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 SS21: Exercise 03 – Tuning and TXs

Published: May 4, 2021 (updates: May 7: T3.2 seed)
Deadline: May 25, 2021, 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 query and transaction processing. The expected result is a zip archive named `DBExercise03_<studentID>.zip` containing all partial results in folders per task (e.g., T3.1, T3.2), 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, Materialized Views and Indexes (6/25 points)

In order to obtain a better understanding of query processing, optimization, and the use of index structures and materialized views, in this task please compare the 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 lists the top 10 athletes by the number of Olympic games they participated in. (return athlete key, athlete name, number of games; sorted descending by number of games, ascending by athlete name).

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

(b) **Materialized Views:** Create a materialized view that could speed up reoccurring queries such as **Q09** and similar queries with *arbitrary length* of the top-k list (e.g. top 100).

Partial Result: SQL script `MatView.sql` for creating the materialized view creation, and the plan of Q09 modified for explicitly using the materialized view, exported as `Q09WithMatView.json`.

(c) **Indexing:** The following query **Q10** returns the average BMI of athletes with a weight less than 50 kg. Obtain the plan for **Q10**, create a secondary (non-clustered) index on an attribute that helps speed up this query, and obtain the plan for **Q10** again.

```
Q10: SELECT round(avg(A.Weight / (A.Height/100.0 * A.Height/100.0)), 2)
       FROM Athletes A
       WHERE A.Weight < 50
```

Partial Result: Plan of **Q10** without index `Q10WithoutIndex.json`, SQL script `Index.sql` for creating the index, and the plan of **Q10** `Q10WithIndex.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 (4/25 points)

(a) Create the tables `Customers(CID, Name, Debt)`, `Products(PID, Name, Price, Stock)`, and `Orders(ODate, CID, PID, Quantity)` with meaningful data types. Then insert `(7, 'C1', 0.0)` into Customer, 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 by Customer `'C1'` for 15 times product `'P1'` as of `2021-05-25`, and modifies the product stock and customer debt accordingly.

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

3.4 Iterator Model and Operator Implementation (9/25 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.
- `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`.
- `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).
- `HashCountAgg(input, attr1, attr2)`: A group-by operator that takes an iterator `input`, as well as attribute positions `attr1` and `attr2`, and performs a hash group-by with grouping attribute `attr1`, aggregate attribute `attr2` and aggregation function count(attr2).

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 **Q11** with the help of these operators.

```
SELECT R.Medal, count(R.EKey)
  FROM Athletes A, Results R
  WHERE A.AKey = R.AKey
    AND A.Gender = 'F'
  GROUP BY R.Medal
```

The tables Athletes and Results are provided as `./Athletes.csv` and `./Results.csv` (identical to the respective target tables from Exercise 2, which means you can run the SQL query to double check your results). Furthermore, please provide a shell (or bat) script for compiling and running your program as follows:

```
./runQuery11.sh ./Athletes.csv ./Results.csv ./output.csv
```

Partial Results: A folder `T3.4` including the source code of all required operators and the test query, as well as the script `./runQuery11.sh` to compile and run the program.