8 The Optimizer

The optimizer component of SECONDO is written in PROLOG and allows one to formulate SECONDO commands as well as queries in an SQL-like language within a PROLOG environment. Commands are passed directly to the SECONDO kernel for execution. Queries are translated to query plans which are then also sent to the kernel for exection. One can also experiment with the optimizer and just see how queries are translated without executing them.

In the following sections after some preparations we discuss the PROLOG environment, the query language, hybrid queries (combining SQL with SECONDO operations) and creation of objects from query results, the optimizer's knowledge about databases, and how the optimizer can be informed about new operators available in SECONDO.

8.1 Preparations

In the following examples, we work with the database `opt`. Hence, enter at any of the user interfaces (e.g. in `SecondoTTYBDB`) the commands:

```
create database opt
restore database opt from opt
```

Now the database is in good shape. When you type `list objects`, you can see that it has the following relations:[1]

```
Orte(Kennzeichen: string, Ort: string, Vorwahl: string, BevT: int)
Staedte(SName: string, Bev: int, PLZ: int Vorwahl: string,
  Kennzeichen: string)
plz(PLZ: int, Ort: string)
ten(no: int)
thousand(no:int)
```

Furthermore, there are the two indexes `plz_Ort` and `plz_PLZ` which index on the `plz` relation the attributes `Ort` and `PLZ`, respectively. All relations are small except for `plz` which is a bit larger, having 41267 tuples.

8.2 Using SECONDO in a PROLOG environment

In Section 5 it was already discussed how the optimizer can be called. In this section, we assume that the single user version `SecondoPL` is used; the client-server interface `SecondoPLCS` behaves similarly. Hence, switch to the directory `Optimizer` and call the optimizer by the command:

```
SecondoPL
```

After some messages, there appears a PROLOG prompt:

```
1 ?-
```

1. There is also a further relation SEC_DERIVED_OBJ used internally to restore indexes and other derived objects, and some more system tables starting SEC2... System tables can be queried like any relation, but they are not persistent and will get lost between two sessions.

When the optimizer is used for the first time after installing SECONDO, some error messages appear; these can safely be ignored. The reason is that some files generated by the running optimizer are not yet there.

We now have a PROLOG interpreter running which understands an additional predicate:

```
secondo(Command, Result) :- execute the Secondo command Command and get
the result in Result.
```

So at the command line, one can type:

```
1 ?- secondo('open database opt', Res).
```

This is executed, some SECONDO messages appear, and then the PROLOG interpreter shows the result of binding variable `Res`:

```
Res = []
```

This is the empty list that SECONDO returns on executing successfully such a command, converted to a PROLOG list. As usual with a PROLOG interpreter we can type `<return>` to accept the first solution, or `;<return>` to see more solutions (if any exist). After typing `<return>`, the interpreter responds

```
Yes
2 ?-
```

Let us try another command:

```
2 ?- secondo('query Staedte feed filter[.Bev > 500000] head[3]
|    consume', R), R = [First, _].
```

Here at the end of the first line we typed `<return>`, the interpreter then put "`|` " at the beginning of the next line. The PROLOG goal is complete only with the final "." symbol, only then interpretation is started. Here as a result we get after some SECONDO messages the result of the query shown in variable `R` and the first element in variable `First`. This illustrates that, of course, we can process SECONDO results further in the PROLOG environment.

By the way, when you later want to quit the running `SecondoPL` program, just type at the prompt either

```
.. ?- halt.
```

or

```
.. ?- quit.
```

The first is the standard PROLOG termination, the second has been introduced to be consistent with other SECONDO interfaces. In the sequel, we omit the PROLOG prompt in the examples.

There is also a version of the `secondo` predicate that has only one argument:

```
secondo(Command) :- execute the Secondo command Command and pretty-print
the result, if any.
```

Hence we can say:

```
secondo('query Staedte').
```

The result is printed in a similar format as in `SecondoTTYBDB` or `SecondoTTYCS`.

In addition, a number of predicates are available that mimic some frequently used SECONDO commands, namely

```
open
create
update
let
delete
query
```

They all take a character string as a single argument, containing the rest of the command, and are defined in PROLOG to be prefix operators, hence we can write:

```
secondo('close database').
open 'database opt'.
create 'x: int'.
update 'x := Staedte feed count'.
let 'double = fun(n: int) 2 * n'.
query 'double(x)'.
delete 'x'.
```

In the remainder of this chapter, we assume that the "standard" version of the optimizer is active. To ensure this, type

```
setOption(standard).
```

after starting `SecondoPL`. This command will do some output and finally print an overview of all available options. A marked checkbox in front of option "standard" indicates, that it is active.

8.3 An SQL-like Query Language

The optimizer implements a part of an SQL-like language by a predicate `sql`, to be written in prefix notation, and provides some operator definitions and priorities, e.g. for `select`, `from`, `where`, that allow us to write an SQL query directly as a PROLOG term. For example, one can write (assuming database `opt` is open):

```
sql select * from staedte where bev > 500000.
```

Note that in this environment all relation names and attribute names are written in lower case letters only. Remember that words starting with a capital are variables in PROLOG; therefore we cannot use such words. The optimizer on its own gets information from the SECONDO kernel about the spellings of relation and attribute names and sends query plans to SECONDO with the correct spelling.

Some messages appear that tell you something about the inner workings of the optimizer. Possibly the optimizer sends by itself some small queries to SECONDO, then it says:

```
Destination node 1 reached at iteration 1
Height of search tree for boundary is 0

The best plan is:

Staedte feed filter[.Bev > 500000] consume

Estimated Cost: 120.64
```

After that appear evaluation messages and the result of the query. If you are interested in understanding how the optimizer works, please read the paper [GBA+04]. If you wish to understand the working of the optimizer in more detail, you can also read the source code documentation, that is, say in the directory `Optimizer`:

```
pdview optimizer.pl
```

Almost all prolog sourcefiles (having filename extension `.pl`) from the `Optimizer` directory can be processed in this way. In the following, we describe the currently implemented query language in detail. Whereas the syntax resembles SQL, no attempt is made to be consistent with any particular SQL standard.

Basic Queries

The SQL kernel implemented by the optimizer basically has the following syntax:

```
select <attr-list>
from <rel-list>
where <pred-list>
```

Each of the lists has to be written in PROLOG syntax (i.e., in square brackets, entries separated by comma). If any of the lists has only a single element, the square brackets can be omitted. Instead of an attribute list one can also write "`*`". Hence one can write (don't forget to type `sql` before all such queries):

```
select [sname, bev]
from staedte
where [bev > 270000, sname starts "S"]
```

To avoid name conflicts, one can introduce explicit variables. In this case one refers to attributes in the form `<variable>:<attr>`. For example, one can perform a join between relations `Orte` and `plz`:

```
select * from [orte as o, plz as p]
where [o:ort = p:ort, o:ort contains "dorf", (p:plz mod 13) = 0]
```

In the sequel, we define the syntax precisely by giving a grammar. For the basic queries described so far we have the following grammar rules:

```
query           ->  select distinct sel-clause from rel-list where-clause

distinct        ->  distinct | empty

sel-clause      ->  * | count(*) | result-list

result          ->  attr | attr-expr as newname

attr            ->  attrname | var:attrname

rel             ->  relname | relname as var

where-clause    ->  where pred-list
                  | empty

pred            ->  attr-boolexpr
```

We use the following notational conventions. Words written in normal font are grammar symbols (non-terminals), words in bold face are terminal symbols. The symbols "`->`" and "`|`" are meta-symbols denoting derivation in the grammar and separation of alternatives. Other characters like "`*`" or "`:`" are also terminals.

The notation `X-`*`list`* refers to a non-empty PROLOG list with elements of type `X`; as mentioned already, the square brackets can be omitted if the list has just one element.

The notation `X-`*`expr`* refers to an expression built from elements of type `X`, constants, and operations available on `X`-values. Hence `attr-`*`expr`* is an expression involving attributes denoted in one of the two forms `attrname` or `var:attrname`. Similarly a predicate (`pred`) is a boolean expression over attributes.

Finally, *empty* denotes the empty alternative. Hence the where-clause or the distinct keyword are optional.

From the grammar, one can see that it is also possible to compute derived attributes in the select-clause. For example:

```
select [sname, bev div 1000 as bevt] from staedte
```

Order

One can add an orderby-clause (and a first-clause, see below), hence the syntax of a query is more completely:

```
query           ->  select distinct sel-clause from rel-list where-clause
                    orderby-clause first-clause

orderby-clause  ->  orderby orderattr-list
                  | empty

orderattr       ->  attrname | attrname asc | attrname desc
```

For example, we can say:

```
select [o:ort, p1:plz, p2:plz]
from [orte as o, plz as p1, plz as p2]
where [o:ort = p1:ort, p2:plz = (p1:plz + 1), o:ort contains "dorf"]
orderby [o:ort asc, p2:plz desc]
```

It is possible to mention derived attributes in the orderby-clause.

Taking Only the First *n* Elements

Sometimes one is interested in only the first few tuples of a query result. This can be achieved by using a first-clause:

```
first-clause    ->  first int-constant
                  | empty
```

For example:

```
select * from plz orderby ort desc first 3
```

This is also a convenient way to see the beginning of a large relation. Only the first few tuples are processed.

Grouping and Aggregation

Aggregation queries have a groupby-clause in addition to what is known already and a different form of the select-clause.

```
query           ->  select aggr-list from rel-list where-clause
                    groupby-clause orderby-clause first-clause

aggr            ->  groupattr | count(*) as newname
                  | aggrop(attr-expr) as newname

groupattr       ->  attr

aggrop          ->  min | max | sum | avg

groupby-clause  ->  groupby attr-list
```

For example, one can say:

```
select [ort, min(plz) as minplz, max(plz) as maxplz, count(*) as cntplz]
from plz
where plz > 40000
groupby ort
orderby cntplz desc
first 10
```

Entries in the select-clause are either attributes used in the grouping or definitions of derived attributes which are obtained by evaluating aggregate functions on the group. Again one can order by such derived values. An aggregate operator like `sum` cannot only be applied to an attribute name, but also to an expression built over attributes.

There is one restriction imposed by the current implementation and not visible in the grammar: the select-clause in an aggregate query must contain a derived attribute definition. Hence

```
select ort from plz groupby ort
```

will not work. This will be optimized but not executed by SECONDO.

Union and Intersection

It is possible to form the union or intersection of a set of relations each of which is the result of a separate query. The queries are written in a PROLOG list. All result relations must have the same schema.

```
mquery          ->  query
                  | union query-list
                  | intersection query-list
```

For example:

```
union [
  select * from plz where ort contains "dorf",
  select * from plz where ort contains "stadt"]
```

Note that in this case, each of the subqueries in the list is optimized separately. One interesting application is to find tuples in a relation fulfilling a very large set of conditions. The optimizer's effort in optimizing a single query is exponential in the number of predicates. It works fine roughly up to 10 predicates. Beyond that optimization times get long. However, it is no problem to use, for example, an intersection query on 30 subqueries each of which has only one or a few conditions.

The query processed by the optimizer is an `mquery`, i.e., the query command is of the form

```
sql mquery
```

The complete grammar can be found in Appendix B.

8.4 Further Ways of Querying

The basic form of querying is using the `sql` predicate in prefix notation, as explained in the previous section, hence

```
sql Term
```

Hybrid Queries

A second form of the `sql` predicate allows one to further process the result of a query by SECONDO operators:

```
sql(Term, SecondoQueryRest)
```

Here `SecondoQueryRest` contains a character string with SECONDO operators, applicable to a stream of tuples returned by the optimized and evaluated `Term`. For example:

```
sql(select * from orte where bevt > 300, 'project [Ort] consume').
```

Note that in the second argument, attribute names have to be spelled correctly as in writing executable queries to the SECONDO kernel. In this example, the same effect could have been achieved by a pure SQL query, but there are cases when this facility is useful.

Creating Objects

The `let` command of SECONDO allows one to create SECONDO objects as the result of an executable query. There is a `let` predicate in the optimizer that allows one to do the same for the result of an optimized query. There are two forms, the second one corresponding to a hybrid query.

```
let(ObjectName, Term)
let(ObjectName, Term, SecondoQueryRest)
```

For example:

```
let(orte2, select ort from [orte, plz as p] where ort = p:ort orderby ort,
'rdup consume').
```

This query creates a relation `orte2` with the names of places ("Orte") that also occur in the postal code relation `plz`. Here duplicate removal was done at the executable level. However, it can also be done directly, by saying:

```
let(orte3, select distinct ort from [orte, plz as p] where ort = p:ort
orderby ort).
```

Just Optimizing

For experimenting with the optimizer it is useful to optimize queries without executing them. This is provided by the `optimize` predicate.

```
optimize(Term)
```

This returns the query plan and the expected cost.

8.5 The Optimizer's Knowledge of Databases

The optimizer and the SECONDO kernel are only loosely coupled. In particular, one can use the kernel independently, create and delete databases and objects within databases out of control of the optimizer.

The optimizer maintains knowledge about the existing database contents within a number of "dynamic predicates" while the optimizer is running, and in files between sessions. It obtains such knowledge from the SECONDO kernel by sending commands or queries to it, for example, `list objects`. Currently there are the following such predicates and corresponding files:

- *storedRels* - relations and their attributes
- *storedSpells* - spellings of relation and attribute names
- *storedIndexes* - for which attributes do and do not exist indexes
- *storedCards* - cardinalities of relations
- *storedTupleSizes* - average tuple sizes (in bytes) of relations
- *storedSels* - selectivities of selection and join predicates
- *storedPETs* - predicate evaluation times for selection and join predicates
- *storedAttrSizes* - average sizes of attributes in bytes
- *storedTypeSizes* - the size of SECONDO datatypes[2]
- *storedOrderings* - known orderings within stored relations

The optimizer distinguishes between different databases by saving the database name with all facts of its knowledge base.

The general principle is that the optimizer retrieves information from SECONDO when it is needed and then stores it for later use. For example, when a relation is mentioned for the first time in a query, the optimizer sends "`list objects`" to the kernel to check whether the relation exists and to get attribute names with their spelling. It also determines whether there are indexes available and creates a small sample relation if there is none yet. It sends a query "`<relname> count`" to

2. Created by the SECONDO kernel, this refers to the fixed size part of a data type representation.

get the cardinality and another "<relname> tuplesize" to get the average tuple size in byte. It also gets average attribute sizes by queries of the form "<relname> attrsize[<attrname>]".

When in a query a selection or join predicate occurs for which the selectivity is not yet known, the optimizer sends a corresponding query on the small sample relation(s) to determine the selectivity and predicate evaluation time (PET).

Note: The optimizer recognizes indexes by a name convention. The name of the index must have the form <relation name>_<attribute name>. These names must be spelled as in the SECONDO kernel except that the first letter must be in lower case (due to its use in PROLOG). Hence an index on attribute Bev of relation Staedte must be called staedte_Bev to be recognized by the optimizer. Such an index can be created by the command:

```
let 'staedte_Bev = Staedte createbtree[Bev]'.
```

Inquiries on the Optimizer's Knowledge Base

The optimizer provides several commands for inquiries on its knowledge base:

- showStoredRels - lists all known relations and their attributes
- showStoredAttrSizes - lists types and sizes for known attributes
- showStoredOrders - list known orderings
- showSels - lists the known selectivities
- showPETs - lists all known PETs

The command

- showDatabase

will show a summary of all relation-based information gathered for the open database, including information on attribute types and sizes, cardinalities and average tuple sizes, known indexes and orderings. Last, the command

- showDatabaseSchema

will list all relations available in the open database (not only the ones known to the optimizer from past queries) together with their attributes and attribute types.

Reinitializing

One can reinitialize the optimizer's knowledge of databases by deleting the files storedRels etc. mentioned above from the directory Optimizer (when the optimizer is not running). In this case, all information needed will be collected afresh on further queries.

If you somehow run into unexplainable problems with the optimizer, it is usually a good idea to quit the optimizer, delete all these stored*.pl files and restart the optimizer.

Creating and Deleting Relations

When new relation objects are created, the optimizer should recognize them automatically as soon as they are used in a query. However, the optimizer will not automatically be aware that a relation has been deleted and will still create query plans for it which will then be refused by the SECONDO kernel. We explain below how the optimizer can be informed about the deletion.

Creating and Deleting Indexes

The optimizer checks for indexes when a relation is mentioned for the first time in a query. Hence, it automatically recognizes indexes created together with a relation before querying. However, once it has been determined that for a given attribute of a relation no index exists, the optimizer will not check further for an index on that attribute. The optimizer also does not notice when an index is deleted.

Informing the Optimizer

Two commands (predicates) are available to explicitly inform the optimizer about changes to relations and indexes.

```
updateRel(Rel)
```

A call of this predicate causes the optimizer to delete all information it has about the relation `Rel`, including selectivities of predicates. An existing sample is also destroyed. A query afterwards involving this relation collects all information from scratch. Existing or non-existing indexes are also discovered. For example:

```
updateRel(plz).
```

resets all information for relation `plz`. The second predicate is:

```
updateIndex.
```

This predicate lets the optimizer check whether any indexes have been added or removed to update its knowledge base. Hence this can be used after creating or destroying an index, without losing all the other information collected for relation `Rel`. For example, after deleting the index `plz_Ort` one should inform the optimizer by saying `updateIndex` once.

Creating Sample Relations Manually

As mentioned above, the optimizer uses small sample relations to determine selectivities of selection or join predicates before actually optimizing the query. These sample relations are normally created automatically, with default sizes. There is one sample relation called `<relname>_sample_s` and another one called `<relname>_sample_j`, to be used for selection and join predicates, and with default sizes 2000 and 500 tuples, respectively. If a relation has less tuples than that, the sample will be the full relation.

However, in some cases this default value is not appropriate. This is mainly the case when a relation has few tuples containing large objects. For example, in the `germany` example database that

comes with SECONDO, there is a relation `Kreis` with 439 tuples, each containing a *region* value with several hundred or thousand edges. In this case a sample with 10% of the tuples is quite sufficient and preferable, as on the geometries often expensive predicates are evaluated. hence in such a case one would like to have samples of size 50, say, for both selection and joins.

Unfortunately it is difficult to build a general rule for such samples into the optimizer, and this has not been done. Instead, the optimizer complains if by default it would create a sample relation of size more than 2 MB, and asks the user to manually create samples. This can be done as follows. The predicate

```
createSamples(RelName, SizeSel, SizeJoin)
```

creates samples with the desired sizes. Note that here the relation name has to be given in quotes as the real Secondo name is used (rather than the lower case version used in queries). For example:

```
createSamples('Kreis', 50, 50)
```

In some configurations of the optimizer, also so-called *small* relations are created, named `<relname>_small`. The optimizer will create small relations automatically. If the automatically chosen size of a small relation is not suitable, one can create it manually by a predicate

```
createSmall(Rel, Size)
```

In this case, however, the relation name has to be spelled as in queries, that is:

```
createSmall(kreis, 50)
```

8.6 Operator Syntax

In queries given to the optimizer one uses atomic operators in predicates and expressions in the select-clause like

```
<, >, <=, #, starts, contains, +, *, div, mod
```

In this section we explain how new operators of this kind can be made available in the optimizer. For using operators in queries, there are two conditions:

1. We must be able to write the operator in PROLOG.
2. The optimizer must know how to translate the operator application to SECONDO syntax.

PROLOG Syntax

Any operator can be written in PROLOG in prefix syntax. For example:

```
length(X), theDate(2004, 5, 9)
```

These are just standard terms in PROLOG. If we want to write a (binary) operator in infix notation, either this operator is defined already in PROLOG. This is the case for standard operators like `+`, `*`, `<`, etc. Otherwise one can explicitly define it in the file `opsyntax.pl` in directory `Optimizer`. For example, in the file we find definitions:

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

```
:- op(800, xfx, intersects).
:- op(800, xfx, adjacent).
:- op(800, xfx, or).
:- op(800, fx, not).
```

Here `inside`, `intersects`, `adjacent`, and `or` are defined to be binary infix operators, and `not` is defined to be a unary prefix operator. New operators can be made available in the same way.

SECONDO Syntax

Translation to SECONDO is controlled firstly, by a few defaults, depending on the number of arguments:

- one argument: translated to prefix notation
  `op(arg)`
- two arguments: translated to infix notation
  `arg1 op arg2`
- three arguments: translated to prefix notation
  `op(arg1, arg2, arg3)`

If a binary operator is to be translated to prefix notation instead, one can place a fact into the file `opsyntax.pl` of the form

```
secondoOp(Op, prefix, 2)
```

For example, to define a `distance` operator with two arguments to be written in prefix notation we can specify:

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

Reload the file after modifying it:

```
[opsyntax].
```

The current contents of the file are shown in Appendix A. For example, we can now use the `distance` operator in a query (on a database `germany`):

```
select [sname, distance(ort, s2:ort) as dist]
from [stadt, stadt as s2]
where [s2:sname = "Dortmund", distance(ort, s2:ort) < 0.3]
```

A Operator Syntax

```
/*
[File ~opsyntax.pl~]

*/

:-
  op(800, xfx, =>),
  op(800, xfx, <=),
  op(800, xfx, #),
  op(800, xfx, div),
  op(800, xfx, mod),
  op(800, xfx, starts),
  op(800, xfx, contains),
  op(200, xfx, :).

:- op(800, xfx, inside).
:- op(800, xfx, insideold).
:- op(800, xfx, intersects).
:- op(800, xfx, adjacent).
:- op(800, xfx, attached).
:- op(800, xfx, overlaps).
:- op(800, xfx, onborder).
:- op(800, xfx, ininterior).
:- op(800, xfx, touchpoints).
:- op(800, xfx, intersection).
:- op(800, xfx, commonborder).
:- op(800, xfx, commonborderscan).

:- op(800, xfx, or).
:- op(800,  fx, not).

:- op(800, xfx, present).
:- op(800, xfx, passes).
:- op(800, xfx, atinstant).
:- op(800, xfx, atperiods).
:- op(800, xfx, at).

:- op(800, xfx, satisfies).
:- op(800, xfx, when).

:- op(800, xfx, simpleequals).

:- op(800, xfx, intersects_new).
:- op(800, xfx, p_intersects).


/*

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

~Op~ is a Secondo operator written in ~Syntax~, with ~NoArgs~ arguments.
Currently implemented:

  * postfix, 1 or 2 arguments: corresponds to \_ \# and \_ \_ \#

  * postfixbrackets, 2 or 3 arguments, of which the last one is put into
```

```
the brackets: \_ \# [ \_ ] or \_ \_ \# [ \_ ]

  * prefix, 2 arguments: \# (\_, \_)

  * prefix, either 1 or 3 arguments, does not need a rule here, is
translated by default.

  * infix, 2 arguments: does not need a rule, translated by default.

For all other forms, a plan\_to\_atom rule has to be programmed expli-
citly.

*/

secondoOp(distance, prefix, 2).
secondoOp(intersection_new, prefix, 2).
secondoOp(intersection, prefix, 2).
secondoOp(theperiod, prefix, 2).
secondoOp(union_new, prefix, 2).
secondoOp(minus_new, prefix, 2).
secondoOp(feed, postfix, 1).
secondoOp(consume, postfix, 1).
secondoOp(count, postfix, 1).
secondoOp(pdelete, postfix, 1).
secondoOp(product, postfix, 2).
secondoOp(filter, postfixbrackets, 2).
secondoOp(puse, postfixbrackets, 2).
secondoOp(pfeed, postfixbrackets, 2).
secondoOp(pcreate, postfixbrackets, 2).
secondoOp(loopjoin, postfixbrackets, 2).
secondoOp(exactmatch, postfixbrackets, 3).
secondoOp(leftrange, postfixbrackets, 3).
secondoOp(rightrange, postfixbrackets, 3).
secondoOp(remove, postfixbrackets, 2).
secondoOp(sortby, postfixbrackets, 2).
secondoOp(loopsel, postfixbrackets, 2).
secondoOp(sum, postfixbrackets, 2).
secondoOp(min, postfixbrackets, 2).
secondoOp(max, postfixbrackets, 2).
secondoOp(avg, postfixbrackets, 2).
secondoOp(tuplesize, postfix, 1).
secondoOp(exttuplesize, postfix, 1).
secondoOp(attrsize, postfixbrackets, 2).
secondoOp(head, postfixbrackets, 2).
secondoOp(windowintersects, postfixbrackets, 3).
secondoOp(windowintersectsS, postfixbrackets, 2).
secondoOp(gettuples, postfix, 2).
secondoOp(sort, postfix, 1).
secondoOp(rdup, postfix, 1).
secondoOp(bbox, prefix, 1).
secondoOp(box3d, prefix, 2).
secondoOp(symmproduct, postfix, 2).
```

B Grammar of the Query Language

We use the following notational conventions. Words written in normal font are grammar symbols (non-terminals), words in bold face are terminal symbols. The symbols "->" and "|" are meta-

symbols denoting derivation in the grammar and separation of alternatives. Other characters like "*" or ":" are also terminals.

The notation X-*list* refers to a PROLOG list with elements of type X; as mentioned already, the square brackets can be omitted if the list has just one element. The notation X-*expr* refers to an expression built from elements of type X, constants, and operations available on X-values. Hence attr-*expr* is an expression involving attributes denoted in one of the two forms attrname or var:attrname. Similarly a predicate (pred) is a boolean expression over attributes. Finally, *empty* denotes the empty alternative. For example, the where-clause is optional.

```
query           ->  select distinct sel-clause from rel-list where-clause
                    orderby-clause first-clause
                  | select aggr-list from rel-list where-clause
                    groupby-clause orderby-clause first-clause

distinct        ->  distinct | empty

sel-clause      ->  * | count(*) | result-list

result          ->  attr | attr-expr as newname

attr            ->  attrname | var:attrname

rel             ->  relname | relname as var

where-clause    ->  where pred-list
                  | empty

pred            ->  attr-boolexpr

orderby-clause  ->  orderby orderattr-list
                  | empty

orderattr       ->  attrname | attrname asc | attrname desc

first-clause    ->  first int-constant
                  | empty

aggr            ->  groupattr | count(*) as newname
                  | aggrop(attr-expr) as newname

groupattr       ->  attr

aggrop          ->  min | max | sum | avg

groupby-clause  ->  groupby attr-list

mquery          ->  query
                  | union query-list
                  | intersection query-list
```