

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.
 - [Define type mapping predicates for new operators.]
2. New query processing operators in the relational algebra:
 - Define the operator syntax to be used in SECONDO.
 - [Define type mapping predicates for new operators.]
 - 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.
- [Define type mapping predicates for new operators.]
- Declare a new physical index type.
- Declare according logical index types.
- 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
- *[Define type mapping predicates for new operators.]*
- *Declare a new physical index type.*
- *Declare according logical index types.*

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 (deprecated)
 - 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(groupby(Stream, GroupAttrs, Fields), Result) :-  
    plan_to_atom(Stream, SAtom),  
    plan_to_atom(GroupAttrs, GAtom),  
    plan_to_atom(Fields, FAtom),  
    concat_atom([SAtom, ' groupby[' , GAtom, '; ' , FAtom, ' ]'], '', Result),  
    !.
```

```
plan_to_atom(field(NewAttr, Expr), Result) :-  
    plan_to_atom(attrname(NewAttr), NAtom),  
    plan_to_atom(Expr, EAtom),  
    concat_atom([NAtom, ': ' , EAtom], '', Result),  
    !.
```

Example query:

```
query plz feed sortby[Ort asc]  
    groupby[Ort; Cnt: group count,  
        MinPLZ: group feed min[PLZ],  
        MaxPLZ: group feed max[PLZ] ]  
consume
```

Writing Optimization Rules

See what the optimizer does. Consider the query

```
select count(*)  
from [orte as o, plz as p]  
where [o:bevt < 500, o:ort = p:ort].
```

See the predicate order graph:

```
3 ?- writeNodes.
```

```
Node: 0
```

```
Preds: []
```

```
Partition: [arp(arg(2), [rel(plz, p)], []), arp(arg(1), [rel(orte, o)], [])]
```

```
Node: 1
```

```
Preds: [pr(attr(o:bevT, 1, u)<500, rel(orte, o))]
```

```
Partition: [arp(res(1), [rel(orte, o)], [attr(o:bevT, 1, u)<500]), arp(arg(2),  
[rel(plz, p)], [])]
```

```
Node: 2
```

```
Preds: [pr(attr(o:ort, 1, u)=attr(p:ort, 2, u), rel(orte, o), rel(plz, p))]
```

```
Partition: [arp(res(2), [rel(orte, o), rel(plz, p)], [attr(o:ort, 1,  
u)=attr(p:ort, 2, u)])]
```

```
Node: 3
```

```
Preds: [pr(attr(o:ort, 1, u)=attr(p:ort, 2, u), rel(orte, o), rel(plz, p)),  
pr(attr(o:bevT, 1, u)<500, rel(orte, o))]
```

```
Partition: [arp(res(3), [rel(orte, o), rel(plz, p)], [attr(o:ort, 1,  
u)=attr(p:ort, 2, u), attr(o:bevT, 1, u)<500])]
```

```
true.
```

```
4 ?-
```

4 ?- writeEdges.

Source: 0

Target: 1

Term: arg(1)select pr(attr(o:bevT, 1, u)<500, rel(orte, o))

Result: 1

Source: 0

Target: 2

Term: join(arg(1), arg(2), pr(attr(o:ort, 1, u)=attr(p:ort, 2, u), rel(orte, o),
rel(plz, p)))

Result: 2

Source: 1

Target: 3

Term: join(res(1), arg(2), pr(attr(o:ort, 1, u)=attr(p:ort, 2, u), rel(orte, o),
rel(plz, p)))

Result: 3

Source: 2

Target: 3

Term: res(2)select pr(attr(o:bevT, 1, u)<500, rel(orte, o))

Result: 3

true.

5 ?-

Translating the Arguments

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

```
arg(N) => feedproject(rel(Name, *), AttrNames) :-  
    argument(N, rel(Name, *)), !,  
    usedAttrList(rel(Name, *), AttrNames).
```

```
arg(N) => rename(feedproject(rel(Name, Var), AttrNames), Var) :-  
    argument(N, rel(Name, Var)), !,  
    usedAttrList(rel(Name, Var), AttrNames).
```

See what happens:

```
10 ?- arg(1) => X.  
X = rename(feedproject(rel(orte, o), [attrname(attr(bevT, 0, u)), attrname(attr(  
ort, 0, u))])), o).
```

```
11 ?- -
```

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.
```

```
% replace (Attr = Term) by (Term = Attr)  
indexselect(arg(N), pr(attr(AttrName, Arg, Case) = Y, Rel)) => X :-  
    not(isSubquery(Y)),  
    indexselect(arg(N), pr(Y = attr(AttrName, Arg, Case), Rel)) => X.
```

```

% generic rule for (Term = Attr): exactmatch using btree or hashtable
% without rename
indexselect(arg(N), pr(Y = attr(AttrName, Arg, AttrCase), _)) =>
    exactmatch(dboject(IndexName), rel(Name, *), Y)
:-
    argument(N, rel(Name, *)),
    hasIndex(rel(Name, *), attr(AttrName, Arg, AttrCase), DCindex, IndexType),
    dcName2externalName(DCindex, IndexName),
    (IndexType = btree; IndexType = hash).

% generic rule for (Term = Attr): exactmatch using btree or hashtable
% with rename
indexselect(arg(N), pr(Y = attr(AttrName, Arg, AttrCase), _)) =>
    rename(exactmatch(dboject(IndexName), rel(Name, Var), Y), Var)
:-
    argument(N, rel(Name, Var)), Var \= * ,
    hasIndex(rel(Name, Var), attr(AttrName, Arg, AttrCase), DCindex, IndexType),
    dcName2externalName(DCindex, IndexName),
    (IndexType = btree; IndexType = hash).

```

See translations:

```
5 ?- writePlanEdges.
```

```
Source: 0
```

```
Target: 1
```

```
Plan  : Orte feedproject[BevT, Ort] {o} filter[(.BevT_o < 500)]
```

```
Result: 1
```

```
Source: 0
```

```
Target: 2
```

```
Plan  : Orte feedproject[BevT, Ort] {o} plz feedproject[Ort] {p} symmjoin[(.Ort_o  
= ..Ort_p)]
```

```
Result: 2
```

```
Source: 0
```

```
Target: 2
```

```
Plan  : Orte feedproject[BevT, Ort] {o} loopjoin[plz_Ort_btree plz exact-  
match[.Ort_o] project[Ort] {p} ]
```

```
Result: 2
```

```
Source: 0
```

```
Target: 2
```

```
Plan  : Orte feedproject[BevT, Ort] {o} plz feedproject[Ort] {p} sortmerge-  
join[Ort_o , Ort_p]
```

```
Result: 2
```


Source: 0
Target: 2
Plan : Orte feedproject[BevT, Ort] {o} plz feedproject[Ort] {p} hashjoin[Ort_o
, Ort_p , 99997]
Result: 2

Source: 0
Target: 2
Plan : plz feedproject[Ort] {p} Orte feedproject[BevT, Ort] {o} hashjoin[Ort_p
, Ort_o , 99997]
Result: 2

Source: 1
Target: 3
Plan : res(1) plz feedproject[Ort] {p} symmjoin[(.Ort_o = ..Ort_p)]
Result: 3

Source: 1
Target: 3
Plan : res(1) loopjoin[plz_Ort_btree plz exactmatch[.Ort_o] project[Ort] {p}]
Result: 3

Source: 1
Target: 3
Plan : res(1) plz feedproject[Ort] {p} sortmergejoin[Ort_o , Ort_p]
Result: 3

Source: 1
Target: 3
Plan : res(1) plz feedproject[Ort] {p} hashjoin[Ort_o , Ort_p , 99997]
Result: 3

Source: 1
Target: 3
Plan : plz feedproject[Ort] {p} res(1) hashjoin[Ort_p , Ort_o , 99997]
Result: 3

Source: 2
Target: 3
Plan : res(2) filter[(.BevT_o < 500)]
Result: 3

true.

6 ?-

Writing Cost Functions

Show sizes of nodes and selectivities of edges.

```
6 ?- writeSizes.
```

'Node'	'Size'
1	493.0
2	11359.3
3	11067.5

'Edge'	'Selectivity'	'Predicate'
0-2	0.000544	attr(o:ort, 1, u)=attr(p:ort, 2, u)
1-3	0.000544	attr(o:ort, 1, u)=attr(p:ort, 2, u)
0-1	0.974308	attr(o:bevT, 1, u)<500
2-3	0.974308	attr(o:bevT, 1, u)<500

```
true.
```

```
7 ?-
```

Show cost edges:

```
17 ?- writeCostEdges.
```

```
Source: 0
```

```
Target: 1
```

```
Plan  : Orte feedproject[BevT, Ort] {o} filter[(.BevT_o < 500)]
```

```
Result: 1
```

```
Size  : 493.0
```

```
Cost  : 1244.08
```

```
Source: 0
```

```
Target: 2
```

```
Plan  : Orte feedproject[BevT, Ort] {o} plz feedproject[Ort] {p} symmjoin[(.Ort_o  
= ..Ort_p)]
```

```
Result: 2
```

```
Size  : 11359.3
```

```
Cost  : 226223.0
```

```
Source: 0
```

```
Target: 2
```

```
Plan  : Orte feedproject[BevT, Ort] {o} loopjoin[plz_Ort_btree plz exact-  
match[.Ort_o] project[Ort] {p} ]
```

```
Result: 2
```

```
Size  : 11359.3
```

```
Cost  : 116624.0
```

Source: 0
Target: 2
Plan : Orte feedproject[BevT, Ort] {o} plz feedproject[Ort] {p} sortmerge-
join[Ort_o , Ort_p]
Result: 2
Size : 11359.3
Cost : 161698.0

Source: 0
Target: 2
Plan : Orte feedproject[BevT, Ort] {o} plz feedproject[Ort] {p} hashjoin[Ort_o
, Ort_p , 99997]
Result: 2
Size : 11359.3
Cost : 241238.0

Source: 0
Target: 2
Plan : plz feedproject[Ort] {p} Orte feedproject[BevT, Ort] {o} hashjoin[Ort_p
, Ort_o , 99997]
Result: 2
Size : 11359.3
Cost : 241238.0

Source: 1
Target: 3
Plan : res(1) plz feedproject[Ort] {p} symmjoin[(.Ort_o = ..Ort_p)]
Result: 3
Size : 11067.5
Cost : 220694.0

Source: 1
Target: 3
Plan : res(1) loopjoin[plz_Ort_btree plz exactmatch[.Ort_o] project[Ort] {p}]
Result: 3
Size : 11067.5
Cost : 113381.0

Source: 1
Target: 3
Plan : res(1) plz feedproject[Ort] {p} sortmergejoin[Ort_o , Ort_p]
Result: 3
Size : 11067.5
Cost : 161204.0

Source: 1
Target: 3
Plan : res(1) plz feedproject[Ort] {p} hashjoin[Ort_o , Ort_p , 99997]
Result: 3
Size : 11067.5
Cost : 240768.0

Source: 1
Target: 3
Plan : plz feedproject[Ort] {p} res(1) hashjoin[Ort_p , Ort_o , 99997]
Result: 3
Size : 11067.5
Cost : 240768.0

Source: 2
Target: 3
Plan : res(2) filter[(.BevT_o < 500)]
Result: 3
Size : 11067.5
Cost : 22249.0

true.

8 ?-

The Cost Function

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

The cost of an executable *Term* representing a predicate *Pred* with selectivity *Sel* is *Cost* and the size of the result is *Size*. Here *Term*, *Sel* and *Pred* have to be instantiated, and *Size* and *Cost* are returned.

Terms look like this:

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

One of the solutions listed is

```
Source = 1,  
Target = 3,  
Term = symmjoin(res(1), rename(feedproject(rel(plz, p), [attrname(attr(ort, 0,  
u))]), p), attr(o:ort, 1, u)=attr(p:ort, 2, u)),  
Result = 3 ;
```


Some cost predicates:

```
cost(rel(Rel, X1_), X2_, Pred_, Size, 0) :-  
    card(Rel, Size).
```

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

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

```
cost(filter(X, _), Sel, P, S, C) :-  
    cost(X, 1, P, SizeX, CostX),  
    getPET(P, _, ExpPET),           % fetch stored predicate evaluation time  
    filterTC(A),  
    S is SizeX * Sel,  
    C is CostX + SizeX * (A + ExpPET).  
%C is CostX.
```

```

cost(symmjoin(X, Y, _), Sel, P, S, C) :-
    cost(X, 1, P, SizeX, CostX),
    cost(Y, 1, P, SizeY, CostY),
    getPET(P, _, ExpPET),           % fetch stored predicate evaluation time
    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 + ExpPET) * (SizeX * SizeY) + % cost to handle buffers and collision
        B * S.                    % cost to produce result tuples

```