TECHNISCHE UNIVERSITÄT MÜNCHEN, QUERY OPTIMIZATION

Exercise 8

Matthias Gollwitzer, Jan Schalkamp

July 7, 2013

Exercise 1+2: Make as usual via *make* command. TPCH data is again expected in *data/tpch/tpch* with semicolons only as delimiter and no other occurrences of them.

1 EXERCISE 1

- Execute via **./bin/homework8-1**
- **src/compiler/randomized/TreeGenerator::generateRandomTree:** Responsible for generating trees with a random rank for structure and permutation of relations. TreeGenerator::generateRandomTree utilizes a Marsenne twister to generate pseudo-random numbers. For the structure rank, this will be a (uniformly distributed) random number between 0 and $C(|\text{Joins}|-1)$ with C as Catalan number. The rank of the permutation of relations will be between 0 and (|relations|)!.
- **src/compiler/randomized/TreeGenerator::generateJoin:** The fun part of this class. Builds the corresponding tree to a given structure/permutation recursively. The idea is, that we can see where in the structure list the right sub tree begins. This is *generally* the case, if the difference between two consecutive values is bigger than the index of the "right" value. An example: structure $:=<1,2,5,6>$. Here $<1,2>$ belongs to the left sub tree (since $2-1=1$ is less/equal to the index of $<2>$ – which is 1) and $<5,6>$ to the right sub tree (as $5-2=3$ is bigger than the index of $<5>$ – which is 2). From these derived sub lists, we build new valid structure lists. Therefore the left list will be change as follows: the head will be deleted – as it was the one depicting the topmost join – and all values will be decreased by 1. The right list keeps all it's elements, but their values will be decreased by $(b+1)$ – with b being the index in the original structure list of the first element that belongs to the right sub tree. In our example, the two new structure lists will be: $<1>$ and $<1,2>$. There is one special case worth mentioning (hence the *generally* earlier). The following example with it's specific length and values might even be the single case, which is not covered by our general rule. Consider the structure $<1,2,4,7>$. Here, each difference between two consecutive values is exactly the index

of the bigger value, wherefore our rule would assume that there is no right side. One can see quite easily, that there should indeed be a right side, as 7 is the latest possible occurrence of an opening bracket in a dyck word with length 4 (and has therefore to be in the right sub tree). Our algorithm checks therefore additionally for the occurrence of the last possible opening bracket (structure length $* 2 - 1$) and then handles it as the right sub tree[1].

- **src/compiler/CostCalculator:** Calculates the cost for a given tree and query graph.
- **src/compiler/strategies/RepeatedRandomStrategy:** Executes TreeGenerator::generateRandomTree multiple times (amount specified in the header file – currently 100) evaluates the returned tree's cost via CostCalculator::getCosts and returns the cheapest one. Also maintains the distribution of occurred costs as a map from cost(double) to occurrences(unsigned).

2 Exercise 2

- Execute via **./bin/homework8-2**
- **src/compiler/strategies/QuickPickTree:** Entrypoint of data structure for storing partial solutions. Header file implements the structure *QPNode*. To store intermediate results we use a reversed tree structure. Each node has a pointer either to the parent node or, in case of a root, to the intermediate AST tree. For each relation we maintain a node in a vector with the relation number used as index. This way it is not necessary to update each affected relation node when joining two subtrees.
- **src/compiler/strategies/QuickPickStrategy:** Creates a random join tree by randomly converting edges of the query graph to joins in the AST. Uses Marsenne twister for random edge selection, too.
- **src/compiler/strategies/RepeatedQuickPickStrategy:** Executes QuickPickStrategy multiple times (amount specified in the header file – currently 100) evaluates the returned tree's cost via CostCalculator::getCosts and returns the cheapest one. Also maintains the distribution of occurred costs as a map from cost(double) to occurrences(unsigned).

3 Exercise 3

Figure 3.1 depicts the chosen query graph, with relations cardinality is $\forall R_i : |R_i| = 10$. Our rule set consists of the following rules:

- Commutativity: $R_1 \bowtie R_2 \rightsquigarrow R_2 \bowtie R_1$
- Left Join Echange: $(R_1 \bowtie R_2) \bowtie R_3 \rightsquigarrow (R_1 \bowtie R_3) \bowtie R_2$

[1] maxOpen is the variable used for this check

Furthermore Figures 3.2 and 3.3 show the chosen start and the optimal solution. As both rules applied to the start solution yield no improvement, the optimal solution will never be found.

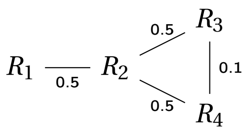

Figure 3.1: Example Query Graph

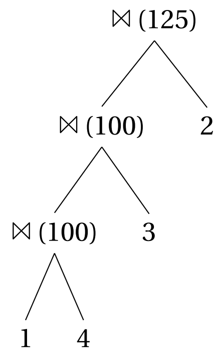

Figure 3.2: Start solution

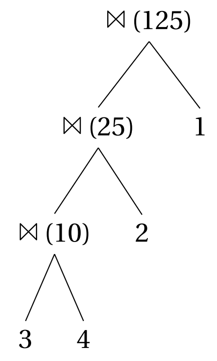

Figure 3.3: Optimal solution