Extending the Optimizer

Ralf Hartmut Güting

Fernuniversität Hagen, Germany

Secondo Extensions

1. New algebras for attribute types:
   - Write a display function for a new type constructor to show corresponding values.
   - Define the syntax of a new operator to be used within SQL and in SECONDO.
   - Write optimization rules to perform selections and joins involving a new operator, including rules to use appropriate indexes.
   - Define a cost function or constant for using the operator.
2. New query processing operators in the relational algebra:
   - Define the operator syntax to be used in SECONDO.
   - Write optimization rules using the new operator.
   - Write a cost function (predicate) for the new operator.
3. New types of indexes:
   - Define the operator syntax to be used in SECONDO for search operations on the index.
   - Write optimization rules using the index.
   - Write cost functions for access operations.

Extension Tasks

- Writing a display function for a type constructor
- Defining operator syntax for SQL
- Defining operator syntax for SECONDO
- Writing optimization rules
- Writing cost functions

Writing a Display Predicate for a Type Constructor

```
display(Type, Value) :-
```

Display the `Value` according to its `Type` description.

```
display(int, N) :-
  !,
  write(N).


display([rel, [tuple, Attrs]], Tuples) :-
  !,
  nl,
  max_attr_length(Attrs, AttrLength),
  displayTuples(Attrs, Tuples, AttrLength).


displayTuples(_, [], _).

displayTuples(Attrs, [Tuple | Rest], AttrLength) :-
  displayTuple(Attrs, Tuple, AttrLength),
  nl,
  displayTuples(Attrs, Rest, AttrLength).
```

```
displayTuple([], _, _).

displayTuple([[Name, Type] | Attrs], [Value | Values], AttrNameLength) :-
  atom_length(Name, NLength),
  PadLength is AttrNameLength - NLength,
  write_spaces(PadLength),
  write(Name),
  write(' : '),
  display(Type, Value),
  nl,
  displayTuple(Attrs, Values, AttrNameLength).
```

Defining Operator Syntax for SQL

Atomic operators working on attribute types like

```
+, <, mod, inside, starts, distance, ...
```

can be used directly in SQL.

- explain PROLOG syntax
- explain translation to Secondo syntax

Several cases:

- syntax is predefined in PROLOG, e.g. for

```
+, -, *, /, <, >
```

- write in prefix syntax (always possible)
- define infix syntax:

```
:- op(800, xfx, adjacent).
```

Defining Operator Syntax for SECONDO

Query language terms written in prefix notation in the optimizer.

```
x y product filter[cond] consume
```

written as

```
consume(filter(product(x, y), cond))
```

For each operator, optimizer needs to know translation to SECONDO.

1. By default
   - 1 or 3 arguments: prefix syntax
   - 2 arguments: infix syntax

   ```
   length(x), x adjacent y, translate(x, y, z)
   ```

2. Syntax specification via predicate `secondoOp` in file `opsyntax.pl`

   ```
   secondoOp(distance, prefix, 2).

   secondoOp(feed, postfix, 1).
   secondoOp(consume, postfix, 1).
   ```

3. Programming a `plan_to_atom` rule.

```
plan_to_atom(X, Y) :-
```

`Y` is the SECONDO expression corresponding to term `X`.

```
plan_to_atom(sortmergejoin(X, Y, A, B), Result) :-
 plan_to_atom(X, XAtom),
 plan_to_atom(Y, YAtom),
 plan_to_atom(A, AAtom),
 plan_to_atom(B, BAtom),
 concat_atom([XAtom, YAtom, 'sortmergejoin[',
   AAtom, ', ', BAtom, '] '], '', Result),
 !.
```

Writing Optimization Rules

See what the optimizer does. Consider the query

```
select *
from staedte as s, plz as p
where s:sname = p:ort and p:plz > 40000
```

Exists as a predefined example in `optimizer.pl`

```
example5 :- pog(
  [rel(staedte, s, u), rel(plz, p, l)],
  [pr(attr(s:sName, 1, u) = attr(p:ort, 2, u), rel(staedte, s, u),
   rel(plz, p, l)),
   pr(attr(p:pLZ, 1, u) > 40000, rel(plz, p, l))],
  _, _).
```

Open database `opt` and then:

```
example5.
```

See the predicate order graph:

```
16 ?- writeNodes.
Node: 0
Preds: []
Partition: [arp(arg(2), [rel(plz, p, l)], []), arp(arg(1), [rel(staedte, s, u)],
[])]

Node: 1
Preds: [pr(attr(s:sName, 1, u)=attr(p:ort, 2, u), rel(staedte, s, u), rel(plz, p,
l))]
Partition: [arp(res(1), [rel(staedte, s, u), rel(plz, p, l)], [attr(s:sName, 1,
u)=attr(p:ort, 2, u)])]

Node: 2
Preds: [pr(attr(p:pLZ, 1, u)>40000, rel(plz, p, l))]
Partition: [arp(res(2), [rel(plz, p, l)], [attr(p:pLZ, 1, u)>40000]),
arp(arg(1), [rel(staedte, s, u)], [])]

Node: 3
Preds: [pr(attr(p:pLZ, 1, u)>40000, rel(plz, p, l)), pr(attr(s:sName, 1,
u)=attr(p:ort, 2, u), rel(staedte, s, u), rel(plz, p, l))]
Partition: [arp(res(3), [rel(staedte, s, u), rel(plz, p, l)], [attr(p:pLZ, 1,
u)>40000, attr(s:sName, 1, u)=attr(p:ort, 2, u)])]

Yes
17 ?-
```

```
17 ?- writeEdges.
Source: 0
Target: 1
Term: join(arg(1), arg(2), pr(attr(s:sName, 1, u)=attr(p:ort, 2, u),
rel(staedte, s, u), rel(plz, p, l)))
Result: 1

Source: 0
Target: 2
Term: select(arg(2), pr(attr(p:pLZ, 1, u)>40000, rel(plz, p, l)))
Result: 2

Source: 1
Target: 3
Term: select(res(1), pr(attr(p:pLZ, 1, u)>40000, rel(plz, p, l)))
Result: 3

Source: 2
Target: 3
Term: join(arg(1), res(2), pr(attr(s:sName, 1, u)=attr(p:ort, 2, u),
rel(staedte, s, u), rel(plz, p, l)))
Result: 3

Yes
18 ?-
```

Translating the Arguments

```
res(N) => res(N).

arg(N) => feed(rel(Name, *, Case)) :-
  isStarQuery,
  argument(N, rel(Name, *, Case)), !.

arg(N) => rename(feed(rel(Name, Var, Case)), Var) :-
  isStarQuery,
  argument(N, rel(Name, Var, Case)), !.
```

See what happens:

```
18 ?- assert(isStarQuery), example5.

Yes
19 ?- arg(1) => X.

X = rename(feed(rel(staedte, s, u)), s)

Yes
20 ?-
```

Translating Selection Predicates

Rule for filtering a stream:

```
select(Arg, pr(Pred, _)) => filter(ArgS, Pred) :-
  Arg => ArgS.
```

Rules for using a B-tree index:

```
select(arg(N), Y) => X :-
  isStarQuery,                      % no projection needed
  indexselect(arg(N), Y) => X.

indexselect(arg(N), pr(attr(AttrName, Arg, Case) = Y, Rel)) => X :-
  indexselect(arg(N), pr(Y = attr(AttrName, Arg, Case), Rel)) => X.

indexselect(arg(N), pr(Y = attr(AttrName, Arg, AttrCase), _)) =>
  exactmatch(IndexName, rel(Name, *, Case), Y)
  :-
  argument(N, rel(Name, *, Case)),
  hasIndex(rel(Name, *, Case), attr(AttrName, Arg, AttrCase), IndexName,
    btree).

indexselect(arg(N), pr(Y = attr(AttrName, Arg, AttrCase), _)) =>
  rename(exactmatch(IndexName, rel(Name, Var, Case), Y), Var)
  :-
  argument(N, rel(Name, Var, Case)), Var \= * ,
  hasIndex(rel(Name,Var,Case), attr(AttrName, Arg, AttrCase), IndexName,
    btree).
```

See translations:

```
2 ?- assert(isStarQuery), example5.

Yes
3 ?- writePlanEdges.
Source: 0
Target: 1
Plan  : Staedte  feed {s} plz  feed {p} symmjoin[(.SName_s = ..Ort_p)]
Result: 1

Source: 0
Target: 1
Plan  : Staedte  feed {s}  loopjoin[plz_Ort plz  exactmatch[.SName_s] {p} ]
Result: 1

Source: 0
Target: 1
Plan  : Staedte  feed {s} plz  feed {p} sortmergejoin[SName_s, Ort_p]
Result: 1

Source: 0
Target: 1
Plan  : Staedte  feed {s} plz  feed {p} hashjoin[SName_s, Ort_p, 99997]
Result: 1
```

```
Source: 0
Target: 2
Plan  : plz  feed {p}  filter[(.PLZ_p > 40000)]
Result: 2

Source: 1
Target: 3
Plan  : res(1)  filter[(.PLZ_p > 40000)]
Result: 3

Source: 2
Target: 3
Plan  : Staedte  feed {s} res(2) symmjoin[(.SName_s = ..Ort_p)]
Result: 3

Source: 2
Target: 3
Plan  : Staedte  feed {s} res(2) sortmergejoin[SName_s, Ort_p]
Result: 3

Source: 2
Target: 3
Plan  : Staedte  feed {s} res(2) hashjoin[SName_s, Ort_p, 99997]
Result: 3
Yes
4 ?-
```

Writing Cost Functions

First assign sizes to the nodes and selectivities to the edges of the predicate order graph:

```
7 ?- assignSizes.

Yes
8 ?- writeSizes.

 'Node'   'Size'
------------------
 1        7015.38
 2        22098.5
 3        3756.74

 'Edge'   'Selectivity'   'Predicate'
--------------------------------------
 0-1      0.00293103      attr(s:sName, 1, u)=attr(p:ort, 2, u)
 2-3      0.00293103      attr(s:sName, 1, u)=attr(p:ort, 2, u)
 0-2      0.5355          attr(p:pLZ, 1, u)>40000
 1-3      0.5355          attr(p:pLZ, 1, u)>40000

Yes
9 ?-
```

Create cost edges:

```
14 ?- createCostEdges.

Yes
15 ?- writeCostEdges.

Source: 0
Target: 1
Plan  : Staedte  feed {s} plz  feed {p} symmjoin[(.SName_s = ..Ort_p)]
Result: 1
Size  : 7015.38
Cost  : 3.37645e+006

Source: 0
Target: 1
Plan  : Staedte  feed {s}  loopjoin[plz_Ort plz  exactmatch[.SName_s] {p} ]
Result: 1
Size  : 7015.38
Cost  : 70942.3

Source: 0
Target: 1
Plan  : Staedte  feed {s} plz  feed {p} sortmergejoin[SName_s, Ort_p]
Result: 1
Size  : 7015.38
```

```
Cost  : 157428

Source: 0
Target: 1
Plan  : Staedte  feed {s} plz  feed {p} hashjoin[SName_s, Ort_p, 99997]
Result: 1
Size  : 7015.38
Cost  : 237241

Source: 0
Target: 2
Plan  : plz  feed {p}  filter[(.PLZ_p > 40000)]
Result: 2
Size  : 22098.5
Cost  : 89962.1

Source: 1
Target: 3
Plan  : res(1)  filter[(.PLZ_p > 40000)]
Result: 3
Size  : 3756.74
Cost  : 11785.8

Source: 2
Target: 3
Plan  : Staedte  feed {s} res(2) symmjoin[(.SName_s = ..Ort_p)]
```

```
Result: 3
Size  : 3756.74
Cost  : 1.79706e+006

Source: 2
Target: 3
Plan  : Staedte  feed {s} res(2) sortmergejoin[SName_s, Ort_p]
Result: 3
Size  : 3756.74
Cost  : 69159.7

Source: 2
Target: 3
Plan  : Staedte  feed {s} res(2) hashjoin[SName_s, Ort_p, 99997]
Result: 3
Size  : 3756.74
Cost  : 185720

Yes
16 ?-
```

The Cost Function

```
cost(Term, Sel, Size, Cost) :-

The cost of an executable Term representing a predicate with selectivity Sel is
Cost and the size of the result is Size. Here Term and Sel have to be instanti-
ated, and Size and Cost are returned.
```

Terms look like this:

```
17 ?- planEdge(Source, Target, Term, Result).
```

One of the solutions listed (for `example5`) is

```
Source = 2
Target = 3
Term = symmjoin(rename(feed(rel(staedte, s, u)), s), res(2), attr(s:sName, 1,
u)=attr(p:ort, 2, u))
Result = 3
```

Some cost predicates:

```
cost(rel(Rel, _, _), _, Size, 0) :-
  card(Rel, Size).

cost(res(N), _, Size, 0) :-
  resultSize(N, Size).



cost(feed(X), Sel, S, C) :-
  cost(X, Sel, S, C1),
  feedTC(A),
  C is C1 + A * S.



cost(filter(X, _), Sel, S, C) :-
  cost(X, 1, SizeX, CostX),
  filterTC(A),
  S is SizeX * Sel,
  C is CostX + A * SizeX.
```

```
cost(symmjoin(X, Y, _), Sel, S, C) :-
  cost(X, 1, SizeX, CostX),
  cost(Y, 1, SizeY, CostY),
  symmjoinTC(A, B),              % fetch relative costs
  S is SizeX * SizeY * Sel,      % calculate size of result
  C is CostX + CostY +           % cost to produce the arguments
    A * (SizeX * SizeY) +        % cost to handle buffers and collision
    B * S.                       % cost to produce result tuples
```