Basic Optimizer Programmer's Guide

1 Extending the Optimizer

In this section we describe how simple extensions to the optimizer can be done. We first give an overview of how the optimizer works in Section 1.1. We then explain in Section 1.2 how various extensions of the underlying SECONDO system lead to extensions of the optimizer, and how these can be programmed.

The optimizer is written in PROLOG. For programming extensions, some basic knowledge of PROLOG is required, but also sufficient.

1.1 How the Optimizer Works

1.1.1 Overview

The current version of the optimizer is capable of handling *conjunctive queries*, formulated in a relational environment. That is, it takes a set of relations together with a set of selection or join predicates over these relations and produces a query plan that can be executed by (the current relational system implemented in) SECONDO.

The selection of the query plan is based on cost estimates which in turn are based on given selectivities of predicates. Selectivities of predicates are maintained in a table (a set of PROLOG facts). If the selectivity of a predicate is not available from that table, then an interaction with the SECONDO system takes place to determine the selectivity. More specifically, the selectivity is determined by sending a selection or join query on small *samples* of the involved relations to SECONDO which returns the cardinality of the result.

The optimizer also implements a simple SQL-like language for entering queries. The notation is pretty much like SQL except that the lists occurring (lists of attributes, relations, predicates) are written in PROLOG notation. Also note that the where-clause is a list of predicates rather than an arbitrary boolean expression and hence allows one to formulate conjunctive queries only.

Observe that in contrast to the rest of SECONDO, the optimizer is not data model independent. In particular, the queries that can be formulated in the SQL-like language are limited by the structure of SQL and also the fact that we assume a relational model. On the other hand, the core capability of the optimizer to derive efficient plans for conjunctive queries is needed in any kind of data model.

The optimizer in its up-to-date version (the one running in SECONDO) is described completely in [Güt02], the document containing the source code of the optimizer. That document is available from the SECONDO system by changing (in a shell) to the `Optimizer` directory and saying

```
make
pdview optimizer
```

If any questions remain open in the sequel, refer to that document. A somewhat detailed description of optimization in SECONDO can also be found in [GBA+04].

1.1.2 Optimization Algorithm

The optimizer employs an as far as we know novel optimization algorithm which is based on *shortest path search in a predicate order graph*. This technique is remarkably simple to implement, yet efficient.

A predicate order graph (POG) is the graph whose nodes represent sets of evaluated predicates and whose edges represent predicates, containing all possible orders of predicates. Such a graph for three predicates *p*, *q*, and *r* is shown in Figure 1.

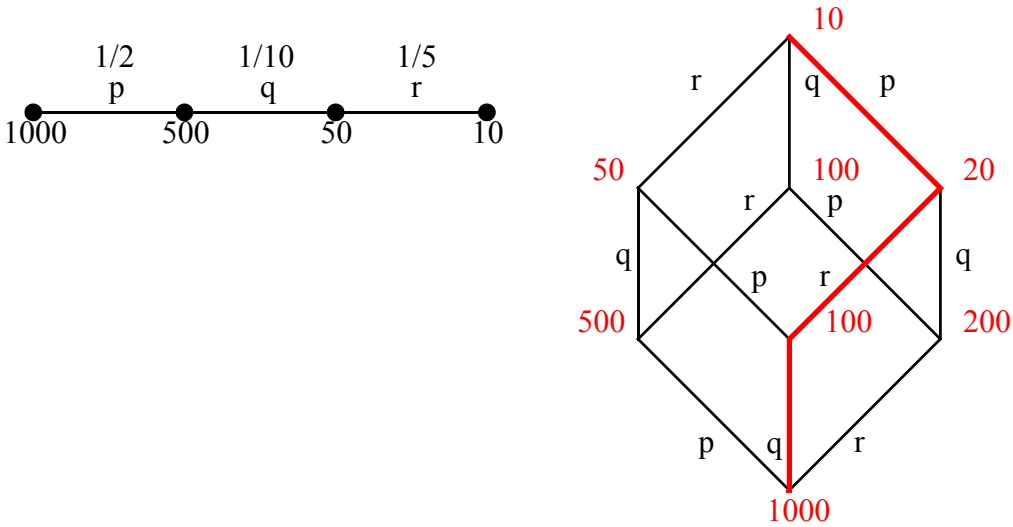

Figure 1: A predicate order graph for 3 predicates

Here the bottom node has no predicate evaluated and the top node has all predicates evaluated. The example illustrates, more precisely, possible sequences of selections on an argument relation of size 1000. If selectivities of predicates are given (for *p* it is 1/2, for *q* 1/10, and for *r* 1/5), then we can annotate the POG with sizes of intermediate results as shown, assuming that all predicates are independent (not *correlated*). This means that the selectivity of a predicate is the same regardless of the order of evaluation, which of course is not always true.

If we can further compute for each edge of the POG possible evaluation methods, adding a new "executable" edge for each method, and mark the edge with estimated costs for this method, then finding a shortest path through the POG corresponds to finding the cheapest query plan. Figure 2 shows an example of a POG annotated with evaluation methods.

In this example, there is only a single method associated with each edge. In general, however, there will be several methods. The example represents the query:

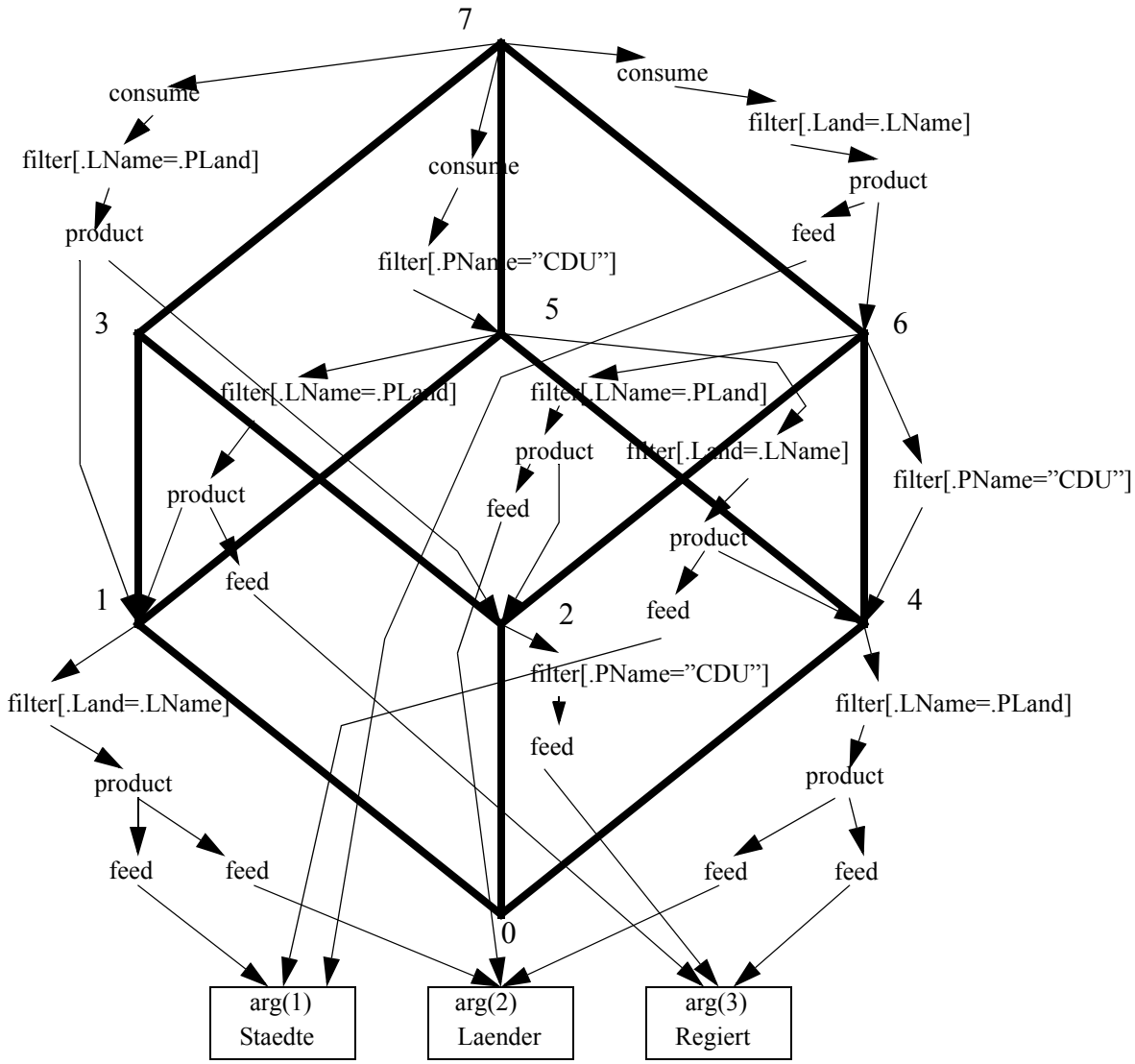

Figure 2: A POG annotated with evaluation methods

```
select *
from Staedte, Laender, Regiert
where Land = LName and PName = 'CDU' and LName = PLand
```

for relation schemas

```
Staedte(SName, Bev, Land)
Laender(LName, LBev)
Regiert(PName, PLand)
```

Hence the optimization algorithm proceeds in the following steps (the section numbers indicated refer to [Güt02]):

1. For given relations and predicates, construct the predicate order graph and store it as a set of facts in memory (Sections 2 through 4).

2. For each edge, construct corresponding executable edges, called *plan edges*. This is controlled by optimization rules describing how selections or joins can be translated (Sections 5 and 6).
3. Based on sizes of arguments and selectivities, compute the sizes of all intermediate results. Also annotate edges of the POG with selectivities (Section 7).
4. For each plan edge, compute its cost and store it in memory (as a set of facts). This is based on sizes of arguments and the selectivity associated with the edge and on a cost function (predicate) written for each operator that may occur in a query plan (Section 8).
5. The algorithm for finding shortest paths by Dijkstra is employed to find a shortest path through the graph of plan edges annotated with costs, called *cost edges*. This path is transformed into a SECONDO query plan and returned (Section 9).
6. Finally, a simple subset of SQL in a PROLOG notation is implemented. So it is possible to enter queries in this language. The optimizer determines from it the lists of relations and predicates in the form needed for constructing the POG, and then invokes step 1 (Section 11).

1.2 Programming Extensions

The following kinds of extensions to the optimizer arise when the SECONDO system is extended by new algebras for attribute types, new query processing methods (operators in the relational algebra), or new types of indexes:

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. [1]
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.

One can see that the same issues arise for various kinds of SECONDO extensions. In the following subsections we cover:

1. In the current version of the optimizer this is not yet done for operations on attribute types. All such operations are assumed to have cost 1. Obviously this is a simplification and in particular wrong for data types of variable size, e.g. *region*. It should be changed in the future.

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

1.2.1 Writing a Display Predicate for a Type Constructor

This is a relatively easy extension. Moreover, it is not mandatory. If the optimizer does not know about a type constructor, it displays the value as a nested list (as in the other user interfaces).

In PROLOG, "functions" are implemented as predicates, hence we actually need to write a display predicate. More precisely, for the existing predicate `display` we need to write a new rule. The predicate is

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

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

The predicate is used to display the list coming back from calling SECONDO. For the result of a query, this list has two elements (<type expression>, <value expression>) which are lists themselves, now converted to the PROLOG form. These are used to instantiate the `Type` and `Value` variables. The structure of the `Type` list is used to control the displaying of values in the `Value` list. The rule to display `int` values is very simple:

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

The following is the rule for displaying a relation:

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

It determines the maximal length of attribute names from the list of attributes `Attr` in the type description and then calls `displayTuples` to display the value list.

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

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

This processes the list, calling `displayTuple` for each tuple.

```
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).
```

Finally, `displayTuple` for each attribute writes the attribute name and calls `display` again for writing the attribute value, controlled by the type of that attribute.

For example, there is no rule to display the spatial type `point`, so let us add one. The list representation of a point value is `[<x-coord>, <y-coord>]`. We want to display a list `[17.5, 20.0]` as

```
[point x = 17.5, y = 20.0]
```

Hence we write a rule:

```
display(point, [X, Y]) :-
  !,
  write('[point x = '),
  write(X),
  write(', y = '),
  write(Y),
  write(']').
```

The predicate `display` is defined in the file `auxiliary.pl`. There, we insert the new rule at an appropriate place before the last rule which captures the case that no other matching rule can be found.

1.2.2 Defining Operator Syntax for SQL

The SECONDO operators that can be used directly within the SQL language are those working on attribute types such as +, <, `mod`, `inside`, `starts`, `distance`, ... In order to not get confused we require that such operators are written with the same syntax in SQL and SECONDO. In this subsection we discuss what needs to be done so that the operator syntax is acceptable to PROLOG within the SQL (term) notation. We also need to tell the optimizer, how to translate such an operator to SECONDO syntax. This is covered in the next subsection.

Some of these operators, for example, +, -, *, /, <, >, are also in PROLOG defined as operators with a specific syntax, so you can write them in this syntax within a PROLOG term without further specification. The operators above are all written in infix syntax; so this is possible also within an SQL where-clause.

For operators that are not yet defined in PROLOG, there are two cases:

- Any operator can be written in prefix syntax, for example

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

This is just the standard PROLOG term notation, so it is fine with PROLOG to write such terms.

- If an operator is to be used in infix syntax, we have to tell PROLOG about it by adding an entry to the file `opsyntax.pl`. For example, to tell that `touches` is an infix operator, we write:

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

  The other arguments besides `touches` determine operator priority and syntax as well as associativity. For our use, please leave the other arguments unchanged.

1.2.3 Defining Operator Syntax for SECONDO

Within the optimizer, query language expressions (terms) are written in prefix notation, as usual in PROLOG. This happens, for example, in optimization rules. Hence, instead of the SECONDO notation

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

a term of the form

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

is manipulated. For any operator, the optimizer must know how to translate it into SECONDO notation. This holds for query processing operators (`feed`, `filter`, ...) not visible at the SQL level as well as for operators on attribute types such as <, `touches`, `length`, `distance`. There are three different ways how operator syntax can be defined, at increasing levels of complexity.:

(1) By *default*. For operators with 1, 2, or 3 arguments that are not treated explicitly, there is a default syntax, namely:

- 1 or 3 arguments: prefix syntax
- 2 arguments: infix syntax

This means, the operators `length`, `touches`, and `translate` with 1, 2, and 3 arguments, respectively, are automatically written as:

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

(2) By a *syntax specification* via predicate `secondoOp` in the file `opsyntax.pl`. This predicate is defined as:

```
secondoOp(Op, Syntax, NoArgs) :-
```

`Op` is a SECONDO operator written in `Syntax`, with `NoArgs` arguments.

Here are some example specifications:

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

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

```
secondoOp(count, postfix, 1).
secondoOp(product, postfix, 2).
secondoOp(filter, postfixbrackets, 2).
secondoOp(loopjoin, postfixbrackets, 2).
secondoOp(exactmatch, postfixbrackets, 3).
```

Not all possible cases are implemented. What can be used currently, is:

- `postfix`, 1 or 2 arguments: corresponds to _ # and _ _ #
- `postfixbrackets`, 2 or 3 arguments, of which the last one is put into the brackets: corresponds to patterns _ # [_] or _ _ # [_]
- `prefix`, 2 arguments: # (_, _)

Observe that `prefix`, either 1 or 3 arguments, and `infix`, 2 arguments, do not need a specification, as they are covered by the default rules mentioned above.

(3) For all other forms, a `plan_to_atom` rule has to be *programmed explicitly*. Such translation rules can be found in Section 5.1.3 of the optimizer source code [Güt02]. The predicate is defined as

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

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

Although not necessary, we could write an explicit rule to translate the product operator:

```
plan_to_atom(product(X, Y), Result) :-
  plan_to_atom(X, XAtom),
  plan_to_atom(Y, YAtom),
  concat_atom([XAtom, YAtom, 'product '], '', Result),
  !.
```

Actually, these rules are not hard to understand: the general idea is to recursively translate the arguments by calls to `plan_to_atom` and then to concatenate the resulting strings together with the operator and any needed parentheses or brackets in the right order into the result string.

An example, where an explicit rule is needed, is the translation of the `sortmergejoin`, which has 4 arguments:

```
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),
  !.
```

In general, very little needs to be done for user level operators as they are mostly covered by defaults, and the specification of query processing operators is also in most cases quite simple via the `secondoOp` predicate.

1.2.4 Writing Optimization Rules

Optimization rules are used to translate selection or join predicates associated with edges of the predicate order graph into corresponding SECONDO expressions. This happens in step 2 of the optimization algorithm described in Section 1.1.2. Before we can formulate translation rules, we need to understand the in detail the representation of predicates.

Consider the query

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

on the optimizer example database `opt` discussed also in the SECONDO User Manual. The optimizer source code [Güt02] contains a rule:

```
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))],
  _, _).
```

This rule corresponds to the query; it says that in order to fulfill the goal `example5`, the predicate order graph should be constructed via calling predicate `pog`. That predicate has four arguments of which only the first two are of interest now. The first argument is a list of relations, the second a list of predicates. A relation is represented as a term, for example,

```
rel(staedte, s, u)
```

which says that the relation name is `staedte`, it has an associated variable `s`, and should in SECONDO be written in upper case (`u`), hence as `Staedte`. A predicate is represented as a term

```
pr(Pred, Rel)
pr(Pred, Rel1, Rel2)
```

of which the first is a selection and the second a join predicate. Hence

```
pr(attr(s:sName, 1, u) = attr(p:ort, 2, u), rel(staedte, s, u),
  rel(plz, p, l))
```

is a join predicate on the two relations `Staedte` and `plz`. An attribute of a relation is represented as a term, for example,

```
attr(s:sName, 1, u)
```

which says that the attribute name is `sName`, it is an attribute of the first of the two relations mentioned in the predicate (`1`), and it should in SECONDO be written in upper case (`u`), hence as `SName`.

Therefore, when you type (after starting the optimizer and opening database `opt`)

```
example5.
```

the predicate order graph for our example query will be constructed. PROLOG replies just `yes`. You can look at the predicate order graph by writing `writeNodes` and `writeEdges`, which lists the nodes and edges of the constructed graph, as follows:

```
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 ?-
```

The information about a node contains the node number which encodes in a way explained in [Güt02] which predicates have already been evaluated; it also contains explicitly the list of predicates that have been evaluated, and some more technical information (`Partition`) describing which of the relations involved have already been connected by join predicates. You can observe that in node 0 no predicate has been evaluated and in node 3 both predicates have been evaluated.

```
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 ?-
```

The information about edges contains the numbers of the source and target node of the POG, and the selection or join predicate associated with this edge. The `Result` field has the number of the node to which the result of evaluating the selection or join is associated; this is normally, but not always (see [Güt02]) the same as the target node of the edge.

Note that the construction of the predicate order graph does not at all depend on the representation of relations or attributes; it does only need to know by a representation `pr(x, a, b)` that this is a join predicate on relations `a` and `b`. For example, you can type

```
pog([a, b, c, d], [pr(x, a), pr(y, b), pr(z, a, b), pr(w, b, c), pr(v, c,
d)], _, _).
```

and the system will construct a POG for the given four relations with two selection and three join predicates. Try it!

After the somewhat lengthy introduction to this subsection we know what the predicates look like that should be transformed by optimization rules into query plans. For example, they can be

```
select(arg(2), pr(attr(p:pLZ, 1, u)>40000, rel(plz, p, l)))

join(arg(1), res(2), pr(attr(s:sName, 1, u)=attr(p:ort, 2, u),
  rel(staedte, s, u), rel(plz, p, l)))
```

In these terms, `arg(N)` refers to the argument number `N` in the construction of the POG, hence one of the original relations, and `res(M)` refers to the intermediate result associated with the node `M` of the POG. Note that an intermediate result is assumed and required to be a stream of tuples so that optimization rules can use stream operators for evaluating predicates on them.

Optimization rules are given in Section 5.2 of the optimizer [Güt02]. Let us consider some example rules.

Translating the Arguments

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

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

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

These rules describe how the arguments of a selection or join predicate should be translated. Translation is defined by the predicate `=>` of arity 2 which has been defined as an infix operator in PROLOG. One might also have called the predicate `translate` and then written, for example

```
translate(res(N), res(N)).
```

However, the form with the “arrow” looks more intuitive; the arrow can be read as “translates into”. The rules above say that a term of the form res(N) is not changed by translation. For `arg(N)`, which is a stored relation, we look up its name and then apply a `feed` and possibly an additional `rename` to convert it into a stream of tuples.

PROLOG is a wonderful environment for testing and debugging. Assuming that we have executed `example5`, we can directly see how things translate. For example, consider

```
18 ?- arg(1) => X.

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

No
19 ?-
```

Hence, we can just pass a term as an argument to the `=>` predicate and a variable for the result and see the translation. We can further check how the translated term is converted into SECONDO notation:

```
21 ?- plan_to_atom(rename(feed(rel(staedte, s, u)), s), X).

X = 'Staedte  feed {s} '

Yes
22 ?-
```

Translating Selection Predicates

Here is a rule to translate a selection predicate:

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

It says that a selection on argument `Arg` can be translated into filtering the stream `ArgS` if `Arg` translates into `ArgS`. A selection can also be translated into an `exactmatch` operation on a B-tree under certain conditions. This is specified by the following three rules:

```
select(arg(N), Y) => X :-
  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).
```

The first rule says that translation of a selection on a stored relation can be reduced to a translation of a corresponding index selection (`indexselect(Arg, Pred)` is just a new term introduced by this

rule). The second rule reverses the order of arguments for an = predicate in order to find the value for searching the index always on the left hand side. The third rule is the most interesting one. It says that index selection with an equality predicate on a stored relation `arg(N)` can be translated into an `exactmatch` operation after looking up the relation and checking that is has an index called `IndexName` on the relevant attribute. At the moment it is still implicit that this index must be a B-tree.

Translating Join Predicates

Finally, let us consider some rules for translating join predicates.

```
join(Arg1, Arg2, pr(Pred, _, _)) => filter(product(Arg1S, Arg2S), Pred) :-
  Arg1 => Arg1S,
  Arg2 => Arg2S.
```

This means that every join can be translated into filtering the Cartesian product of the streams corresponding to the arguments.

```
join(Arg1, Arg2, pr(X=Y, R1, R2)) => JoinPlan :-
  X = attr(_, _, _),
  Y = attr(_, _, _), !,
  Arg1 => Arg1S,
  Arg2 => Arg2S,
  join00(Arg1S, Arg2S, pr(X=Y, R1, R2)) => JoinPlan.

join00(Arg1S, Arg2S, pr(X = Y, _, _)) => sortmergejoin(Arg1S, Arg2S,
  attrname(Attr1), attrname(Attr2))   :-
  isOfFirst(Attr1, X, Y),
  isOfSecond(Attr2, X, Y).

join00(Arg1S, Arg2S, pr(X = Y, _, _)) => hashjoin(Arg1S, Arg2S,
  attrname(Attr1), attrname(Attr2), 997)   :-
  isOfFirst(Attr1, X, Y),
  isOfSecond(Attr2, X, Y).
```

These rules specify ways for translating an equality predicate. The first rule checks whether on both sides of the = predicate there are just attribute names (rather than more complex expressions).[1] In that case, after translating the arguments into streams, the problem is reduced to translating a corresponding term which has a `join00` functor. The second rule specifies translation of the `join00` term into a `sortmergejoin`, the third into a `hashjoin`, using a fixed number of 997 buckets.

The latter two rules use auxiliary predicates `isOfFirst` and `isOfSecond` to get the name of the attribute (among `X` and `Y`) that refers to the first and the second argument, respectively. Note also that the attribute names passed to `sortmergejoin` or `hashjoin` are given as, for example `attrname(Attr1)` rather than `Attr1` directly. The reason is that a normal attribute name of the form `attr(sName, 1, u)` is converted later into the SECONDO "." notation, hence into `.SName` whereas `attrname(attr(sName, 1, u))` is converted into `SName`.

1. There are other rules corresponding to the first one that allow one to translate to a hash join or a sortmergejoin, even if one or both of the arguments to the equality predicate are expressions. The idea is to first apply an `extend` operation which adds the value of the expression as a new attribute, then performing the join using the new attribute, and finally to remove the added attribute again by applying a `remove` operator.

Using Parameter Functions in Translations

In all the rules we have seen so far, it was possible to use the implicit notation for parameter functions in SECONDO. Recall that the implicit form of a `filter` predicate is written as

```
... filter[.Bev > 500000]
```

whereas the explicit complete form would be

```
... filter[fun(t:TUPLE) attr(t, Bev) > 500000]
```

However, sometimes it is necessary to write the full form in SECONDO. For example, in the `loopjoin` operator's parameter function we may need to refer to an attribute of the outer relation explicitly. A join as in our example query

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

could be formulated as a `loopjoin`:

```
query Staedte feed {s} loopjoin[fun(var1:TUPLE) plz feed {p} fil-
ter[attr(var1, SName_s) = .Ort_p]] consume
```

Although currently there are no translation rules yet in the optimizer using the explicit form, there is support for using it. The PROLOG term representing a parameter function has the form:

```
fun([param(Var1, Type1), ..., param(VarN, TypeN)], Expr)
```

Furthermore, when writing rules involving such functions, variable names need to be generated automatically. There is a predicate available

```
newVariable(Var)
```

which returns on every call a new variable name, namely `var1`, `var2`, `var3`, ... For the type operators such as `TUPLE` that one needs to use in explicit parameter functions, a number of conversions to SECONDO are defined by:

```
type_to_atom(tuple, 'TUPLE').
type_to_atom(tuple2, 'TUPLE2').
type_to_atom(group, 'GROUP').
```

The `attr` operator of SECONDO is available under the name `attribute` (in PROLOG) in order to avoid confusion with the `attr(_, _, _)` notation for attribute names. Of course, it is converted back to `attr` in the `plan_to_atom` rule.

Hence a PROLOG term corresponding to

```
Staedte feed filter[fun(t:TUPLE) attr(t, Bev) > 500000]
```

is

```
filter(feed(rel(staedte, *, u)),
  fun([param(t, tuple)], attribute(t, attrname(attr(bev, 0, u))) > 500000))
```

Showing Translations

One can see the translations that the optimizer generates for all edges of a given POG by the goal `writePlanEdges`. For the `example5` above we get:

```
17 ?- example5.

Yes
18 ?- writePlanEdges.
Source: 0
Target: 1
Plan: Staedte  feed {s}  plz  feed {p}  product  filter[(.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, 997]
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)  product  filter[(.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, 997]
Result: 3

Yes
19 ?-
```

1.2.5 Writing Cost Functions

The next step in the optimization algorithm described in Section 1.1.2, step 3, is to compute the sizes of all intermediate results and associate the selectivities of predicates with all edges. No extensions are needed in this step. We can call for an execution of this step by the goal `assignSizes` (`deleteSizes` to remove them again) and see the result by `writeSizes`. Continuing with `example5`, we have:

```
20 ?- assignSizes.

Yes
21 ?- writeSizes.
Node: 1
Size: 7419.81

Node: 2
Size: 22696.9

Node: 3
Size: 4080.89

Source: 0
Target: 1
Selectivity: 0.0031

Source: 0
Target: 2
Selectivity: 0.55

Source: 1
Target: 3
Selectivity: 0.55

Source: 2
Target: 3
Selectivity: 0.0031

Yes
22 ?-
```

The next step 4 is to assign costs to all generated plan edges. This step can be called explicitly by the goal `createCostEdges` (removal by `deleteCostEdges`), and plan edges annotated with costs can be listed by `writeCostEdges`. For `example5`, we have now:

```
29 ?- createCostEdges.

Yes
30 ?- writeCostEdges.
Source: 0
Target: 1
Plan: Staedte  feed {s}  plz  feed {p}  product  filter[(.SName_s = .Ort_p)]
Result: 1
Size: 7419.81
Cost: 8.23362e+006
```

```
Source: 0
Target: 1
Plan: Staedte  feed {s}  loopjoin[plz_Ort plz  exactmatch[.SName_s] {p} ]
Result: 1
Size: 7419.81
Cost: 75027

Source: 0
Target: 1
Plan: Staedte  feed {s} plz  feed {p} sortmergejoin[SName_s, Ort_p]
Result: 1
Size: 7419.81
Cost: 157724

Source: 0
Target: 1
Plan: Staedte  feed {s} plz  feed {p} hashjoin[SName_s, Ort_p, 997]
Result: 1
Size: 7419.81
Cost: 92569.4

Source: 0
Target: 2
Plan: plz  feed {p}  filter[(.PLZ_p > 40000)]
Result: 2
Size: 22696.9
Cost: 89962.1

Source: 1
Target: 3
Plan: res(1)  filter[(.PLZ_p > 40000)]
Result: 3
Size: 4080.89
Cost: 12465.3

Source: 2
Target: 3
Plan: Staedte  feed {s}  res(2)  product  filter[(.SName_s = .Ort_p)]
Result: 3
Size: 4080.89
Cost: 3.8703e+006

Source: 2
Target: 3
Plan: Staedte  feed {s} res(2) sortmergejoin[SName_s, Ort_p]
Result: 3
Size: 4080.89
Cost: 71374

Source: 2
Target: 3
Plan: Staedte  feed {s} res(2) hashjoin[SName_s, Ort_p, 997]
Result: 3
Size: 4080.89
Cost: 40289.9
```

```
Yes
31 ?-
```

Here we can see that each plan edge has been annotated with the expected size of the result at the result (usually target) node of that edge. This is the same size as in the listing for `writeSizes`; it has just been copied to the plan edges. More important is the computation of the cost for each plan edge, which is listed in the `Cost` field.

The cost for an edge is computed by a predicate `cost` defined in Section 8.1 of the optimizer [Güt02]. The predicate is:

`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 instantiated, and `Size` and `Cost` are returned.

The predicate `cost` is called by the predicate `createCostEdges` which passes to it a term (resulting from translation rules and associated with a plan edge) and the selectivity associated with that edge. The predicate is then evaluated by recursively descending into the term. For each operator applied to some arguments, the cost is determined by first computing the cost of producing the arguments and the size of each argument and then computing the cost and result size for evaluating this operator.

Somewhere in the term is an operator that actually realizes the predicate associated with the edge. For example, this could be the `filter` or the `sortmergejoin` operator. This operator uses the selectivity `Sel` passed to it to determine the size of its result.

We can see the existing plan edges after constructing the POG by writing:

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

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

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

For each operator or argument occurring in a term there must be a rule describing how to get the result size and cost. Let us consider some of these rules.

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

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

These rules determine the size and cost of arguments. In both cases the cost is 0 (there is no computation involved yet) and the size is looked up. For a stored relation it is found via a fact `card(Rel, Size)` stored in the file `database.pl` (see the SECONDO User Manual); for an intermediate result it

was computed by `assignSizes` and can be looked up via predicate `resultSize`. The `Sel` argument passed is not used.

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

This is the rule for the `feed` operator. It first determines size `S` and cost `C1` for the argument `X`. The size of the result is for `feed` the same as the size of the argument (relation). The cost is determined by using a "feed tuple constant" that describes the cost for evaluating feed on one tuple. Such constants are determined experimentally and stored in a file `operators.pl` (see Appendix C of the optimizer [Güt02]). There we find an entry

```
feedTC(0.4).
```

The cost for evaluating feed is therefore `C1` (we know that is 0 by the rule above) plus 0.4 times the number of tuples of the argument relation.

```
cost(rename(X, _), Sel, S, C) :-
  cost(X, Sel, S, C1),
  renameTC(A),
  C is C1 + A * S.
```

The rule for `rename` is quite similar. The operator just passes the tuple that it receives to the next operator; the `rename` tuple constant happens to be 0.1. The cost is added to the cost of the argument.

```
cost(product(X, Y), _, S, C) :-
  cost(X, 1, SizeX, CostX),
  cost(Y, 1, SizeY, CostY),
  productTC(A, B),
  S is SizeX * SizeY,
  C is CostX + CostY * SizeY * B + S * A.
```

The `product` operator first collects its second argument stream `Y` into a buffer. It then processes the first argument stream `X`, combining each of its tuples with each tuple in the buffer. The result size is, as everyone knows, the product of the sizes of the arguments. The cost is the sum of the costs for producing the arguments, reading argument `Y` into the buffer (per tuple cost given by constant `B`), and producing product tuples (per tuple cost `A`).

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

The `filter` operator determines the cost and size for its argument. Note that it passes a selectivity 1 to the argument cost evaluation, because the selectivity is actually "used" by this operator. The result size for `filter` is determined by applying the `Sel` factor.

For the moment cost estimation is still rather simplistic, but one can see the principles. For example, in the `filter` operator we assume a constant cost for evaluating a predicate regardless of what the predicate is. In the rule above the predicate is not considered. A refinement would be to also model

the cost for all operators that can occur in predicates, and then to model the cost for predicate evaluation precisely. In addition one would need for variable size attribute data types statistics about their average size. For example, for a relation with an attribute of type `region`, one should have a predicate (similar to `card`) stating the average number of edges for `region` values in that relation. Alternatively, similar to the current selectivity determination, such statistics could be retrieved by a query to SECONDO and then be stored for further use.

Cost estimation needs to be extended when a new operator is added that is used in translations of predicates. From the algorithm implementing the operator one should understand how the sizes of arguments determine the cost and write a corresponding rule. The relevant factors for the per tuple cost need to be determined in experiments; they should be set relative to the other existing factors in the file `operators.pl`. Of course, the relationship between these factors and the actual running times depend on the machine where the experiments are run.