B-trees `BTreeAfterInsert.txt` and `BTreeAfterDelete.txt`.

3.3 Transaction Processing (6/25 points)

(a) Create the tables `Vendors(VID, Name, Profit)`, `Products(PID, Name, Price, Stock)`, and `Sales(SDate, VID, PID, Quantity)` with meaningful data types. Then insert `(9, 'V1', 0.0)` into Vendors, and `(1, 'P1', 25, 100)` into Products.

Partial Result: SQL script `TXSetup.sql` that robustly handles existing tables.

(b) Write a(n) SQL transaction (in an isolation level preventing dirty reads) that atomically adds a *new order* (into Sales) for 10 instances of product `'P1'` by vendor `'V1'` sold as of `2022-04-07`, and modifies the product stock and vendor profit accordingly (the profit can be computed directly from the quantity and price of the product).

Partial Result: SQL script `TXNewOrder.sql` containing the new order transaction.

(c) 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.

3.4 Iterator Model and Operator Implementation (8/25 points, extra credit 706.010)

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 string types for all attributes.

- `TblScan(filename)`: A table scan operator that takes a string `filename` of a csv file, and returns its rows (each as an array of strings) in the sequence they appear in the file.
- `CmpSelect(input, attr, cmp, value)`: A selection operator that takes an iterator `input` (i.e., a sub query), an attribute position `attr`, a comparison operator `cmp` (specifically, `ge` for greater-than-or-equals and eq for equals), an integer or string `value`, and returns only rows `t` where `t[attr] cmp value`.
- `HashJoin(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 hash join with condition `t1[attr1] == t2[attr2]` (expecting `input2[attr2]` to be unique).

Your implementation can use existing library data structures like hash maps (or dictionaries), and reuse the code for reading input files from Exercise 2. For testing, implement the following query **Q10** with the help of these operators.

```
SELECT *
  FROM  Districts D, PopByGender PG
  WHERE D.DKey = PG.DKey
    AND D.Population >= 30000
    AND PG.PopDate = '2022-01-01'
```

Please also provide a shell (or bat) script for compiling and running your program as follows:

```
./runQuery10.sh ./Districts.csv ./PopByGender.csv ./out.csv
```

Partial Results: A folder `T3.4` including the source code of all required operators and the test query, as well as the script `./runQuery10.sh` to compile and run the program.