CS 4321/5321 Project 5

Fall 2017

Due November 30th, 11:59

This project is out of 75 points and counts for 22% of your grade.

1 Goals and important points

In this last Project, you will bring together the work you did in Projects 3 and 4 and you will add optimization functionality to your interpreter.

- you will add code that gathers statistics about the data, in order to allow informed optimization.
- you will improve your query plan by pushing selections in a more thorough way than previously.
- you will implement an algorithm that chooses the join order for each query rather than following the order in the `FROM` clause.
- you will implement algorithms/heuristics to choose the best implementation for each selection and each join operator in your improved query plan.

You do not have to do any further optimizations, such as pushing projections. The goal in this Project is not to develop a full-fledged optimizer, but to focus on a subset of important optimization techniques.

You will still support the same subset of SQL as in the previous Projects and should follow the same specification for sorting (`ORDER BY`). In addition, if a query has a `SELECT *` and multiple tables in the `FROM` clause, you should return columns in the order specified by the `FROM` clause. For example in `SELECT * FROM R,S,T WHERE ...` return the columns of `R` followed by the columns of `S` followed by `T`. This ordering happened "naturally" in previous projects because your join tree followed the `FROM` clause; in this project you will be changing the join order so you need to explicitly enforce the above behavior for `SELECT *`.

2 Input and output formats

The format for input and output is the same as in Project 4, with the following exceptions.

First, the `PhysicalPlanBuilder` configuration file is no longer needed. Previously, you told the plan builder whether to use indexes, and what implementation of join to use. Now, the plan builder will make those decisions by itself. You may hard-code in the `PhysicalPlanBuilder` the choice of sorting algorithm (use external sort unless you were not able to implement it) as well as the number of buffer pages to be used for BNLJ.

Second, you need to gather and write statistics about your data in a `stats.txt` file. This will sit directly inside the `db` subdirectory, at the same level as `schema.txt` and `index_info.txt`. `stats.txt` contains information on each relation, namely the number of tuples in the relation, and for every attribute, the minimum and the maximum value that this attribute takes. A sample file is given below:

```
Reserves 10000 G,0,5000 H,10,10000
Boats 5000 D,10,500 E,10,500 F,10,500
```

The first line of this file states that relation `Reserves` has 10000 tuples, with the values of attribute `G` ranging from 0 to 5000, and the values of `H` ranging from 10 to 10000. The attributes are listed in the same order as in the `schema.txt` file, and for each attribute the attribute name, the min and the max values are separated by commas. The lines in the file may be in any order. We will not test your code with databases containing empty relations, so you do not have to worry about the format of the statistics when a relation is empty.

Finally, the configuration file for the interpreter is simplified. It will now only contain three lines, with the input, output and temporary sort directories. You are basically passing it the same information as you did in Project 3, but since you had an interpreter config file in Project 4, we chose to keep this information in the interpreter config file for Project 5 rather than go back to command line arguments.

When your interpreter is run, it should gather statistics as described above, build indexes based on the `index_info.txt` file, and execute the queries. In addition to printing out the answers, it should also print out both the logical and physical query plans to the output directory. The logical plan for the *i*th query goes into file `queryi_logicalplan` and the physical plan goes into the file `queryi_physicalplan`. Thus your output directory will contain three different files per query: the file with the physical plan, the file with the logical plan, and the file with the actual results.

The formats for printing out the logical and physical query plans are described next.

2.1 Formats for logical and physical query plans

This section explains the format for physical and logical query plans. It should be skipped on first reading; return to this section after you understand the specific structure of the logical and physical query plans we expect, so that you understand e.g. what a union-find is and what it is doing inside a logical join operator.

We will be grading your plans through a combination of manual and automated grading; while there is some flexibility in the formats as explained below, you should follow it as closely as possible, since blatant disregard for the format will attract point deductions.

The format to be used for the logical plans is illustrated by the following example.

Consider the query

```
SELECT DISTINCT S.A, R.G
FROM Sailors S, Reserves R, Boats B
WHERE S.B = R.G AND S.A = B.D AND R.H <> B.D AND R.H < 100
ORDER BY S.A;
```

This gives the following logical query plan:

```
DupElim
-Sort[S.A]
--Project[S.A, R.G]
---Join[R.H <> B.D]
[[S.B, R.G], equals null, min null, max null]
[[S.A, B.D], equals null, min null, max null]
[[R.H], equals null, min null, max 99]
----Leaf[Sailors]
----Select[R.H <= 99]
-----Leaf[Reserves]
----Leaf[Boats]
```

There is one operator per line except for join which takes up several lines, and the dashes represent levels in the tree. The top operator is a duplicate elimination operator, followed by a sort, then a project, then a join having three children: a leaf, a select with a leaf under it and a third leaf.

The dashes represent the levels in the tree; an operator with m dashes before it is at level m in the tree, with the root being at level 0.

Every operator is printed with relevant information as follows:

- duplicate elimination has no extra information
- sort and projection have a list of attributes for the sort/projection; order is important for those.
- selection has a selection expression; the order of the comparisons in the selection is *not* important.
- leaf operators have the name of the base table which is being scanned
- the join operator has two pieces of information: the residual join expression and the union-find (See Section 3.2.). The residual join expression is printed first, followed by the union-find. In the residual join expression the ordering of the subexpressions doesn't matter. In the union find, we have a list of all the elements, printed one to a line. Order of the elements doesn't matter. In every element of the union find, there is one or more attributes (order does not matter), and constraints for equality, min and max.

For physical plans, the format is similar with a few minor differences. Here is a sample physical plan corresponding to the above logical plan:

```
DupElim
-ExternalSort[S.A]
--Project[S.A, R.G]
---BNLJ[R.H <> B.D AND S.A = B.D]
----SMJ[R.G = S.B]
-----ExternalSort[R.G]
------Select[R.H <= 99]
-------TableScan[Reserves]
-----ExternalSort[S.B]
------TableScan[Sailors]
----TableScan[Boats]
```

The join operators now need to state the name of the join implementation (BNLJ, SMJ or TNLJ) as well as the join expression. The first child for the join is the left (outer) child, and the second child is the right (inner). Every physical sort operator sort specifies whether it is an ExternalSort or an InMemorySort, and leaf operators become either TableScan or IndexScan operators. For IndexScan operators, you must provide the table name, attribute, low and high keys; here is an example of another physical plan for another query where an IndexScan is involved.

```
SELECT S.A FROM Sailors S, Reserves R
WHERE S.B = R.G AND R.H < 100 AND S.A >= 9050;
```

```
Project[S.A]
-SMJ[R.G = S.B]
--ExternalSort[R.G]
---Select[R.H <= 99]
----TableScan[Reserves]
--ExternalSort[S.B]
---IndexScan[Sailors,A,9050,10000]
```

In this plan the high key for the index scan is `10000` because the physical plan builder in the reference implementation accessed the `stats.txt` file and determined the max value of attribute `Sailors.A` is `10000`. If your implementation does not do this, it is ok to leave the high key as `null` since no high key is specified in the query.

3 Implementation instructions

3.1 Gathering statistics and refactoring old code

Start by writing the functionality for creating a `stats.txt` file from your database as described in Section 2. This is fairly straightforward. Also, make sure your database catalog is able to provide this statistics info on demand to any other components of your code.

It is recommended that you improve your data generator from Project 3 so that it can read a `stats.txt` file and generate data with the statistical properties described in that file. This will help with testing later on.

Also, this is a good time to refactor your code so that you can handle the case where you have more than one index per relation. In Project 4 we assumed there was at most one index per relation; this assumption will no longer hold here. You may still assume that all indexes will still be on a single attribute and that a relation has either one or zero indexes on each attribute. At most one index on the relation can be clustered. You may assume that `index_info.txt` will list any clustered index on a relation before any unclustered indexes on the same relation.

The above means your interpreter code needs to be able to build more than one index per relation, and your DB catalog needs to have internal data structures that "know" about all indexes for each relation.

3.2 Pushing selections

Your first nontrivial task is pushing selections. This should be done early in the life cycle of your query, i.e. while building the logical plan. The reason is that in our setting, pushing selections is always a desirable optimization regardless of data properties, so we should do it as early as possible.

Your task is to refactor your logical plan builder so that it:

- no longer chooses a join order; that is, your logical join operator should no longer have just two children. It should have as many children as there are tables in the `FROM` clause, and the children should be listed in the same order as in the `FROM` clause.
- maximally pushes selections following the algorithm described below.

This logical plan will then be passed to the physical plan builder, which will choose a join order, build a left deep join tree, and choose an implementation for each join and each selection.

Refactoring to achieve the first goal above should be straightforward, so we focus on selection pushing.

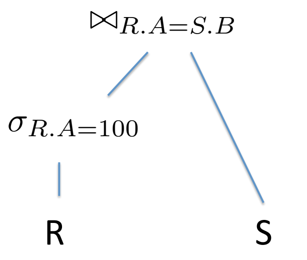
3.2.1 Which selections to push

You may already have done some selection pushing in previous projects, where you pushed selections down past the joins. However, in this project you will push selections much more aggressively, including pushing them both up and down the tree.

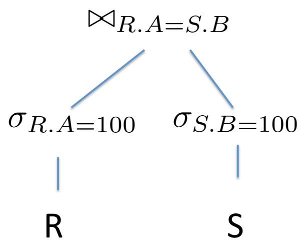
The rationale for pushing selections up the tree is the following. Consider the query

```
SELECT * FROM R, S WHERE R.A = 100 AND R.A = S.B
```

If you push the selection on `R.A`, you would end up with this tree:



However, note that if `R.A` is constrained to be equal to 100, so is `S.B`, because of the equality constraint `R.A = S.B`. This means an equivalent and more efficient plan can be obtained by adding a selection on `S.B` as well:



In some sense we want to push the selection back up the tree and then down on all possible paths. Another way to think about it is that we want to propagate the selection through all possible equality constraints.

Your job is to push selections as suggested in the above example. Specifically, you need to do the following.

Consider any constraint in your `WHERE` clause of the form *att OP val*, where *att* is an attribute, *val* is an integer value, and *OP* is one of `=, <, <=, >=, >`. Suppose you have a set of equality constraints of the form $att = att_1$, $att_1 = att_2, \cdots att_{k-1} = att_k$. Then you must infer a constraint att_i *OP val* for every i in the chain.

For example if your `WHERE` clause consists of `R.A < 100 AND R.A = R.B AND R.B = S.C AND S.C > 50 AND S.D = 42 AND S.D = T.F`, you should infer all the following selection conditions: `R.A < 100, R.A > 50, R.B < 100, R.B > 50, S.C > 50, S.C < 100, S.D = 42, T.F = 42`.

You are *not* required to propagate selections through constraints of the form att_1 *OP* att_2, where att_1 and att_2 are attributes and OP is anything other than equality. One might imagine that if you know `R.A > 5 AND S.B > R.A` you could infer `S.B > 5` or even possibly `S.B > 6`; however, this kind of constraint propagation adds sufficient complexity that you are not required to implement it. In addition, queries that compare attributes by something other than equality are quite rare, and in practice it makes the most sense to optimize the common case.

You are also not required to do anything about constraints using a not-equals (`!=` or `<>`) operator. If you have a comparison such as `R.A <> 5` you are not required to propagate it to any other attributes, and if you have a comparison such as `R.A <> R.B` or `R.A <> S.B` you are not required to propagate any other constraints through it.

Thus, we distinguish between *usable* comparisons and *unusable* comparisons in the `WHERE` clause. A usable comparison is an equality comparison between two attributes or any comparison between an attribute and a value that does NOT use <> (or ! =). Any other comparison is *unusable*.

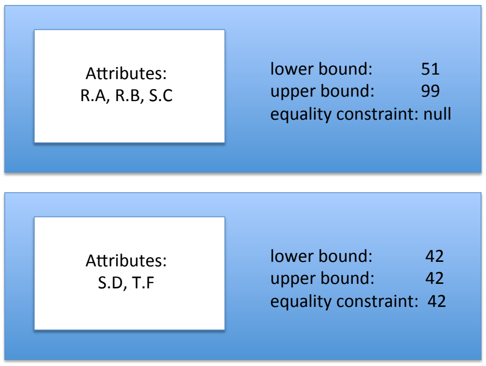
3.2.2 Implementing selection pushing with union-find

To propagate selections through equalities, you need to implement and use a *union-find* data structure. This is also known as a disjoint-set data structure and is described online in various sources.

Fundamentally, a union-find is a collection of elements which are disjoint sets. The idea is that you will partition all attributes that are featured in some usable comparison the `WHERE` clause into subsets based on equality constraints between them. The union-find allows you to keep track of this partitioning. E.g. if you have `WHERE R.A = R.B and S.C = T.D AND T.D = U.F` then your union-find will have two elements, the first containing `R.A` and `R.B`, and the second containing `S.C, T.D` and `U.F`.

In every union-find element, you will also keep track of three numeric constraints: any known lower bound for every element in the set, any known upper bound, and any known equality constraint. If there is a known equality constraint, the lower and upper bound must be set to that equality constraint as well.

Your goal is to process the `WHERE` clause and build up a union-find; once your processing is done, the union-find contains all the information you need to push selections. For our example clause from before, `R.A < 100 AND R.A = R.B AND R.B = S.C AND S.C > 50 AND S.D = 42 AND S.D = T.F`, your union-find that you build should look like this:



Every blue box is a single element in the union-find.

Now, how do you build the union-find from the `WHERE` clause? Start by writing a class that implements a union-find data structure. It should contain a collection of elements, where each element consists of a set of attributes and three numeric constraints. The API you need for your union-find is as follows:

- given a particular attribute, *find* and return the union-find element containing that attribute; if no such element is found, create it and return it.
- given two union-find elements, modify the union-find data structure so that these two elements get *union*ed, i.e. merged.
- given a union-find element, set its lower bound, upper bound, or equality constraint to a particular value.

Now, write a visitor that walks the `WHERE` clause and builds up a union-find. It should process each comparison as follows:

- if the comparison has the form $att1 = att2$, *find* the two elements containing $att1$ and $att2$ and *union* them.
- if the comparison has the form $att\ OP\ val$, where val is an integer and OP is $=, <, <=, >=, >$, *find* the element containing att and update the appropriate numeric bound in the element
- if the comparison has any other form, it is an unusable comparison; put it aside for separate processing later as it does not go into the union-find.

At the end of this process, you have a union-find that captures a lot of constraints. You may also have some "residual" unusable comparisons.

You may assume we will not give you "bad" where clauses which would generate inconsistent constraints of any sort (e.g. `R.A = 1 AND S.B = R.A AND S.B = 2`), and that all comparisons in the `WHERE` clause will involve one or two attributes. That is, you will not encounter comparisons of the form $val_1\ OP\ val_2$ with val_1 and val_2 both integers.

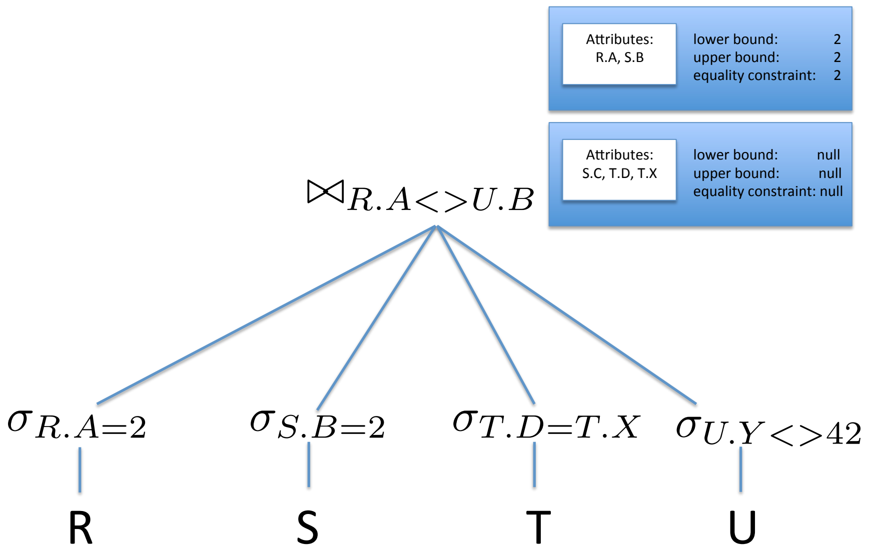
3.2.3 Bringing it all together

Now you are ready to use the union-find to generate a logical query plan with fully pushed selections. For every child of the logical join use the union-find to retrieve all numerical bounds/constraints that apply to attributes of that child. Make a selection operator with those constraints. In addition, make sure to add to that selection any "residual" unusable comparisons that couldn't go into the union-find and that relate just to that child.

This leaves you with some state that needs to be placed inside the logical join operator. This state is the union-find data structure which you will need later, and any residual unusable comparisons that connect attributes from different tables. For example, for the query

```
SELECT *
FROM R, S, T, U
WHERE R.A <> U.B AND R.A = S.B AND S.C = T.D AND R.A = 2 AND T.D = T.X AND U.Y <> 42
```

You want the following logical query plan:



Note that the $R.A <> U.B$ is not the whole join expression, it is just the remaining unusable comparison that will have to go into a join somewhere but did not make it into the union-find.

In summary, the overall structure of your logical plan should now be as follows (all operators are optional except at least one leaf). The topmost operator is a duplicate elimination operator, followed by a sort, a projection, and a single logical join operator. The children of the join are selections or leaves as appropriate.

3.3 Choosing selection implementations

Now that you have an optimized logical plan, the rest of your work will be in the physical plan builder. As mentioned in Section 2, the `PhysicalPlanBuilder` configuration file is no longer needed. Previously, you told the plan builder whether to use indexes for selections, and what implementation of join/sort to use. Now, the plan builder will make those decisions by itself. You may hard-code in the `PhysicalPlanBuilder` the choice of sorting algorithm (use external sort unless you were not able to implement it) as well as the number of buffer pages to be used for BNLJ.

Your next task is to have your `PhysicalPlanBuilder` choose an implementation for each selection. This is simliar to what you did for Project 4, except that now you may have multiple indexes to choose from and you need to pick the best access path (index or scan) based on estimated cost.

For each selection and each available access path, you should estimate the cost in number of I/O's. For the scan, you can estimate the cost by asking your friendly database catalog how many tuples the base table has, multiplying by the size of one tuple, and dividing that by the page size which is still 4096 bytes.

For using an index, estimate the cost as follows.

Start by computing the reduction factor for the portion of the selection that is handled via the index. This is the range of values being selected divided by the total range of values for the attribute. For example if your selection condition contains `R.A >= 10 and R.A < 21` then there are 11 possible values in the selection range; if your catalog tells you the values in the base relation `R` go from 0 to 109, the total range of values is 110, the selection covers one-tenth of that range, and your reduction factor is 0.1. Of course you need to have some way to compute range boundaries if the selection only specifies an upper bound and no lower bound, or vice versa.

Now that you have the reduction factor, you can compute the cost of the index scan as follows. Assume the cost of root-to-leaf traversal is 3. If the index is clustered, you can go directly to the first relevant data page and then need to access a fraction of all the data pages as determined by the reduction factor. If the index is unclustered, you need to scan a fraction of the leaves (determined by the reduction factor) and then in the worst case one page I/O for every tuple that matches the selection.

Thus, if p is the number of pages in the relation, t the number of tuples, r the reduction factor and l the number of leaves in the index, the cost for a clustered index is $3 + p * r$ while for an unclustered index it is $3 + l * r + t * r$.

The above computations require knowledge of how many leaf pages each index contains. Your database catalog can help with that - remember, each index stores the number of leaves on its header page.

Once you know the cost for each index, choose the best access path (which may be an index or a scan) and implement the selection appropriately. As in Project 4, you may end up with a two-part implementation where a portion of the selection is handled via index scan and the remainder via a full-scan select operator that has the index scan as its child.

3.4 Choosing a join order

Your physical plan builder receives a single logical join operator with arbitrarily many children, and needs to translate this into a "real" left-deep join tree. There are two parts to this task: choosing a join order, and choosing an implementation for each join. This section explains the former, Section 3.5 the latter.

3.4.1 Dynamic programming for choosing join orders

To choose a join order, you will use a variant of the dynamic programming algorithm discussed in the 4320 lectures and textbook. The general idea is as follows.

You are only considering left-deep trees, so given k relations to join you have k options for the topmost (last) relation in the join. Suppose your relations are $R_1, R_2, \cdots R_k$. Suppose you choose R_i as the topmost relation. Then you have $k-1$ relations remaining to join together. Whatever join order you choose for those, it should be the one with the lowest cost (more on computing costs later). Suppose that cost is c. Then the total cost of your entire plan is c plus the cost of the last join with R_i. You need to check all k possible options for R_i and pick the one which gives the lowest total cost.

Now, how do you compute the lowest-cost plan for the $k-1$ remaining relations? You can recursively pick one of the remaining $k-1$ relations as the topmost one, and continue. Of course, this would be grossly inefficient. Therefore, instead of computing costs "top-down" recursively for smaller and smaller subsets of relations, you need to use a *dynamic programming* approach and compute the costs "bottom-up" for larger and larger subsets.

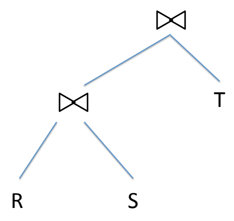
Thus, given a set of relations $\mathcal{R} = \{R_1, R_2, \cdots R_k\}$, you need to iterate over all subsets of $\mathcal{R}$, in increasing order of size. I.e. start with all singleton subsets $\{R_1\}, \{R_2\}$, etc, then move on to pairs, etc. For every subset, you need to compute and store *cost of the best plan*, and the best plan itself, i.e. the join order associated with that lowest cost.

3.4.2 Computing plan costs

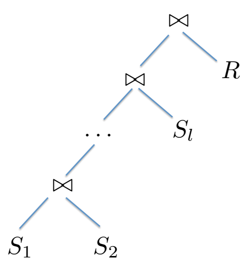
A crucial part of the above algorithm is a method to compute the cost of a particular plan, i.e. a particular left-deep join tree. In CS 4320 you saw very detailed cost calculations based on number of page I/Os; here, you will use a coarser notion of cost which is simpler to compute. Remember, all cost calculations are approximate anyway, and all that really matters is that we use a cost metric that correctly *ranks* the possible plans with respect to each other.

With this in mind, we will define the cost of a plan as the sum of the sizes, in tuples, of all intermediate relations not counting the final answer. That is, the cost of a one-relation plan is zero. The cost of a two-relation plan is zero; however, **you should choose the smaller relation to be the outer in the best plan**; as you know the size of the outer has a direct impact on the I/O cost of the join, whether you use BNLJ (the size of the outer determines the number of scans of the inner) or SMJ (the outer needs to be materialized so we can sort it).

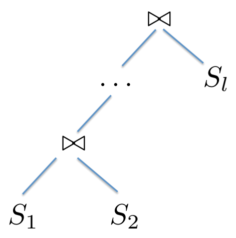
For three relations and up, things become more regular. The cost of a three-relation join tree like this



is the size of $R \bowtie S$ in tuples. More generally, the cost of a plan of the form:



is the size of $S_1 \bowtie S_2 \cdots \bowtie S_l$ plus the cost of the subplan not involving R, that is,



To make this more concrete with an example, suppose you are joining R, S, T and U in that order.

The cost of $R \bowtie S$ is 0 by definition/assumption.

The cost of $(R \bowtie S) \bowtie T$ is $|R \bowtie S|$. This can be computed as the cost of the left child (zero in this case) plus the size of the relation produced as the result of the left child, i.e. $R \bowtie S$.

The cost of $((R \bowtie S) \bowtie T) \bowtie U$ is $|R \bowtie S| + |(R \bowtie S) \bowtie T|$. This is the cost of the left child, i.e. $|R \bowtie S|$ as computed previously, plus the size of the relation produced as the result of the left child, i.e. $(R \bowtie S) \bowtie T$.

Thus the total cost of the plan $((R \bowtie S) \bowtie T) \bowtie U$ is the sum of the sizes of the intermediate relations not including the final result, because there are two intermediate relations, namely $R \bowtie S$ and $(R \bowtie S) \bowtie T$.

This cost metric is coarser than what you saw in 4320, but it is likely to be sufficient and still rank the plans correctly. The rationale is that the intermediate relations whose sizes we include in the cost each act as the outer relation in some join. By the same argument as we gave for two-relation subsets, the size of the outer relation in the join has a big influence on the cost of the join, whatever the join algorithm used and whether or not the outer is materialized during computation.

From the above discussion, it should be clear that costs are easy to compute in a bottom-up fashion using the dynamic programming algorithm above, as long as you also keep track for each subset of relations how big their join is. This last piece of the algorithm is discussed next.

3.4.3 Computing intermediate relation sizes

As you know, the size of the join of two relations can vary from 0 to the size of the product of the two inputs, and we would like something more precise than either of these extremes. In this project, we will use an algorithm based on data statistics. For more information on this algorithm, see the textbook *Database systems : the complete book*, by Hector Garcia-Molina, Jeffrey D. Ullman and Jennifer Widom. The book is available on course reserve in the Uris Library, and the relevant chapter is Chapter 16.

We start by introducing a new kind of statistical information. For relation R and attribute A, $V(R, A)$ is the number of distinct values that A takes in R. We call those statistics V-values (the textbook cited above doesn't call them that, but we will, in order to refer to them more easily).

We will explain how to compute V-values soon. For now, let's explain how you can use them to compute the size in tuples of the join of two relations R and S.

Let's start with the simplest case where R and S are joined on just one pair of attributes, say $R.A = S.A$. We are going to assume a correlation between $R.A$ and $S.A$ as follows:

- if $V(R, A) \leq V(S, A)$ then every value of $R.A$ appears as a value of $S.A$
- if $V(R, A) \geq V(S, A)$ then every value of $S.A$ appears as a value of $R.A$

The underlying assumption is that $R.A$ and $S.A$ are the "same attribute", taking values from the same domain, and that both R and S choose their values from the front of a fixed list of values from the domain.

This is realistic in practice, because we normally join tables on attributes that represent the same real-world data. In particular, most joins are primary key-foreign key joins. Every possible value that appears as a foreign key in one table is – by definition – found as a primary key in the other table, and the table with the primary key is the one with the larger V-value.

This assumption helps us estimate the cost of joining R and S. Let $|R|$ and $|S|$ be the sizes of R and S in tuples, respectively.

First, suppose $V(R, A) \leq V(S, A)$. Then every tuple in R has a $\frac{1}{V(S,A)}$ chance of joining with any specific tuple from S, so in expectation it joins with $\frac{|S|}{V(S,A)}$ tuples from S. Since there are $|R|$ tuples in R, the total number of tuples in the join result is $\frac{|R||S|}{V(S,A)}$. By a symmetric argument, if S is the relation with the larger V-value, i.e. $V(R, A) \geq V(S, A)$, we expect the join to contain $\frac{|R||S|}{V(R,A)}$ tuples. Thus in the general case, the join size is:

$$\frac{|R||S|}{max(V(R, A), V(S, A))}$$

Now, suppose you have an equality join on more than one attribute, such as $R.A = S.A \wedge R.B = S.B$. If you work through the above argument for join sizes and calculate the probabilities that a tuple from R will match a tuple from S on both A and B, you will find that the expected join size is now

$$\frac{|R||S|}{max(V(R, A), V(S, A))max(V(R, B), V(S, B))}$$

i.e., we take the product of the maximum V-value for each attribute for the denominator. (This argument is developed in more depth in the textbook mentioned above).

Finally, if your join condition contains other than equality constraints, such as $R.A < S.A$, they do not significantly reduce the size of the join from the cross product. Thus, if you have a join with both equality and other-than-equality comparisons, disregard the other-than-equality comparisons in your size computation. If you have a join that has only other-than-equality constraints, consider that join as a cross product; its size is the product of the sizes of the two inputs.

3.4.4 Computing V-values

We are narrowing down the last remaining challenge to the computation of V-values. Once you know this, you can estimate join sizes, calculate costs of plans, and find the best join order. So, how do you compute V-values for a relation? There are three cases, depending on whether the relation is a base table, the result of a selection, or the result of a join.

Suppose the relation is a **base table** R. If you assume uniform distribution, and if you know the minimum (min) and maximum (max) values that A takes in R, $V(R, A)$ is just $max - min + 1$.

If the relation is a **selection on a base table** R, start with the V-values from R and adjust them based on the reduction factors and ranges for the selection. If an attribute is mentioned in the selection, its V-value shrinks by an appropriate reduction factor that is computed as in Section 3.3. Some expressions that you may encounter in a selection conditions may not have obvious reduction factors associated with them, for example comparisons like `R.A != 42` or `R.A > R.B`. For those comparisons (basically, those that use not-equals and/or compare two attributes to each other) you may disregard them here; such comparisons are rare in practice and we are optimizing for the common case. Assume that all attributes of each relation are independent, so a selection on one attribute has no effect on the V-values of other attributes.

Also note that you can compute the size of R as the size of the base table multiplied by the product of all the reduction factors on all the attributes mentioned in the selection (we are assuming independence of all attributes from each other).

A technical point: if a relation has r tuples, none of its V-values can be larger than r, so once you know the total number of tuples in the selection result you may need to adjust your computed V-values downwards. E.g. suppose you have a relation R where A has values from 0 to 999. So you have an original V-value for $V(R, A)$ of 1000. Now suppose your selection on A is $A < 500$, so your computed V-value for A after the selection is 500. However, suppose R also has a second attribute B which is mentioned in the selection and has its own reduction factor, and your estimate of the final size of the selection is 100 tuples. Then $V(A)$ should be set to 100 instead of 500.

Also, no relation size or V-value should be 0, so if your calculations yield very small sizes or V-values, round them up to 1.

The final case is when you have a relation that is a **join of two other relations**, say R and S. How do you compute V-values for the join from the V-values of R and S?

Consider a particular attribute A in the join of R and S. There are two cases: either A is mentioned in the join condition or it is not. As before, we assume the join condition consists only of equality comparisons; that is, if A appears in the join condition but only in an other-than-equality comparison, we will treat it as though it didn't appear in the join condition.

If A does appear in the join condition, its V-value is the minimum of all the V-values of the attributes that it gets equated with. For example, if you have a condition $R.A = S.A \wedge R.A = S.B$, each of the three attributes $R.A, S.A, S.B$ has the same V-value in the join result, and that V-value is the minimum of the three original V-values for $R.A, S.A, S.B$.

If A does not appear in the join condition, we assume that its V-value is preserved, i.e. it has the same V-value as it did in whichever relation it came from. This assumption is true for all primary key-foreign key joins, and is realistic in practice even for many other joins.

Finally, as for selections, remember that no V-value may be bigger than the number of tuples in the relation, so V-values obtained via the above computations may need to be "clamped down" based on your estimate of the join size.

3.4.5 Putting it all together

Your goal is to implement the above dynamic programming algorithm and find the best join order for the children of your logical join operator. Obviously you want to do this in a separate class from the main physical plan builder, since a lot of logic and data structures will be involved.

3.5 Choosing join implementations

Now that you have a join order chosen, you are almost done; you just need to choose an implementation for each join operator and actually build your left-deep tree to complete your physical plan.

Every join should be implemented either as a SMJ or BNLJ. You are required to have some strategy for choosing between SMJ and BNLJ. The strategy must be nontrivial, that is, you may not simply implement every join as SMJ (or as BNLJ). Your plan builder must make a choice somewhere based on some factors. You must explain your strategy very clearly in your comments and README, giving a rationale why you believe it's a good strategy.

An example strategy and rationale might be as follows: "In my Project 3 benchmarking, I noticed that SMJ runs much faster than BNLJ, so I implement all joins as SMJ where possible. However, SMJ does not apply to joins that have other-than-equality comparisons or to pure cross-products, so those are implemented using BNLJ".

4 Grading

For this project, the must-have requirements are as follows:

- You must do selection pushing on the logical query plan and the remainder of the optimizations while building the physical query plan from the logical query plan. In particular this implies your logical query plan should contain a single join operator with potentially more than two children.
- You must implement each of the optimization algorithms/heuristics as we have described them. Minor variations may be permitted (get permission by asking on Piazza or in office hours), as long as you explain to us why your variation/improvement is a good idea, and as long as it is clear you understand the original algorithm and are not trying to get out of implementing it.

Next we give the grading breakdown.

4.1 Code style and comments (10 points)

As in previous projects, you must provide comments for every method you implement. At minimum, the comment must include one sentence about the purpose/logic of the method, and `@params`/`@return` annotations for every argument/return value respectively. In addition, every class must have a comment describing the class and the logic of any algorithm used in the class. As in previous projects, if you follow the above rules and write reasonably clean code that follows our overall architecture, you are likely to get the full 10 points for code style.

4.2 Automated tests (65 points)

We will grade your code by running it on our own dataset with 10 queries. We will look at your generated `stats.txt` file and check that matches ours, for 5 points. The remaining 60 points are allocated at six points per query:

- (2 points) your logical plan matches our expected logical plan
- (2 points) your physical plan is reasonable given the data and your join/selection choice heuristics
- (2 points) your query answers match our expected query answers

When we grade your physical plans, your plans do not need to match ours exactly, but we are looking to see that you produce the "obvious" good plans in very clear-cut cases. E.g. if we give you a selection with a matching clustered index and a very narrow selection range (high reduction factor), we expect to see you using the index; if we give you a join of two tiny relations and a third huge one, we expect to see the third one as the last one in the join order, etc.

5 Submission instructions

Create a README text file containing the following information. **Submissions without a README will receive zero points.**

- a line stating which is the top-level class of your code (the interpreter/harness that reads the input and produces output).
- for each of the below algorithms/functionalities, an explanation of where the implementation is found (i.e. which classes/methods perform it), as well as an explanation of your logic, especially if your logic diverges in any way from the instructions. If your logic is adequately explained in comments in your code, you may provide a reference to the comment rather than copying the comment:
  - the selection pushing
  - the choice of implementation for each logical selection operator
  - the choice of the join order
  - the choice of implementation for each join operator
- any other information you want the grader to know, such as known bugs.

Submit via CMS a .zip archive containing:

- your Eclipse project folder
- a .jar of your project that can be run on the command line; name this `cs4321_p5.jar`
- your README file
- an acknowledgments file if you consulted any external sources, as required under the academic integrity policy