Optimisation Strategy

Oushuo Huang

1 Initialisation

In this implementation, `Optimiser` is also a subclass of `PlanVistor`. By applying different visitor methods, it can collect operators' information on three global variables:

- `Set<Predicate> predicateSet`: a set of all predicates in the query, which will be used in reordering joins and moving down σ steps.
- `Set<Scan> scanSet`: a set of all lowest-level `Scan` operators in the query, which will be used for the reordering join step.
- `List<Attribute> finalAttributes`: the top-level `Project` operator's attributes, which will be used in moving down π step.

After defining these three global variables, it will initialise the three global variables and start optimisation by calling `plan.accept(this)`.

2 Moving Down σ

The idea of this step is to iterate each operator in the lowest level and find suitable predicates with matching attributes. If the form is `attr = value` then it only need to consider the left attribute, otherwise, it will consider both left and right attributes.

3 Reordering Joins

To optimise the order, I need to go through the operator list created from the previous step and find out the smallest cost from all possible orders.

Since the tuple count and distinct value count of each operator will be updated from the previous `Join` or `Product` operator, I use the idea of dynamic programming which can store the intermediate results and avoid repeated computation. The input is the operator list after moving down σ, so that each operator is the smallest element to be joined or producted.

```
Queue<Map.Entry<Operator, List<Operator>>> joinQueue = new LinkedList<>();

Queue<Set<Predicate>> predicateQueue = new LinkedList<>();
```

The following image shows the processes of this function.

Figure 1: Data structure for reordering.

Note that in the practical implementation I use a queue with `Map.Entry`, so the actual effect is the same as 3 queues.

- If the first queue is empty, it will iterate the second queue and add the operator to the first queue.
- If the first queue is not empty, it will again iterate the second queue and try to perform a `Join` or `Product` based on the predicates in the third queue. Meanwhile, after each opeation, the function will call `Estimator` to update the tuple count and distinct value count of the new operator.
- If the remaining operators are 0, which means a plan has been found, it will compare the cost with the current minimum cost and update the minimum cost if necessary.
- No matter what happens, the function will pop the first element in each queue. If the queue is empty, which means all possible plans have been checked, then it will return the minimum cost.

4 Move Down π

To improve the performance, I integrate this step within the reordering step, because reordering will go through all operators and predicates from the lowest level.

The idea is, each operator can only keep the attributes that the remaining predicates have. To do so, the function `MoveDownProject` will output a `Project` if there is an overlap between the original operator's attributes and the remaining predicates' attributes.

The following figures are the proposed plan and the actual output of the `Q5`.

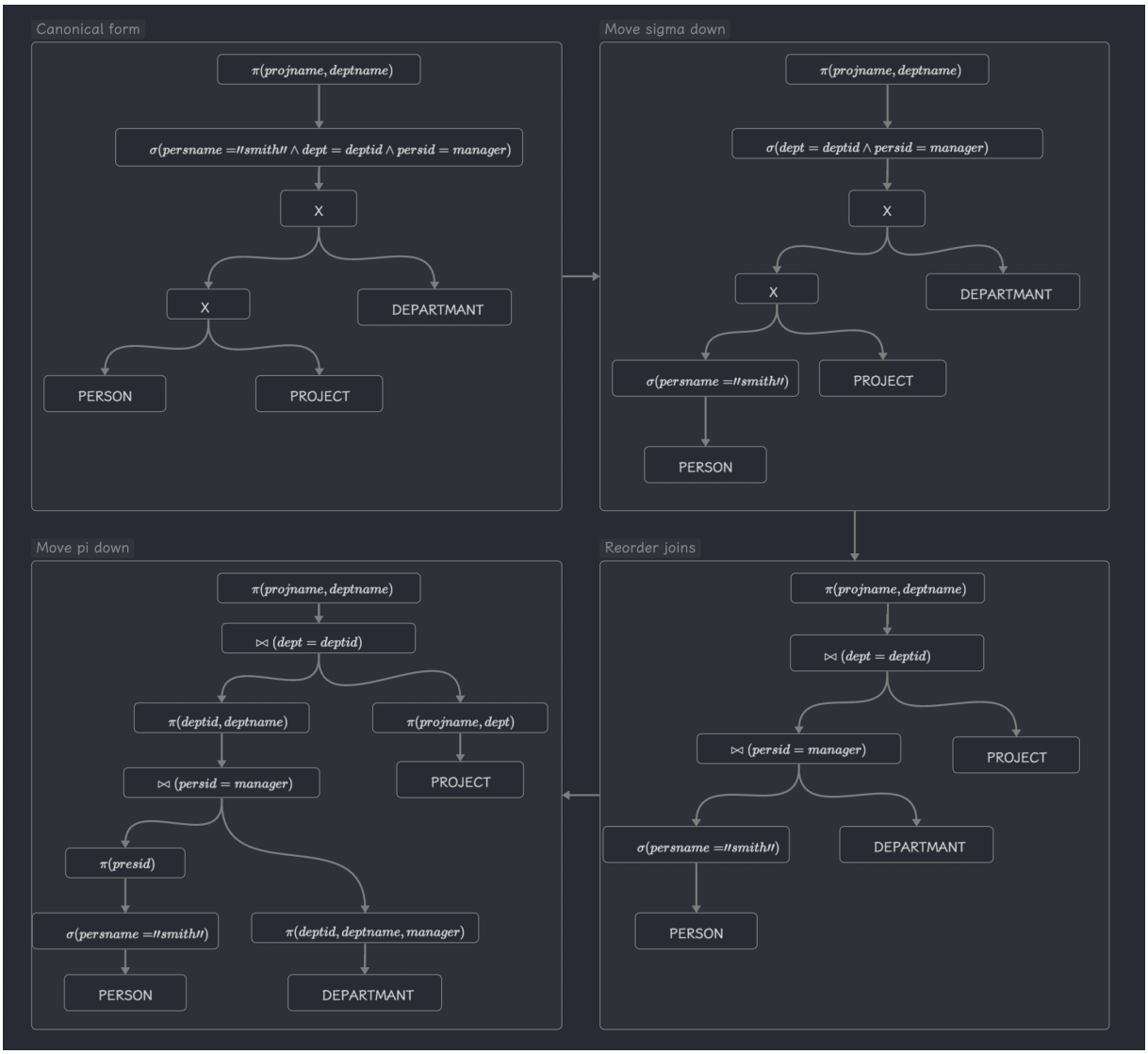

Figure 2: Proposed plan for the query.

```
PROJECT [projname,deptname] ((PROJECT [projname,dept] (Project)) JOIN [dept=deptid]
(PROJECT [deptid,deptname] ((PROJECT [persid] (SELECT [persname="Smith"] (Person)))
JOIN [persid=manager] (PROJECT [manager,deptid,deptname] (Department)))))
```