Assignment 8

Issue Date: January 29, 2019
Due Date: February 11, 2019, 10:00 A.M.
$\sum$ 40 Points

Database System Architecture and Implementation
INF-20210
WS 2018/19

Universität Konstanz

University of Konstanz
Database and Information Systems
Prof. Dr. Michael Grossniklaus
Leonard Wörteler

Cost Estimation and Query Optimization

i General Notes

Please observe the following points in order to ensure full participation in the lecture and to get full credit for the exercises. Also take note of the general policies that apply to this course as listed on the course website `http://www.informatik.uni-konstanz.de/grossniklaus/education/inf-20210/`.

- Submit your solutions through Ilias **before the deadline** published on the website and the assignment.
- Since this is a programming assignment, you will submit an archive (ZIP file or similar) that only contains the **relevant** source code files and a `README.txt` file (see Section Submission below).
- The use of external libraries is **not** permitted, except if they are explicitly provided by us.
- Submissions with compile-time errors due to syntax or Checkstyle problems will **immediately** be disqualified. Whereas solutions with run-time errors will be considered, they will not result in a good grade.

Disclaimer: Please note that Minibase is neither open-source, freeware, nor shareware. Refer to the file `COPYRIGHT.txt` in the `doc` folder of the Minibase distribution for terms and conditions. Also do not push the code to publicly accessible repositories, e.g., GitHub.

i Assignment Overview

The best place to start understanding this assignment are the lecture slides and the source code itself. In this assignment, you will work on the query optimizer layer, which is part of Minibase's query processor and is located in package `minibase.query.optimizer`. Your task is to add an additional physical operator to the cost model of the query optimizer as well as to implement a transformation and an implementation rule. The `minibase.query.optimizer` package is organized as follows.

Here be dragons: The optimizer currently implemented in Minibase is directly translated from other peoples' research code (see `http://web.cecs.pdx.edu/~len/Columbia/`). As such, do expect (some) bugs or unexpected behaviors. While you can report those, we are probably aware of them already. Your task is to use the optimizer despite these challenges.

`m.o` The main package contains the implementation of the search space data structure, which consists of the classes `SearchSpace`, `Group`, and `MultiExpression`. Furthermore, it contains the representations of the various properties that search space expressions can have. Finally, another important class is `Expression`, which represents both the input and output of the query optimizer, cf. method `optimize()` of class `QueryOptimizer`. Expressions are basically tuples of the form $(op, in_1, in_2, \ldots, in_n)$, where op is the operator and in_i are the inputs. Expressions can either be created programmatically or using a simple language (see file `Sailors.qry` in `src/test/resources/minibase/query/optimizer` for an example).

m.o.operators Contains all operators that are known to the query optimizer. Mainly these are the logical operators of the relational algebra and their implementations as physical operators. Note that the operator classes used by the optimizer only *describe* these operators and do not provide any functionality for query processing. The third category of operators are so-called element operators that operate on single elements instead of entire sets. Element operators are used to represent, for example, selection predicates. You will add your new physical operator to this package.

m.o.rules Contains all the rules used by the query optimizer to transform expressions. In addition to the rules, this package also contains the `RuleBindery` that matches rules to expressions in the search space. Finally, the class `RuleManager` is also part of this package. By configuring the rule manager, you can activate and deactivate the rules used by the optimizer. The two new rules will be added to this package.

m.o.tasks Contains the tasks that are performed by the optimizer in order to explore the search space. You will not need to edit or add classes in this package.

m.o.util Contains mysterious classes that perform shockingly unexpected functionality.

A good way to check whether you have set up your Eclipse project correctly, is to launch the `SearchSpaceTest` JUnit test. The search space JUnit test creates a transient system catalog using the metadata specified by `Sailors_catalog.xml`, which is also located under `src/test/resources/minibase/query/optimizer`. Based on this system catalog, it then optimizes the following query, which is located in `Sailors.qry`.

```
(DISTINCT,
   (PROJECT(<S.sname, B.bname>),
      (SELECT,
         (EQJOIN(R.sid, S.sid),
            (EQJOIN(B.bid, R.bid),
               GET(Boats, B),
               GET(Reserves, R)
            ),
            GET(Sailors, S)
         ),
         (OP_OR, (OP_GT, ATTR(S.rating), INT(3)), (OP_LT, ATTR(S.rating), INT(5)))
      )
   )
)
```

The SQL query that corresponds to this expression is as follows.

```
SELECT DISTINCT S.sname, B.bname
FROM Sailors AS S
        JOIN Reserves AS R ON S.sid = R.sid
        JOIN Boats AS B ON R.bid = B.bis
WHERE S.rating > 3 OR S.rating < 5
```

Once the query is optimized, the JUnit test prints out the chosen plan followed by a dump of the optimizer search space in terms of all its groups. The printout of the optimized plan shows all the physical operators chosen by the optimizer as well as the estimated costs and cardinalities. The first line of the plan, which denotes its root operator (`HashDuplicates`), reads approximately as follows.

```
HashDuplicates (cost=3.4, i/o=0.8, cpu=2.6, card=1500, ucard=1500, twidth=0.012)
```

Therefore, the plan is estimated to have an overall cost of 3.4 and return (unique) $1{,}500$ rows. We can also see that in the chosen plan the I/O costs are *lower* than the CPU costs. The last value (`twidth`) denotes the width of tuples as a fraction of a disk page.

Exercise 1.1: Transformation Rule (10 Points)

The first exercise is to implement a transformation rule, i.e., an optimizer rule that transforms one logical expression into another. The rule you will implement exploits that equi-joins commute: $A \bowtie B \equiv B \bowtie A$. This rule is implemented by class `EquiJoinCommute`, which is provided as a skeleton with the source code for this assignment.

a) First, you will need to complete the constructor of the class. The only thing the constructor needs to do is to call `super()` to correctly configure the rule. In order to do so, you need to replace the dummy-values in the skeleton with meaningful values.
   - The first parameter is the type of the rule according to the `RuleType` enumeration.
   - The second parameter is a bit-mask that denotes the rules that should be blocked on a particular expression after this rule has been applied. Typically, the rule itself is included here to prevent flip-flopping. In the present case, it is also wise to block equi-join left-to-right and right-to-left rotation as well as the exchange rule.
   - The third parameter is the expression pattern that the rule bindery will look for in order to bind this rule to an expression. As you can see in other rules, this pattern is created by instantiating an `Expression` with a corresponding operator (`EquiJoin` for this rule). Placeholders in the rule are represented by `Leaf` "operators" that have no inputs. Each leaf operator is given an id in its constructor.
   - The fourth and last parameter is the output expression pattern generated by this rule. Similarly you need to create an expression consisting of operators and leaves. In order to denote what happens with the leaves during transformation, the ids assigned in the original pattern are reused.

b) In the second step, you will need to implement the `nextSubstitute()` method. This method needs to perform the following steps. Assume a standard Grace Hash Join for your calculations.
   - Retrieve the join predicate of the original (`before`) expression. Join predicates are given by two lists of column references (e.g., `R.bid`), which are compared pair-wise by the join for equality.
   - Create a new equi-join operator with a reversed join condition.
   - Create and return a new expression by using the new operator and by switching the inputs of the original expression.

Once the rule is implemented, do not forget to activate it in class `RuleManager`.

Exercise 1.2: Physical Operator (10 Points)

In the second exercise your task is to add a new physical operator (`HashJoin`) in terms of its cost model to the optimizer. Again, a skeleton of the class is also provided in the source code of this assignment. In order to compute the I/O and CPU cost of a hash join during optimization, you will need to complete the `getLocalCost()` method to perform the following steps.

- First, you will need to retrieve the cardinality of result ($card_{out}$) as well as the left ($card_{left}$) and the right ($card_{right}$) input of the hash join operator. These cardinalities can be retrieved from the parameters given to the `getLocalCost()` method. *Hint:* You may need to access the parameters as instances of `LogicalCollectionProperties`.
- Then, you have to find out the width of the tuples contained in the left ($width_{left}$) and right ($width_{right}$) input. The width is a property of the schema, which is also part of the logical collection properties and can be retrieved using method `Schema#getLength()`.
- Finally, you need to compute the I/O and CPU cost of the hash join and return a new `Cost` object that wraps these two values.
  - For the I/O cost, you may assume that there is enough buffer space to perform the hash join in two passes. The cost of one (sequential) page I/O is given by `CostModel.IO_SEQ` and the overall I/O operations are computed based on the cardinality and width of the left and right input.
  - The CPU cost consists of the cost for hashing (`CostModel.HASH_COST`) input tuples, probing (`CostModel.HASH_PROBE`) into a hash table, and forming output tuples and passing them to the next iterator (`CostModel.TOUCH_COPY`).

Exercise 1.3: Implementation Rule (10 Points)

After completing the implementation of the hash join operator, you will need to create a corresponding implementation rule that enables the query optimizer to use this new physical operator. Class `EquiJoinToHashJoin` provides a skeleton for this rule. Apart from the constructor and the `nextSubstitute()` method, you will also need to implement the `getPromise()` method of the rule.

a) As for the `EquiJoinCommute` rule, the constructor of `EquiJoinToHashJoin` simply calls `super()` to configure the rule. In the `super()` call, you will need to specify the type of the rule (from enumeration `RuleType`) as well as the original and substitute patterns. Note that it is *not* necessary to define a bit-mask as implementation rules are a "one-way street" and, therefore, no flip-flopping can occur. Repeated application of the same rule is prevented by the default behavior implemented in `AbstractRule`.

b) The `nextSubstitute()` method extracts the left and right column references from the logical `EquiJoin` operator of the `before` expression of the rule. These are then used to create a new physical operator based on your implementation of `HashJoin`. Finally, a new expression needs to be constructed and returned that uses this operator and combines it with the same inputs as the `before` expression.

c) In contrast to the `EquiJoinCommute` rule, the `EquiJoinToHashJoin` rule also needs to implement the `getPromise()` method. This method is called by `ApplyRuleTask` in order to rank the rules that are applicable to a given expression. It can, therefore, be used to prevent that this rule is applied to an equi-join that actually is a cross-product, i.e., has an empty join condition. If this is the case, your implementation should return `Promise.NONE` instead of `Promise.HASH`.

Again, do not forget to activate the rule in class `RuleManager` once it is fully implemented.

Exercise 2: Cost Estimations (2 Points)

To improve cost estimations, cost indicators can be attached to the operators in a query plan. Name two important factors that can influence the costs of operators of a single operator type (e.g. *Projection*, *HashJoin*, ...).

Exercise 3: Cost Calculations (2 Points)

The following costs and selectivities are given:

Operator	Selectivity	Costs
O_1	$S_1 = 0.1$	100
O_2	$S_2 = 0.2$	20
O_3	$S_3 = 0.8$	10

Choose and name one of the three strategies given in the lecture slides (*DNF*, *CNF*, *Bypass*), and calculate the costs of the two following operations: a) $(O_1 \vee O_2) \wedge O_3$ b) $(O_1 \wedge O_2) \vee O_3$

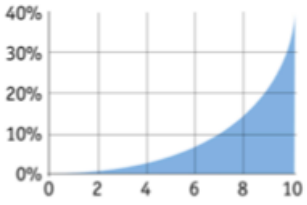
Exercise 4: Join Conditions (4 Points)

Rank the following join operators according to how well the (core) algorithm can support non-equi joins ($\neq, \leqslant, >$, predicates calling user-supplied functions, ...). Justify each item's position in one short sentence considering, for example, algorithm specific properties.

• Sort-Merge • Index-nested Loops • Hash • Block-nested Loops

Exercise 5: Histograms (2 Points)

The histogram on the right shows the (approximate!) data distribution (few entries near value 0 exist, most entries have high values) of a specific floating-point column `A.x` of a table `A` in the database. Formulate a sample SQL query that can be optimized to be substantially faster if the optimizer has access to the histogram than without it, and describe the differences in the resulting execution plans.



i Submission

Solutions are submitted electronically via Ilias. Your submission must consist of a single zipped archive that contains only the fleshed-out skeleton files containing your implementation (without any other files from the project, especially not generated classes or eclipse-specific configuration). The following files should be included regardless of whether you were able to solve the respective exercise: `EquiJoinCommute`, `HashJoin`, `EquiJoinToHashJoin`, and `RuleManager`. *Do not rename the classes!*

The name of the archive containing your submission should be `grp#-asg#.zip`, where # is your group and the assignment number, respectively. You will also include a **PDF** file called `grp-#-asg#.pdf` that states your group number as well as the following information.

a) Information about your implementation

Overall Status A one paragraph overview of the status of your implementation. If you were unable to finish the project, please give details about what is and is not complete. Be short and to the point!

Time Spent Please include how many hours you spent on this project. Note that the time spent has no impact on your grade. This information will only be used in planning future projects.

b) Solutions to written exercises