Chair of applied computer science III

UNIVERSITY OF MANNHEIM

Prof. Dr. Guido Moerkotte

Email: moerkotte@uni-mannheim.de

Query Optimization Exercise sheet 8

Exercise 1

Exercise 1 a)

Read about *computer algebra*, also known as *symbolic computation*.
Wikipedia is your friend:
`https://en.wikipedia.org/wiki/Computer_algebra`

Exercise 1 b)

Implement `ExhaustiveTransformation`. You may use the helper classes provided in the solution code.

Solution

See Code.

Exercise 1 c)

Discuss ways implement non-exhaustive termination criteria for `ExhaustiveTransformation`.

Solution

Introduce counters for number of trees in `Done`, number of calls to `ApplyTransformations`, number of trees returned returned by all `ApplyTransformations` executions so far. Terminate when (combination of) counters hit a defined maximum.

Exercise 1 d)

Discuss how to adjust `ExhaustiveTransformation` to avoid cross products. (See script.)

Solution

Applying associativity can lead to cross products. If we do not want to consider cross products, we only apply any of the two associativity rules if the resulting expression does not contain a cross product. It is easy to extend `ApplyTransformations` to cover this by extending the if conditions with
`and (ConsiderCrossProducts||connected(·)`
where the argument of connected is the result of applying a transformation.

Exercise 1 e)

BONUS
Read and present in the next exercise session:
`https://pdfs.semanticscholar.org/5d91/f42e767c167ecf188a46608802ba5fe52347.pdf`

Exercise 2

Update the classification table for join algorithms from one of the previous exercises with the following algorithms:

- Transformation-based approach
- Memoization
- Generate permutations
- Quick Pick
- Iterative Improvement
- Simulated Annealing
- TabuSearch

Solution

Name	Query Graph	Join-Tree	Cross products	Cost functions	Complexity	optimal	Remarks
Permutations	arbitrary	left-deep	yes	arbitrary	$O(2^n)$	yes	
Memoization	arbitrary	bushy	optional	bleiebig	$O(2^n) \dots O(3^n)$	yes	complexity depends on query graph
Transformation	depends on rule set			arbitrary	$O(2^n) \dots O(3^n)$	yes	complexity depends on rule set
QuickPick	connected	bushy	no	arbitrary	$O(n)$	no	complexity per iteration
IterImp	depends on rule set			arbitrary	$O(n)$	no	complexity per iteration
SimAnn	depends on rule set			arbitrary	$O(n)$	no	complexity per iteration
TabuSearch	depends on rule set			arbitrary	$O(n)$	no	complexity per iteration

Exercise 3

Find reasonable plans for the following using queries using indices. Explicitly state the assumptions you make regarding the indices.

Exercise 3 a)

```
select a.v
from   a a
where a.w > 10
  and a.w < 30
```

Solution

- Index-scan based on lower bound (10) and upper bound (30).
- (B-tree) index on attribute w must exist.

Exercise 3 b)

```
select *
from a
where exists (select *
              from b
              where a.v = b.key)
```

Solution

- Index-scan on relation b. Index for key attribute *b.key* must exist.
- Since the join predicate is a comparison based on the keys of b, we can efficiently apply an Index-Nested-Loop-Join or a D-Join. Ideally, Relation a is sorted by $a.v$ in ascending order before the join.

  $\sigma_{0<count(\sigma_{a.v=b.key}(B[b]))}(A[a])$

  $\sigma_{0<count}(\chi_{count:count(\sigma_{a.v=b.key}(B[b]))})(A[a])$

  $\sigma_{count>0}(A[a] < \chi_{count:count(B_{key}[x;key=a.v])}(\Box) >)$

Exercise 3 c)

```
select a.b, min(a.c)
from a
group by a.b
```

Solution

- Index-scan on attributes $a.b, a.c$ of relation a must exist.
- Start with a minimal value for $a.b$. Then use gap-skipping to find the next value of $a.b$. Each time, The first value found is the min(a.c) for the current a.b.

Gap skipping (taken from the script): Sometimes also called zig-zag skipping, continues the search for keys in a new keyrange from the latest position visited. The implementation details vary but the main idea of it is that after one range has been completely scanned, the current (leaf) page is checked for its highest key. If it is not smaller than the lower bound of the next range, the search continues in the current page. If it is smaller than the lower bound of the next range, alternative implementations are described in the literature. The simplest is to start a new search from theroot for the lower bound. Another alternative uses parent pointers to go up a page as long as the highest key of the current page is smaller than the lowerbound of the next range. If this is no longer the case, the search continues downwards again.

Exercise 4

For each of the following queries, write down one plans with explicit attribute accesses and another one with implicit attribute accesses.

Exercise 4 a)

```
select s.name, s.matrnr
from studenten s
where s.age>27 and
      count(s.hoert)=0
```

Solution

implicit : $\Pi_{matrnr}(\sigma_{count(hoert)=0}(studenten[s; age > 27]))$

explicit : $\chi_{matrnr:s.matrnr}(\sigma_{count=0}(\chi_{count:count(hoert)}(\chi_{hoert:s.hoert}(\sigma_{age>27}(\chi_{age:s.age}(studenten[s]))))))$

Exercise 4 b)

```
select s.name, s.matrnr
from studenten s
where s.age>27 and
      count(s.hoert)<5 and
      exists v in s.hoert:
         v.name="AO"
```

Solution

implicit : $\Pi_{matrnr}(\sigma_{\exists_{v \in hoert}: v.name=''AO''}(\sigma_{count(hoert)<5}(studenten[s; age > 27])))$

explicit : $\chi_{matrnr:s.matrnr}(\sigma_{\exists_{v \in hoert}: v.name=''AO''}(\sigma_{count<5}(\chi_{count:count(hoert)}(\chi_{hoert:s.hoert}($
$\sigma_{age>27}(\chi_{age:s.age}(studenten[s])))))))$