SECONDO

Programmer's Guide

Version 5, February 2, 2009

Ralf Hartmut Güting, Victor Teixeira de Almeida, Dirk Ansorge,

Thomas Behr, Markus Spiekermann, Christian Düntgen

FernUniversität in Hagen

Faculty for Mathematics and Computer Sience

Database Systems for New Applications

58084 Hagen, Germany

Table of Contents

1 Algebra Module Implementation

In SECONDO, data types and operations are provided by algebra modules. Such modules are initialized in SECONDO by the algebra manager which their data types and operations available in the query processor. Figure 2 in Section 1 of the SECONDO User Manual gives a good impression of SECONDO's structure and how algebras are linked into the system. Generally in this Programmers' Guide we assume that you are somewhat familiar with SECONDO concepts and use from the User Manual.

Naturally, an algebra implementor first has to write the C++-Code providing the data types and operations. Any such algebra must provide a set of algebra support functions, e.g. `In`- and `Out`-functions, which convert the "internal" data type representation to the nested list representation used as an "external" representation in SECONDO. Finally, when the implementation is done, the algebra has to be registered in the system.

In this chapter, two examples are given about how algebras are implemented and embedded in SECONDO. First, the simple `PointRectangleAlgebra` is described providing two new data types and two operations. Second, an algebra which provides operators for handling streams of `int` values is presented.

The SECONDO system has been developed over some time, and for some of its programming interfaces there exist older and more recent versions. The newer interfaces are usually safer and more comfortable to use. On the other hand, a lot of code in the system has been written using the older interfaces. For this reason, in the sequel we will sometimes explain old as well as recent versions so that you are able to understand older code, but also to program with the current versions. Note that older as well as newer interfaces work so that you can choose which one to use.

1.1 The PointRectangleAlgebra

The `PointRectangleAlgebra` is a simple algebra which was implemented just for the purpose of having a small example algebra for SECONDO. It offers data types to represent a point and a rectangle in the 2D-plane, and operations *inside* and *intersects*. The first checks whether either a point or a rectangle lies in another rectangle. The second checks whether two rectangles intersect each other. Formally, it has the following specification:

kinds SIMPLE

type constructors

	→	SIMPLE	<u>*xpoint*</u>, <u>*xrectangle*</u>

operators

<u>*xpoint*</u> × <u>*xrectangle*</u>	→	<u>*bool*</u>	inside
<u>*xrectangle*</u> × <u>*xrectangle*</u>	→	<u>*bool*</u>	inside
<u>*xrectangle*</u> × <u>*xrectangle*</u>	→	<u>*bool*</u>	intersects

The types are named *xpoint* and *xrectangle* in order to avoid name conflicts with the more complex types *point* and *rectangle* as implemented in the SpatialAlgebra module. Moreover, the type *bool* is not implemented in this algebra module, it belongs to the so called `StandardAlgebra`.

The implementation of the module `PointRectangleAlgebra` is part of the SECONDO distribution and is located in the directory `Algebras/PointRectangle`. All algebras must be located below directory `Algebras`. However, not all algebras, which are present there, have to be necessarily embedded in the system.

The file `PointRectangleAlgebra.cpp` can be viewed pretty printed by using `PDView` [Güt95], a tool for formatting programs and documentation which is included into SECONDO. It is strongly recommended to study this example carefully before starting to implement an own algebra. In the following, only important and interesting parts of the code are shown and explained. The complete code is provided in Appendix C.

The Typical Structure of an Algebra Implementation File

Every new algebra must be a subclass of the class `Algebra`. An algebra is a collection of type constructors and operators. For each new data type a C++-class together with some additional support functions must be provided. For each operator basically a type mapping and a value mapping function (explained later) must be provided. Then a type will be represented by class `TypeConstructor` and an operator by class `Operator`. The constructors of these C++ classes take the support functions as arguments. Finally, instances of `TypeConstructor` and `Operator` are inserted into the new Algebra subclass.

In the following we will give a sketch of the overall structure of an algebra's implementation file. First, we need some `include` directives in order to import class declarations of various SECONDO modules.

```
#include "Algebra.h"
#include "..."
```

Next, references to instances of the `NestedList` and the `QueryProcessor` classes are declared here. They are instantiated by the processing framework of SECONDO.

```
extern NestedList* nl;
extern QueryProcessor* qp;
```

Now, the classes for the new data types `xpoint` and `xrectangle` are defined, followed by the implementation of its operations. To avoid name conflicts, it is more safe to embed the algebra implementation into a namespace.

```
namespace prt {

class XPoint{ ... };
...
class XRectangle { ...};
...
```

The algebra itself is implemented by declaring a class derived from class `Algebra` and calling add-functions for all type constructors and operators in the constructor of this class.

```
class PointRectangleAlgebra : public Algebra
{
 public:
  PointRectangleAlgebra() : Algebra()
  {
    // code for adding type constructors and operators, e.g.
    AddOperator( intersectsInfo(), intersectFun, RectRectBool );
  }
  ~PointRectangleAlgebra() {};
};
} // end of namespace prt
```

Finally, the implementation file must be equipped with an initialization function which must be named by the pattern `Initialize`<*algebra name*>.

```
extern "C"
Algebra*
InitializePointRectangleAlgebra( NestedList* nlRef, QueryProcessor* qpRef )
{
  // The C++ scope-operator :: must be used to qualify the full name
  return new prt::PointRectangleAlgebra;
}
```

The implementation of this function will be always similar to the one above. For some implementation dependent reasons it is needed by the algebra manager.

1.2 Implementing Types

As explained above, each SECONDO type is represented by a C++ class. This class needs to provide some support functions which will be used by the query processing framework. These functions will be discussed in the following subsections.

1.2.1 Nested List Representation/Conversion

Every SECONDO type needs to define a nested list representation for its values. This representation is used as an external interface, e.g. to send the value to a user interface or to store it in a file. An element of a nested list may be either a nested list (this is why they are called “nested” lists) or an atom. Atoms of a nested list can be only simple values of type integer, boolean, string, text or symbol (mathematical symbols or identifiers). An algebra module must provide for each type functions which take a nested list and create an instance of the class which represents the type and vice versa. These functions are called `In-` and `Out`-functions.

An `XPoint` value has two coordinates, represented by integer values `x` and `y`. The nested list representation can be chosen freely, here we choose

```
(x y)
```

which means that every xpoint value can be represented as a list of two integer atoms. Inside SECONDO, nested lists are represented by the C++-type `ListExpr`. Textual nested lists are translated by a parser into this internal representation. Operations on type `ListExpr` are provided by a central instance of class `NestedList` which is referenced by the pointer variable `nl`.

```
ListExpr
XPoint::Out( ListExpr typeInfo, Word value )
{
  XPoint* point = static_cast<XPoint*>( value.addr );
  return nl->TwoElemList( nl->IntAtom(point->GetX()),
                          nl->IntAtom(point->GetY()) );
}
```

Nested lists are used in `XPoint::Out` as follows: In the first line, the argument which has the generic type `Word` must be converted by a type cast into a pointer of type `XPoint`. The type `Word` has a member `addr` which can be a pointer of any type. Then a new list with two elements is constructed with `TwoElemList`, which is a function of class `NestedList`. As arguments two `ListExpr` values which contain simple `int` values, so called atoms, are passed and inserted into the list. They are constructed using `IntAtom`, again a function of class `NestedList`. The `int` values are derived from the `XPoint` value using `GetX`- and `GetY`-functions of class `XPoint`. The use of the `NestedList` data structure is straightforward. Just have a look at file `include/NestedList.h`.

More recently, an alternate nested list programming interface which has a less noisy syntax has been implemented; you will find it in `NList.h`. The example algebra makes also use of it for the implementation of class `XRectangle`. For your own implementations we recommend to use this newer interface.

```
Word
XPoint::In( const ListExpr typeInfo, const ListExpr instance,
            const int errorPos, ListExpr& errorInfo, bool& correct )
{
  Word w = SetWord(Address(0));
  if ( nl->ListLength( instance ) == 2 )
  {
    ListExpr First = nl->First(instance);
    ListExpr Second = nl->Second(instance);

    if ( nl->IsAtom(First) && nl->AtomType(First) == IntType
      && nl->IsAtom(Second) && nl->AtomType(Second) == IntType )
    {
      correct = true;
      w.addr = new XPoint(nl->IntValue(First), nl->IntValue(Second));
      return w;
    }
  }
  correct = false;
  cmsg.inFunError("Expecting a list of two integer atoms!");
  return w;
}
```

`XPoint::In` creates a new `XPoint` instance. Both `int` values are extracted from the nested list and passed to the `XPoint` type constructor. Finally, the newly created `XPoint` instance is returned. Note, that in the `In`-function it is necessary to check carefully and completely whether the passed list has the correct structure. Such lists can be written by users directly (using a text editor) and may have other structures than expected. Accessing a not existing element of a list will cause process abortion raised by assertions inside the implementation of class `NestedList`.

The parameters `errorPos` and `errorInfo` can be ignored in simple cases like this one. In principle they can be useful to provide detailed error information for types which can have big lists as representations, e.g. type *rel*. Success or failure must be indicated by setting parameter `correct` to its appropriate value. In case of failure an error message is sent to the user by using the function `cmsg.inFunError`.

1.2.2 Persistent Storage and Related Generic Functions

A SECONDO object belongs to a database and needs to be stored persistently. Hence it needs a representation on disk, in files or records of the underlying storage manager.

To be used in query processing, an object must be *opened*. That is, in addition to the disk representation some main memory representation must be created that makes it possible to access the value. Hence an object has two states:

- *closed*: only the disk representation exists,
- *opened*: disk and memory representations exist.

There are six generic functions for each type constructor that allow one to create and delete a value (or SECONDO object) of the type, open and close it, save it and clone it. These operations return a value in a particular state and are also applicable to a value in some state. The state diagram in Figure 1 shows how operations manipulate object states.

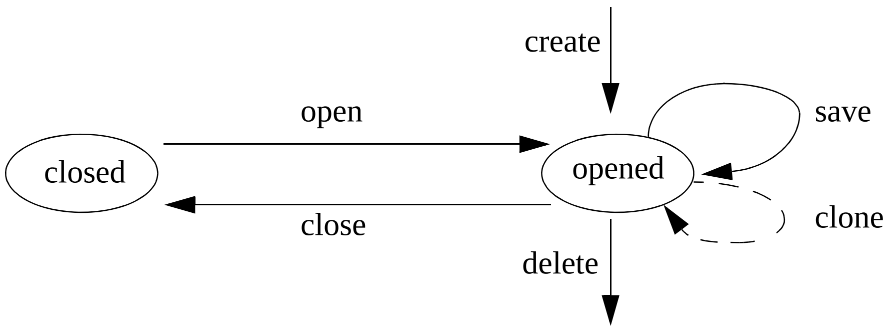


Figure 1: Generic functions and object states

The operations have the following meaning:

- `create`: create a value in open state (disk and memory part)
- `delete`: delete the value, i.e., remove disk part and memory part
- `open`: given a disk representation, create the memory part
- `close`: release the memory part
- `save`: propagate changes from the memory part to the disk part
- `clone`: make a deep copy of the object (disk part and memory part)

To make life easier for the implementor of a simple data type like `int` or `xpoint`, there exists a default mechanism for persistent storage. Here the text form of the nested list representation for the type is stored in a record on disk. Hence to store an object, its `Out` function is called; the resulting list is converted to text form and written to the record. To open an object, the text representation is read from the record and converted to a nested list for which then the `In` function is called to create the value.

This means that for a type constructor the functions `create`, `delete`, `close`, and `clone` *must* be implemented. The remaining two functions open and save *may* be implemented. If they are not implemented, the default persistent storage mechanism is used.

To make things yet a bit more complex, there is a variant of the default mechanism. Observe that the `In` function needs to perform a complete check of the argument list to see whether it has a correct structure and whether the value described by it is correct. For example, for the *region* data type in the spatial algebra, expensive tests need to be made. To avoid this, it is possible to define an alternative nested list structure just to be used internally for storage. In this case, the data structure (e.g. for a *region*) can be directly transformed into an appropriate list structure describing the elements of this data structure, and it is easy to recreate the data structure from this list representation.

To use an alternative list structure, the implementor of a data type needs to provide two functions called `SaveToList` and `RestoreFromList`.

Hence for persistent storage, there are three alternatives:

- If `open` and `save` are implemented, a specific persistent implementation is used.
- Otherwise, if `SaveToList` and `RestoreFromList` are implemented, then the default mechanism is used but with a specific list structure.
- Otherwise, the default mechanism is used with the external list representation, using `In` and `Out` functions.

The module `PointRectangleAlgebra` uses the default mechanism for the type *xpoint* and (for demonstration) an implementation of `open` and `save` for the type *xrectangle*. The implementations for the other object state transitions of *xpoint* are discussed below.

```
Word
XPoint::Create( const ListExpr typeInfo )
{
  return (SetWord( new XPoint( 0, 0 ) ));
}
```

This function creates a new `XPoint` instance. The query processor calls it if the result of an operation is of type *xpoint*.

```
void
XPoint::Delete( const ListExpr typeInfo, Word& w )
{
  delete (XPoint*)w.addr;
  w.addr = 0;
}
```

The delete-function is called for intermediate results in a query tree, after the query has been processed. `XPoint::Delete` does not need to remove any disk parts of the object since it is just a simple type which does not have disk parts. For such types delete and close often do the same.

Here the expression `(XPoint*)w.addr` does a C++ type cast, telling the C++ runtime environment that there is a pointer of type `XPoint`. Thus delete will call the destructor function of class `XPoint`. It is only sure to do such type casts if you know that `w.addr` really holds a pointer of this type, which is true here, since the create function above does so. But in general type casts are a source of pointer errors. For that reason another syntax, as shown below, is recommended since it is more noticable. Moreover, after deleting a pointer it is always safe to set it to null.

```
void
XPoint::Close( const ListExpr typeInfo, Word& w )
{
  delete static_cast<XPoint *>( w.addr; )
  w.addr = 0;
}
```

The close-function is called for all database objects which are involved in a query, since they were all opened before.

```
Word
XPoint::Clone( const ListExpr typeInfo, const Word& w )
{
  XPoint* p = static_cast<XPoint *>( w.addr; )
  return SetWord( new XPoint(*p);
}
```

The clone-function is used by the `let` command which inserts the result of a query into the currently open database. Here `XPoint::Clone` calls the copy constructor `XPoint(`**`const XPoint&`**`)` to create a new instance which is a copy of the one pointed to by `p`. For a more complex data type which has also a disk part, code doing a deep copy must be provided here also.

Auxiliary Functions

There are two more functions which are important in the context of persistent storage: a function which returns the size of the type's representing class and a function which reconstructs an object which was loaded from disk to memory.

The `SizeOfObj`-function must be implemented to provide information about the object's size.

```
int
XPoint::SizeOfObj()
{
  return sizeof(XPoint);
}
```

When types should act as attributes of relations (refer to Section 4) a so called `Cast`-function is needed for the proper re-construction of persistent C++ objects. This function would look as follows:

```
void* XPoint::Cast( void* addr ) {
  return (new (addr) XPoint);
}
```

But it is not implemented in the example algebra, since the type is not intended to be used as an attribute type in relations. This special syntax of the C++ new operator tells the C++ runtime environment that there is an object of type `XPoint` at address `addr`. The memory pointed to by `addr` must be unchanged by this operation.

Note: This can only be guaranteed if the standard constructor is an empty function. Moreover, this function should be the only one which calls the standard constructor.

1.2.3 Kind Checking

When entering a type expression into a user interface, the query processor first does a kind checking. This means that the structure of the argument passed to each type constructor is checked. If it is not correct, the input is not accepted. Actually, this checking is done by so-called kind checking functions implemented in the algebra. At this point we have to specify the kind checking function for the `XPoint` constructor. Since it has no arguments, this is trivial.

```
bool
XPoint::CheckKind( ListExpr type, ListExpr& errorInfo )
{
  return (nl->IsEqual( type, XPOINT ));
}
```

In contrast to this example, writing kind checking functions may get much more difficult for other types such as relations in the `RelationAlgebra`. There for example you have to check if the type of every attribute is a member of kind `DATA`.

Note: don't mix up the symbol `XPOINT` (all letters capitalized) with the class `XPoint` here. The first one is defined in file `include/Symbols.h` where every algebra should define its string symbols.

1.2.4 Type Description

At the user interface, the command `list type constructors` lists all type constructors of all currently linked algebra modules. The information listed is generated by the algebra module itself. To be more precise, it is generated by the *property*-function of each type. Basically, this function returns a nested list which explains the type. The older interface for this looks as follows. In the `PointRectangleAlgebra` it is used for `XPoint`.

```
ListExpr
XPoint::Property()
{

  return (nl->TwoElemList(
            nl->FiveElemList(nl->StringAtom("Signature"),
                             nl->StringAtom("Example Type List"),
                             nl->StringAtom("List Rep"),
                             nl->StringAtom("Example List"),
                             nl->StringAtom("Remarks")),
            nl->FiveElemList(nl->StringAtom("-> DATA"),
                             nl->StringAtom("xpoint"),
                             nl->StringAtom("(<x> <y>)"),
                             nl->StringAtom("(-3 15)"),
                             nl->StringAtom("x- and y-coordinates must be "
                             "of type int."))));
}
```

Basically a list has to be returned with two sublists where the first contains labels and the second entries for these labels.

There exists a more recent interface for this, demonstrated for type `XRectangle`. In this case one has to implement a subtype of class `ConstructorInfo`, e.g.

```
struct xrectangleInfo : ConstructorInfo {

  xrectangleInfo() : ConstructorInfo() {

    name         = XRECTANGLE;
    signature    = "-> " + SIMPLE;
    typeExample  = XRECTANGLE;
    listRep      =  "(<xleft> <xright> <ybottom> <ytop>)";
    valueExample = "(4 12 8 2)";
    remarks      = "all coordinates must be of type int.";
  }
};
```

Here again the symbols `XRECTANGLE` and `SIMPLE` are string constants defined in file `include/Symbols.h`.

1.3 Implementing Operators

An operator implementation needs to provide the following:

1. A type mapping function, which checks, whether its argument types are correct or not. If the arguments are correct the result type (not value) will be returned.
2. A selection function to decide for overloaded operators which of several value mapping functions should be used.
3. A value mapping function, which calculates the result value of an operation.
4. An operator description, to be used by the `list operators` command.
5. A syntax specification and an example query using this operator, for automatic testing.

1.3.1 Type Mapping Functions

Before the query processor builds an operator tree, the query is analyzed and annotated. During this phase a list containing the arguments' type expressions is passed to the operator's type mapping function. A type mapping function checks, whether its argument types are correct or not.

If not, the result type is a list expression consisting of the symbol `typeerror` which will also cause subsequent type mappings to fail. Thus the query will be not accepted in this case, since it contains a type mismatch. Generally it is a good idea not only to return `typeerror`, but also to explain what kind of error occurred.

```
ListExpr
insideTypeMap( ListExpr args )
{
  if ( nl->ListLength(args) != 2 )
  {
    cmsg.typeError("Type mapping function got a "
                                "parameter of length != 2.");
    return nl->TypeError();
  }
```

```
    ListExpr arg1 = nl->First(args);
    ListExpr arg2 = nl->Second(args);

    if ( nl->IsEqual(arg1, XPOINT) && nl->IsEqual(arg2, XRECTANGLE) )
      return nl->SymbolAtom(BOOL);

    // second alternative of expected arguments
    if ( nl->IsEqual(arg1, XRECTANGLE) && nl->IsEqual(arg2, XRECTANGLE) )
      return nl->SymbolAtom(BOOL);

    if ((nl->AtomType(arg1) == SymbolType) &&
        (nl->AtomType(arg2) == SymbolType))
    {
      cmsg.typeError("Type mapping function got parameters of type "
                             +nl->SymbolValue(arg1)+" and "
                             +nl->SymbolValue(arg2));
    }
    else
    {
      cmsg.typeError("Type mapping function got wrong "
                             "types as parameters.");
    }
    return nl->TypeError();

  }
```

1.3.2 Selection Functions

Functions may be overloaded. Selection functions are used to select one of several evaluation functions for an overloaded operator, based on the types of the arguments. Below the selection function for operator **inside** is shown. Here the alternate list programming interface defined in `include/NList.h` is used.

```
int
insideSelect( ListExpr args )
{
  NList list(args);
  if ( list.first().isSymbol( XRECTANGLE ) )
    return 1;
  else
    return 0;
}
```

We will see below that an array of value mapping functions is constructed; the index 0 or 1 selects the appropriate function from that array.

1.3.3 Generic Type Mapping and Selection Functions

Many operators have very simple type mappings in the sense that their arguments and result types are atomic. The examples above also fall into this category. To liberate the user from the burden of implementing such type mappings, there are some generic functions called `SimpleMap` and `Simple-`

`Select` which can handle those simple cases.[1] Examples can be found in the `StandardAlgebra`. For example, the comparison operator “=” is overloaded many times but its type mappings and selection functions can be simply implemented by the following code lines:

```
const string maps_comp[6][3] =
{
  {INT,    INT,    BOOL},
  {INT,    REAL,   BOOL},
  {REAL,   INT,    BOOL},
  {REAL,   REAL,   BOOL},
  {BOOL,   BOOL,   BOOL},
  {STRING, STRING, BOOL}
};

ListExpr
CcMathTypeMapBool( ListExpr args )
{
  return SimpleMaps<6,3>(maps_comp, args);
}

int
CcMathSelectCompare( ListExpr args )
{
  return SimpleSelect<6,3>(maps_comp, args);
}
```

For non overloaded functions it is even more simple:

```
ListExpr
IntReal( ListExpr args )
{
  const string mapping[] = {INT, REAL};
  return SimpleMap(mapping, 2, args);
}
```

One only needs to define an array of mappings as shown above and to pass the array and its dimensions to the generic functions.

1.3.4 Value Mapping Functions

For any operation and each allowed combination of argument types a value mapping function must be defined. Inside of these functions the internal functions are applied to the passed arguments. The resulting value then is written directly to an object provided by the query processor. Below we show the code for the value mapping of operator **inside** in the case of *xpoint* × *xrectangle*→ *bool*.

1. Those functions are part of namespace `mappings`.

```
int
insideFun_PR ( Word* args, Word& result,
                int message, Word& local, Supplier s)
{
  XPoint* p = static_cast<XPoint*>( args[0].addr );
  XRectangle* r = static_cast<XRectangle*>( args[1].addr );

  result = qp->ResultStorage(s);   //query processor has provided
                                   //a CcBool instance to take the result

  CcBool* b = static_cast<CcBool*>( result.addr );

  bool res = ( p->GetX() >= r->GetXLeft() && p->GetX() <= r->GetXRight()
        && p->GetY() >= r->GetYBottom() && p->GetY() <= r->GetYTop() );

  b->Set(true, res); //the first argument says the boolean
                     //value is defined, the second is the
                     //real boolean value)
  return 0;
}
```

The parameters `message` and `local` can be ignored here. They are only needed for operators which process streams and are explained later in other example code. Pointers to the arguments of the operator are stored in the `args` array and need to be type casted to pointer variables of their respective type. The result of this operation is a boolean value represented by the C++ class `Ccbool`. Apart from a `bool` member variable which holds the value it has another `bool` member which tells whether a value for it is defined or not. Many types offer the alternative value `undefined` which allows to return values of the correct type in critical cases, e.g. division by zero.

In total there are two value mapping functions collected in an array of pointers to functions.

```
ValueMapping insideFuns[] = { insideFun_PR, insideFun_RR, 0 };
```

The implementor has to take care, that (i) this array is null terminated, (ii) the value mapping alternatives are on their correct positions with respect to the operator's selection function.

1.3.5 Operator Descriptions

For use in the `list operators` command, each operator needs to supply a description. This is similar to the information about type constructors above. We show them for both operators `inside` and `intersect` to illustrate also the description of an overloaded operator.

```
struct intersectsInfo : OperatorInfo {

  intersectsInfo() : OperatorInfo()
  {
    name      = INTERSECTS;
    signature = XRECTANGLE + " x " + XRECTANGLE + " -> " + BOOL;
    syntax    = "_" + INTERSECTS + "_";
    meaning   = "Intersection predicate for two xrectangles.";
  }

};
```

```
struct insideInfo : OperatorInfo {

  insideInfo() : OperatorInfo()
  {
    name      = INSIDE;

    signature = XPOINT + " x " + XRECTANGLE + " -> " + BOOL;
    // since this is an overloaded operator we append
    // an alternative signature here
    appendSignature( XRECTANGLE + " x " + XRECTANGLE
                                              + " -> " + BOOL );
    syntax    = "_" + INSIDE + "_";
    meaning   = "Inside predicate.";
  }
};
```

1.3.6 Operator Syntax and Examples

Every algebra must provide syntax and example specifications. They are defined in the following files:

1. The `.spec`-file, which provides important information for the parser about the algebra's operators.
2. The `.example`-file, which provides query examples for the implemented operations.

The `.spec` file for the `PointRectangleAlgebra` looks like this:

```
operator intersects alias INTERSECTS pattern _ infixop _
operator inside alias INSIDE pattern _ infixop _
```

As you can see, both available operators are listed. By convention all operators are written in lower case. The structure of such a specification is:

```
operator <name> alias <ALIAS> pattern <pattern>
```

The `ALIAS` is the name of the token for the operator needed for the lexical analysis. This is needed for operators which use a mathematical symbol, e.g. +. All other operators can simply use the capitalized version of their name. Then, the `pattern` for the operator is given. The symbol _ denotes the places of arguments and `infixop` shows the position of the operator. Therefore, the operator `intersects` would be used as `a intersects b` in a query. Other examples for operator specifications can be found in the complete `spec`-file which is the concatenation of all algebra `.spec`-files. This file is located in the `Algebras` directory.

Note, that operator names may be used in more than one algebra. It is necessary that the syntax specification of operators with the same name is identical. Otherwise, an error will be reported during the compilation process.

The `.example` file is located in the directory of the respective algebra and parsed during the startup of `Secondo`. If examples are missing, you will notice error messages. This guarantees, that all examples have a correct syntax. Moreover, the examples need to specify a database on which they can be processed together with expected results. This is done at the beginning of the file:

```
Database : prttest
Restore  : NO
```

The `Restore` flag indicates whether the database should be restored before the following example queries are executed. For every operator records like the following must be defined.

```
Operator : inside
Number   : 1
Signature: xpoint x xrectangle -> bool
Example  : query p1 inside r1
Result   : TRUE
```

In case of an overloaded operator the field `Number` must contain subsequent numbers. The other fields should be self explanatory.

Example queries are executed when the command `Selftest <example-file>` is invoked in the `secondo/bin` directory; the example file is (after a `make`) available in a subdirectory `tmp`. The command `Selftest` executes all example queries of all activated algebras in SECONDO.

1.3.7 Linking the PointRectangleAlgebra to SECONDO

To link the algebra to the SECONDO system, it has to be registered in the algebra manager and to be plugged into the build process. Therefore it has to be entered in one configuration file and a `makefile`.

The first thing to do is to create a new directory in the `Algebras` directory of the SECONDO system. By convention the new directory name should be the same as the algebra name omitting the suffix “Algebra” (`PointRectangle` in this case). Afterwards the algebra file has to be copied to the new directory.

Then you need a `makefile` for the algebra directory including information about which files have to be compiled during the compilation process of the system. To create the `makefile` the easiest way is to copy a `makefile` from another algebra module and adapt it to the new algebra. In most cases you need to do nothing since there are default rules for creating `.o` files from `.cpp` files.

Next, the file `makefile.algebras` (located in the SECONDO main directory) has to be changed. This file contains two entries for every algebra. The first defines the directory name and the second the name of the algebra module like in the example below :

```
...
ALGEBRA_DIRS += PointRectangle
ALGEBRAS     += PointRectangleAlgebra

ALGEBRA_DIRS += BTree
ALGEBRAS     += BTreeAlgebra
```

```
    ...
```

The last step is to register the algebra in the algebra manager. To do this it has to be added to the list of algebras in `AlgebraList.i.cfg` in the `Algebras/Management` directory. Here, the algebra must be entered together with its name implied by the name of the initialize-function and a unique algebra number (just choose an unused number).

```
    ...
    ALGEBRA_INCLUDE(1,StandardAlgebra)
    ALGEBRA_INCLUDE(2,FunctionAlgebra)
    ALGEBRA_INCLUDE(3,RelationAlgebra)
    ALGEBRA_INCLUDE(4,PointRectangleAlgebra)
    ALGEBRA_INCLUDE(5,StreamExampleAlgebra)
    ...
```

Now, all configuration is done for the `PointRectangleAlgebra`. Execute the `make` file in the SECONDO main directory to link the new algebra. After the process has finished, start SECONDO and type `list algebra PointRectangleAlgebra` to see its operators and type constructors.

1.4 Handling Streams

The `StreamExampleAlgebra` is a small example demonstrating how to implement operators which process streams of objects. For data types which may have very big representations like a relation with millions of tuples, streaming is necesseary to provide efficient implementations (otherwise big intermediate results must be materialized) of operations. Hence this example helps to understand much more sophisticated algebras using streams such as the `RelationAlgebra`.

In this example algebra, operators for the construction of streams of `int` values are provided together with some operators which have such streams as arguments. In contrast to the `PointRectangleAlgebra` this algebra doesn't have any constructors, since a new stream is constructed using an operator, not a constructor. The algebra provides the following operators:

- **intstream**: `int` × `int` → `stream(int)`
  Creates a stream of integers containing all integers from the first up to the second argument. If the second argument is smaller than the first, the stream will be empty.
- **count**: `stream(int)` → `int`
  Returns the number of elements in an integer stream.
- **printintstream**: `stream(int)` → `stream(int)`
  Prints out all elements of the stream. Returns the argument stream unchanged.
- **filter**: `stream(int)` × `(int` → `bool)` → `stream(int)`
  Filters the elements of an integer stream by a predicate.

Again, we strongly recommend to have a close look at the implementation of the algebra. Here, we only describe the new concepts and the exciting parts of the new algebra.

Type Mapping Functions

The first thing to do is to implement the type mapping functions for the four operators. As an example for this the mapping for **intstream** will serve:

```
ListExpr
intstreamType( ListExpr args )
{
  const string errMsg =   "Type mapping function expects (int int)";

  if ( nl->ListLength(args) != 2 )
    ErrorReporter::ReportError( errMsg );

  ListExpr arg1 = nl->First(args);
  ListExpr arg2 = nl->Second(args);

  if ( nl->IsEqual(arg1, INT) && nl->IsEqual(arg2, INT) )
    return nl->TwoElemList(nl->SymbolAtom(STREAM), nl->SymbolAtom(INT));

  ErrorReporter::ReportError( errMsg );
  return nl->TypeError();
}
```

Note that a stream is treated like an atom in the nested list representation. Nothing more of this piece of code should be new. The type mapping for other operators is quite similar.

Value Mapping Functions

To be able to understand how the value mapping functions work, we have to take a closer look at stream operators in general. Stream operators manipulate streams of objects. They may consume one or more input streams, produce an output stream, or both. For a given stream operator α, let us call the operator receiving its output stream its successor and the operators from which it receives its input streams it predecessors. Now, stream operators work as follows: Operator α is sent a `REQUEST`-message from its successor to receive a stream object. Operator α in turn sends a `REQUEST` to its predecessors. The predecessors either deliver the object (sending a `YIELD` message) or don't have objects any more (sending a `CANCEL` message). Figure 2 shows the protocol dealing with streams.

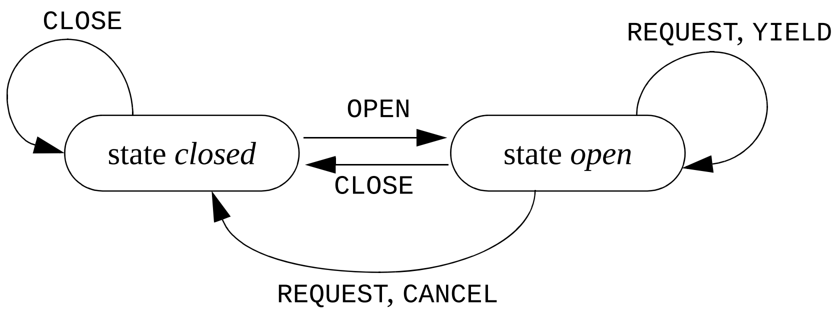


Figure 2: State Diagram for Streams

The first message sent to an operator producing a stream must be `OPEN`. This converts the stream from the state *closed* to the state *open*. Afterwards, either `REQUEST` or `CLOSE` messages can be sent to the stream. Sending `REQUEST` means, that the successor would like to receive an object. The response can be a `YIELD` message (giving the object to the successor, remaining in state *open*) or a `CANCEL` message (no further objects are available, switching to state *closed*). The successor may send a `CLOSE` message, which means that it does not wish to receive any further objects even though they may be available. This also transforms the stream into state *closed*.

When trying to simulate stream operators by algebra functions, it can be observed that a function need not relinquish control (terminate) when it sends a message to a predecessor. This can be treated pretty much like calling a parameter function. However, the function needs to terminate when it sends a `YIELD` or `CANCEL` message to the successor. This makes it necessary to write the function in such a way that it has some local memory and that each time when it is called, it just delivers one object to the successor.

In the following example for a value mapping function for `intstream`, the possible messages are encoded by the special symbols:

`OPEN, REQUEST, CLOSE, YIELD,` and `CANCEL`

The messages are passed as parameter and in the case of an request message `YIELD` and `CANCEL` are used as return value. The parameter `local` is used to store local variables that must be maintained between calls to the stream. More information about stream operators can be found in [GFB+97].

```
int
intstreamFun
  (Word* args, Word& result, int message, Word& local, Supplier s)
{
  struct Range {  // an auxiliary record type
    int current;
    int last;

    Range(CcInt* i1, CcInt* i2) {
      if (i1->IsDefined() && i2->IsDefined())
      {
        current = i1->GetIntval();
        last = i2->GetIntval();
      }
      else
      {
        current = 1;
        last = 0;
      }
    }
  };
  Range* range = 0;
  CcInt* i1 = 0;
  CcInt* i2 = 0;
  CcInt* elem = 0;
```

```
    switch( message )
    {
      case OPEN: // initialize the local storage

        i1 = ((CcInt*)args[0].addr);
        i2 = ((CcInt*)args[1].addr);
        range = new Range(i1, i2);
        local.addr = range;
        return 0; // no special return message

      case REQUEST: // return the next stream element

        range = ((Range*) local.addr);
        if ( range->current <= range->last )
        {
          elem = new CcInt(true, range->current++);
          result.addr = elem;
          return YIELD;
        }
        else
        {
            result.addr = 0;
          return CANCEL;
        }

      case CLOSE: // free the local storage
        range = ((Range*) local.addr);
        delete range;
        return 0;
    }
    /* should never happen */
    return -1;
  }
```

As we can see, three different messages, namely `OPEN`, `REQUEST` and `CLOSE`, are handled in the function. In the section for `OPEN` the argument values are extracted and a value for `range` is set and stored in the local variable. In the `REQUEST` section `range` is read and a new `int` value is computed if the current value for `range` is smaller than the last stream value. In this case, `YIELD` is returned. Otherwise the result is `CANCEL`. Finally, in the `CLOSE` section, the stream is closed.

The value mapping function for `count` shows how a stream is consumed:

```
  ...
  while ( qp->Received(args[0].addr) )
    {
      count++;
      ((Attribute*)elem.addr)->DeleteIfAllowed(); //consume stream object
      qp->Request(args[0].addr, elem);
    }
  ...
```

Note, that for any operator that produces a stream, its arguments are not evaluated automatically. To get the argument value, the value mapping function needs to use `qp->Request` to ask the query processor for evaluation explicitely. A call of `Received` returns `TRUE`, if the previous call of `Request` responded with a `YIELD` message. If it responded with `CANCEL`, the result is `FALSE`. Together with streaming a mechanism for reference counting is needed to avoid unnecessary in memory copies of objects and memory leaks. The `DeleteIfAllowed` function deletes a stream element if it is not used anywhere else. Section 2 will explain more details about this.

In `filterFun` a parameter function is used to let through only selected stream objects:

```
Word elem, funresult;
ArgVectorPointer funargs;

...

case REQUEST:

      // Get the argument vector for the parameter function.
      funargs = qp->Argument(args[1].addr);

      // Loop over stream elements until the function yields true.
      qp->Request(args[0].addr, elem);
      while ( qp->Received(args[0].addr) )
      {
        // Supply the argument for the parameter function.
        (*funargs)[0] = elem;

        // Instruct the parameter function to be evaluated.
        qp->Request(args[1].addr, funresult);
        CcBool* b = static_cast<CcBool*>( funresult.addr );
        bool funRes = b->IsDefined() && b->GetBoolval();

        if ( funRes )       // Element passes the filter condition
        {
          result = elem;
          return YIELD;
        }
        else              // Element is rejected by the filter condition
        {
          // consume the stream object (allow deletion)
          static_cast<Attribute*>(elem.addr)->DeleteIfAllowed();

          // Get next stream element
          qp->Request(args[0].addr, elem);
        }
      }

      // End of Stream reached
      result = SetWord(Address(0));
      return CANCEL;
```

2 The Relational Algebra and Tuple Streams

The Relational Algebra implements relations formed by a collection of tuples, which in turn contain a collection of attributes. It also provides basic operators to interoperate between relations and tuple streams, thus tuple streams can be created from a relation using the **feed** operator, while a relation can be created by applying the **consume** operator to a tuple stream. In this chapter, you will learn how to implement operators dealing with tuple streams and contained tuples and attributes.

At this point, you should be familiar with the SECONDO Relational Algebra as a user, which means that you know how to create and query relations.

You should also know about creating algebras in detail, i.e., creating type constructors, operators, etc. because more sophisticated concepts about type constructors and operators will be handled in this section. You should know, which functions a SECONDO data type may inherit from class `Attribute` and which of these you are required to overwrite.

It is also important that you understand the concepts in the `StreamExampleAlgebra`, namely data streams and parameter functions, which are used in almost all operators in the Relational Algebra.

2.1 The Relational Algebra Implementation

The Relational Algebra provides two type constructors: *rel* and *tuple*. The structural part of the relational model can be described by the following signature:

kinds IDENT, DATA, TUPLE, REL

type constructors		**example typeconstructors**
	$\rightarrow$ DATA	*int*, *real*, *string*, *bool*
$(\text{IDENT} \times \text{DATA})^+$	$\rightarrow$ TUPLE	*tuple*
TUPLE	$\rightarrow$ REL	*rel*

Therefore a tuple is a list of one or more pairs <*identifier*, *type*>. A relation is built from such a tuple type.

In the Relational Algebra, relations are files (`SmiRecordFile` class) containing variable length records (`SmiRecord` class), each record storing one tuple. For more detail on files and records, see Section 5.

Another important structure that is used by the Relational Algebra is the Database Array (`DBArray` class). DBArrays are used to implement complex attribute types, such as *region* for example.

To explain its structure, we need first to explain the concept of FLOBs, which stands for **F**aked **L**arge **OB**jects. There is a tradeoff between storing large objects inline with tuples and in a separate record. Not always an instance of a large object uses a large amount of storage. As an example, imagine that the <u>*region*</u> type constructor has a large object to store its array of segments. Imagine then that we have in the system a relation that stores rectangles as regions. Rectangles need to store only four segments and then, the storage needed for rectangles is not large. Therefore, it is preferable to store the rectangles inline with the tuple. To solve this problem, the concept of a FLOB was created. It is an abstraction of a large object (LOB) that decides where to store the objects. If the size of the large object is smaller than a specified threshold, then the "large" object is stored inline with the tuple, and otherwise it is stored in a separate record.

Every relation that contains at least one attribute with FLOBs has a separate variable length record file to store the large FLOBs. It is important to note that large FLOBs are only read from disk when needed.

Database arrays are constructed on top of FLOBs. The difference between FLOBs and DBArrays is that a FLOB is a sequence of bytes (in disk or in memory) without structure, and a DBArray is a structured array implemented as a C++ template. For more detailed information about FLOBs and DBArrays, see Section 3.

For efficient retrieval of FLOBs, a FLOB cache is used. The size of the FLOB cache can be set in the `SecondoConfig.ini` configuration file. The FLOB cache is also important for better memory utilization, since there is a limit of memory utilization per operator.

We are now able to present the tuple representation (Figure 3):

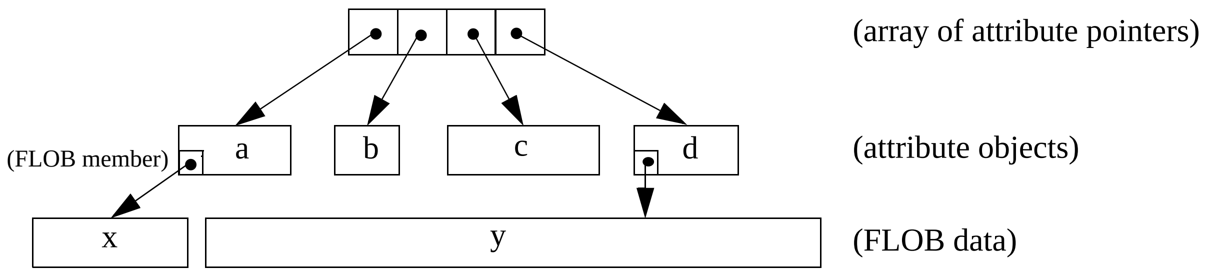


Figure 3: Tuple Representation in Memory

A tuple contains an array of pointers to attributes (class `Attribute` and its subclasses). It may be the result of the In-function for example, where every attribute is created separately. When a tuple needs to be written to disk (see Figure 4), the "root blocks" of all attributes (a, b, c, d) are copied to form a contiguous block in memory. "Small" FLOBs ("x" in this case) are stored also contiguously after the attributes in a memory block called tuple-extension. Then, the complete block is moved to a "Record File" on disk. Large FLOBs are kept separately; they are written into a separate "FLOB File" on disk. When a tuple is read from disk, its compacted root record is brought into memory and the structure described by Figure 3 is reconstructed. Data from the FLOB File is only loaded, when explicitly accessed.

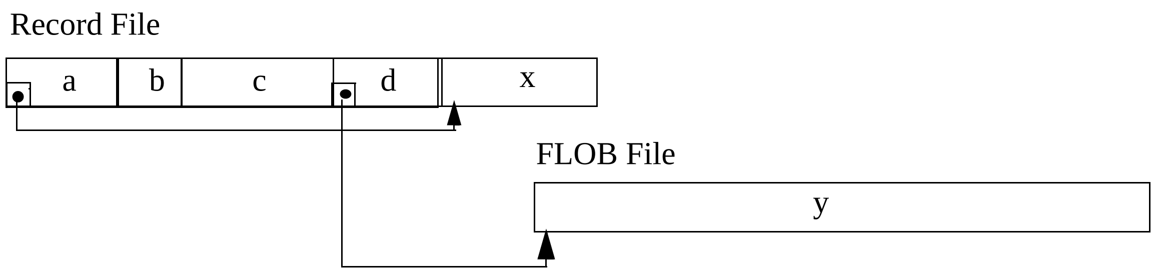


Figure 4: Tuple Representation on Disk

Finally, to avoid copying tuples and attributes when it is not necessary, references are passed and reference counters are kept. The object data is only really deleted when the reference counter is 0. The following functions are used in order to provide this functionality:

Attribute

For attributes, reference counters are maintained automatically:

`Copy`

This function is used to create a new reference to an attribute, incrementing the reference counter of the attribute. Since we use one byte to store this reference counter in order to keep it small, it is possible to overflow this value. In this case a clone of the attribute is done using the next function. Remember, that due to this fact, you cannot be sure, that changes to the object data of one reference will affect all “copies“ at once!

`Clone`

This (virtual abstract) function does a forced clone of the attribute. Each clone has its own object data, and reference counter, starting with reference counter `refs = 1`.

`DeleteIfAllowed`

This function decreases the reference counter and, if it becomes 0, deletes the attribute.

Tuple

For each tuple, it is within the responsibility of the programmer to maintain the correct state of the reference counters. You must keep track of every single pointer to a tuple you maintain. When creatin a new tuple, the reference counter is initialized with 1.

`IncReference`

This function increments the reference counter and must be used when a copy of a tuple pointer has been created, and this pointer is kept by some operation. If you call `IncReference`, you are obliged to call `DeleteIfAllowed` when releasing the reference.

`DeleteIfAllowed`

This function decreases the reference counter of a tuple and deletes it if the counter becomes 0. You must call this function, if you give up any reference to a tuple (and the tuple is not used any more). If you hand over the tuple reference, e.g. as the result of a value mapping function, do not call `DeleteIfAllowed`, unless you created further references by yourself.

`CopyAttribute`

This function is very important and copies an attribute from one tuple to another. It calls the `Copy` function of the attribute.

In the next sections the implementation of some operators is explained in detail. If you look for the implementation files you should know that the Relational Algebra is divided into two algebra modules called `RelationAlgebra` and `ExtRelationAlgbra`. The first one implements the *rel* and *tuple* type constructors and basic operators like **feed**, **project**, and **consume** whereas the second one implements some more special operations. There is even a third module, by historic reasons it is called `OldRelationAlgebra`. It provides the type constructors *mrel* and *mtuple*, which implement relations which stay in memory. We will not explain the `OldRelationAlgebra` any further here.

2.2 Operators

The most important operators in the relational algebra are described below:

- **feed**: produces a stream of tuples from a relation.
- **consume**: the contrary of **feed**, i.e., produces a relation from a stream of tuples.
- **rename**: changes only the type, not the value of a stream by appending the characters supplied as argument to each attribute name.
- **filter**: receives a stream of tuples and passes along only tuples for which the parameter function evaluates to `true`.
- **attr**: retrieves an attribute value from a tuple. The dot "." notation is used inside some operators, like **filter** for example.
- **project**: implements the relational projection operation on streams.
- **product**: implements the relational cartesian product operation on streams.
- **count**: counts the number of tuples in a stream or in a relation.

It is important to note that most of the operators of the relational algebra run on streams of tuples instead of directly on relations. Exceptions are the **feed** operator, that produces the streams, and the **count** operator, that is allowed to count the number of tuples also directly on relations.

2.3 Type Mapping Functions and APPEND

Up to now, in the algebra implementation tasks, the type mapping functions of all operators were very simple. A type mapping function takes a nested list as an argument. Its contents are type descriptions of an operator’s input parameters. A nested list describing the output type of the operator is returned. The **feed** operator’s type mapping function, for example, can be described as

```
((rel x)) -> (stream x)
```

where *x* is a tuple type. This means that it receives a relation of tuples of type *x* as an argument and returns a stream of the same tupletype *x*.

In the relational algebra we have some type mapping functions that are quite complex and use the special keyword `APPEND` in the result type. We will show the need and the effects of the `APPEND` keyword using the **attr** and **project** type mapping functions as examples.

Attr Type Mapping Function

The **attr** operator takes a tuple and retrieves an attribute value from it. The type mapping function should be:

((tuple ((x1 t1)...(xn tn))) xi) $\rightarrow$ ti)

where xi is an attribute name, and ti is its data type, both indexed by i. The type mapping function of the attr operator is:[1]

```
ListExpr AttrTypeMap(ListExpr args)
{
  ListExpr first, second, attrtype;
  string  attrname, argstr;
  int j;

  CHECK_COND(nl->ListLength(args) == 2,
    "Operator attr expects a list of length two.");

  first = nl->First(args);
  CHECK_COND( (nl->ListLength(first) == 2) &&
              (TypeOfRelAlgSymbol(nl->First(first)) == tuple),
  "Operator attr expects as first argument a list with structure "
  "(tuple ((a1 t1)...(an tn))).");

  second  = nl->Second(args);
  CHECK_COND( nl->IsAtom(second) &&
              nl->AtomType(second) == SymbolType,
  "Operator attr expects as second argument a symbol atom "
  "(attributename).");
```

1. Some error reporting code has been omitted to increase readability.

```
    attrname = nl->SymbolValue(second);
    j = FindAttribute(nl->Second(first), attrname, attrtype);
    if (j)
      return nl->ThreeElemList(nl->SymbolAtom("APPEND"),
             nl->OneElemList(nl->IntAtom(j)), attrtype);
    else
    {
      nl->WriteToString( argstr, nl->Second(first) );
      ErrorReporter::ReportError(
      "Attribute name '" + attrname + "' is not known in the tuple.\n"
      "Known Attribute(s): " + argstr);

      return nl->SymbolAtom("typeerror");
    }
  }
```

The “function“ `CHECK_COND` is a macro in real, that will be extended by the preprocessor. It first evaluates its first parameter, that should be a boolean expression. If it computes to `false`, it sends the error message passed as second parameter and `returns` with `typeerror` as result (ending the type mapping function). If the first parameter computes to `true`, nothing else happens (despite of possible side effects caused by the condition - which should be avoided).

The variable `first` contains the list `(tuple ((x1 t1)...(xn tn)))`and the `second` contains the attribute name `xi` that we are interested in retrieving. The function `FindAttribute` receives a list of pairs of the form `((x1 t1)...(xn tn))`, an attribute name and a data type that will be filled as a result. It then determines, whether the attribute name occurs as one of the attributes in this list. If so, the index in the list (beginning from 1) is returned and the corresponding data type is put in the attribute data type argument. Otherwise 0 is returned.

In this way, the value mapping function can get a tuple and an attribute name as arguments. But, in this level it does not know about the tuple type, and therefore it would be impossible to determine the attribute index. Moreover, it would be inefficient to compute the index once for every tuple processed in a stream. Since we calculated the index in the type mapping function, it would be nice that another argument should be added and used in the value mapping function. This will be done using the `APPEND` keyword. The technique used is that the type mapping function returns not just the mere result type, but a list of the form

```
(APPEND (<newarg1> ... <newargn>) <resulttype>)
```

`APPEND` tells the query processor to add the elements of the following list `(<newarg1> ... <newargn>)` to the argument list of the operator as if they had been written in the query. Using this approach, we can pass the attribute index as another argument to the value mapping function of the **attr** operator as it is shown below.

```
((tuple ((x1 t1)...(xn tn))) xi)    -> (APPEND (i) ti)
```

This resulting type mapping function will pass the tuple `t`, the attribute name `xi` and the attribute index `i` to the value mapping function as arguments, and set the attribute type `ti` to be the result type of the operator.

The main difference is that instead of returning only the attribute type, the type mapping function returns a three element list containing first the keyword APPEND, then the attribute index, i.e. what we want to be appended, and finally the attribute type, which will be the result type of the operator. The value mapping function[1] is then:

```
int
Attr(Word* args, Word& result, int message, Word& local, Supplier s)
{
  Tuple* tupleptr;
  int index;

  tupleptr = (Tuple*)args[0].addr;
  index = ((CcInt*)args[2].addr)->GetIntval();
  result = SetWord(tupleptr->GetAttribute(index - 1));
  return 0;
}
```

It takes the arguments from the vector `args`. The first argument in position 0 is the tuple, the second (position 1) is the attribute name which is not used in the value mapping (but was used in the Type Mapping, as described before), and the third in position 2 is the attribute index passed with the APPEND command. The function then returns the attribute value at this position pointed to by the attribute index.

Project Type Mapping Function

Following the same idea presented above for the **attr** operator, the **project** operator also uses the APPEND command. The **project** operator's type mapping function acts like the description below.

```
((stream (tuple (($x_1$ $T_1$) ... ($x_n$ $T_n$)))) ($a_{i1}$ ... $a_{ik}$))$\rightarrow$
    (APPEND
      (k ($i_1$ ... $i_k$))
      (stream (tuple (($a_{i1}$ $T_{i1}$) ... ($a_{ik}$ $T_{ik}$)))))
```

which means that it receives a stream of tuples with tuple description $((x_1\ T_1)\ \dots\ (x_n\ T_n))$ and a set of attribute names $(a_{i1}\ \dots\ a_{ik})$. The type mapping function will return not only the result type, which is a stream of tuples containing only the attributes in the argument set `(ai1 ... aik)`, i.e., a stream of tuples with description $((a_{i1}\ T_{i1})\ \dots\ (a_{ik}\ T_{ik}))$. It uses the APPEND command to append a set of attribute indexes $(i_1\ \dots\ i_k)$, and the number of attributes `k` contained in the set. The **project** operator's type mapping function is shown below[2]:

```
ListExpr ProjectTypeMap(ListExpr args)
{
  bool firstcall = true;
  int noAttrs=0, j=0;

  ListExpr first=nl->TheEmptyList();
```

1. The details on Value Mapping Fuctions for tuple streams are explained in the following section. Here, we only want to demonstrate how to use the APPENDed argument.
2. As for the **attr** operator, the more complicated type checking and error reporting are omitted.

```
ListExpr second=first, first2=first,
         attrtype=first, newAttrList=first;
ListExpr lastNewAttrList=first, lastNumberList=first,
         numberList=first, outlist=first;
string attrname="";

CHECK_COND(nl->ListLength(args) == 2,
  "Operator project expects a list of length two.");

first = nl->First(args);
second = nl->Second(args);

CHECK_COND(
  nl->ListLength(first) == 2 &&
  TypeOfRelAlgSymbol(nl->First(first)) == stream &&
  nl->ListLength(nl->Second(first)) == 2 &&
  TypeOfRelAlgSymbol(nl->First(nl->Second(first))) == tuple,
  "Operator project expects a list with structure "
  "(stream (tuple ((a1 t1)...(an tn))))."); 



CHECK_COND(
  nl->ListLength(second) > 0 &&
  !nl->IsAtom(second),
  "Operator project expects a list with attributenames "
  "(ai...aj).");

noAttrs = nl->ListLength(second);
while (!(nl->IsEmpty(second)))
{
  first2 = nl->First(second);
  second = nl->Rest(second);
  if (nl->AtomType(first2) == SymbolType)
    attrname = nl->SymbolValue(first2);
  else
  {
    ErrorReporter::ReportError(
      "Attributename in the list is not of symbol type.");
    return nl->SymbolAtom("typeerror");
  }
  j = FindAttribute(nl->Second(nl->Second(first)),
                    attrname, attrtype);
  if (j)
  {
    if (firstcall)
    {
      firstcall = false;
      newAttrList =
        nl->OneElemList(nl->TwoElemList(first2, attrtype));
      lastNewAttrList = newAttrList;
      numberList = nl->OneElemList(nl->IntAtom(j));
      lastNumberList = numberList;
    }
    else
```

```
          {
            lastNewAttrList =
              nl->Append(lastNewAttrList,
                         nl->TwoElemList(first2, attrtype));
            lastNumberList =
              nl->Append(lastNumberList, nl->IntAtom(j));
          }
        }
        else
        {
          ErrorReporter::ReportError(
            "Operator project: Attributename '" + attrname +
            "' is not a known attributename in the tuple stream.");
              return nl->SymbolAtom("typeerror");
        }
      }
      outlist =
        nl->ThreeElemList(
          nl->SymbolAtom("APPEND"),
          nl->TwoElemList(
            nl->IntAtom(noAttrs),
            numberList),
          nl->TwoElemList(
            nl->SymbolAtom("stream"),
            nl->TwoElemList(
              nl->SymbolAtom("tuple"),
              newAttrList)));
      return outlist;
    }
```

This function starts setting the `first` and `second` variables with the lists of the tuple representation and the set of attribute names desired in the projection, respectively. The number of resulting attributes is taken from the length of the attribute names list, and a variable `first2` is used to iterate inside the set of attribute names. The `FindAttribute` function is used again to retrieve the attribute index given an attribute name and a list of these attributes indexes called `numberList` is constructed. A list containing the pairs <*attribute name*, *attribute type*> is also constructed using the `newAttrL-ist` variable. The return of the **project** operator's type mapping function is a list of three elements, the first containing the `APPEND` command, the second containing the information that will be appended as arguments passed to the value mapping function, i.e., the set of attributes indexes and the size of this set, and the third containing the result type of the **project** operator, which is a stream of tuples containing the pairs in the `newAttrList` variable.

This example shows that we can pass more than one argument with the `APPEND` command. In this case, we pass the number of attributes and the list containing the attribute indexes.

2.4 Value Mapping Functions

In this section, we will take a closer look into the **feed, project** and **consume** operators' value map-ping functions. The **feed** operator's value mapping function is a good example of the stream algebra concepts, whereas with the **consume** operator we can show how tuple deletion works. Finally, with the **project** operator we show how attributes are copied.

Feed Operator Value Mapping Function

Let us now take a closer look on the value mapping function of the **feed** operator, for a review of the stream algebra concepts:

```
int
Feed(Word* args, Word& result, int message, Word& local, Supplier s)
{
  GenericRelation* r;
  GenericRelationIterator* rit;

  switch (message)
  {
    case OPEN :
      r = (GenericRelation*)args[0].addr;
      rit = r->MakeScan();

      local = SetWord(rit);
      return 0;

    case REQUEST :
      rit = (GenericRelationIterator*)local.addr;
      Tuple *t;
      if ((t = rit->GetNextTuple()) != 0)
      {
        result = SetWord(t);
        return YIELD;
      }
      else
      {
        return CANCEL;
      }

    case CLOSE :
      rit = (GenericRelationIterator*)local.addr;
      delete rit;
      return 0;
  }
  return 0;
}
```

When the message is an `OPEN`, the **feed** operator retrieves the argument relation into `r`, initializes the iterator `rit`, and stores this iterator in a special variable called `local`. This `local` variable is used to keep the state of the operator, i.e., it is a way to simulate the storage of local variables as static. Then, in the `REQUEST` message, the **feed** operator gets the iterator back from the `local` variable, retrieves the next tuple from it, and puts it into the `result` variable. If there are no more tuples available in the iterator, then `CANCEL` is returned, otherwise `YIELD` is returned. This `result` variable together with the return of the function (`YIELD` or `CANCEL`) will be used by the operator which sent the `REQUEST` message to the **feed** operator. Finally, in the `CLOSE` message, the iterator is closed and 0 is returned, which means success.

Bear in mind, that the **feed** operator receives the tuples from the `GenericRelation` object (with its reference counter = 0). Therefore, and because it only passes on the reference to each tuple, but does not maintain any further reference on it, we do not need to copy each tuple or call `IncReference()` for it. But, whoever requests a tuple from a tuple stream operator (like our **feed**) and receives it, takes over the responsible for finally calling `DeleteIfAllowed()` on it. Usually, this responsibility is passed on from stream operator to stream operator until a “tuple sink“ operator, like **consume** or **count**, is reached.

Consume Operator Value Mapping Function

Let us now take a closer look into the **consume** operator’s value mapping function:

```
int
Consume(Word* args, Word& result, int message,
        Word& local, Supplier s)
{
  Word actual;

  GenericRelation* rel = (Relation*)((qp->ResultStorage(s)).addr);
  if(rel->GetNoTuples() > 0)
  {
    rel->Clear();
  }
  qp->Open(args[0].addr);
  qp->Request(args[0].addr, actual);
  while (qp->Received(args[0].addr))
  {
    Tuple* tuple = (Tuple*)actual.addr;
    rel->AppendTuple(tuple);
    tuple->DeleteIfAllowed();
    qp->Request(args[0].addr, actual);
  }
  result = SetWord(rel);
  qp->Close(args[0].addr);
  return 0;
}
```

As mentioned before, the **consume** operator receives tuples from a stream and builds a relation containing these tuples. Actually the relation creation is not a task of the **consume** operator, but of the query processor. The query processor previously creates storage at the query tree construction time for the type constructor’s result type returned in the type mapping function. The function `ResultStorage` of the query processor returns a pointer (as a `Word`) to this created space.

There are special cases where the **consume** operator can be called inside a loop - in the **loopjoin** operator for example - and the storage is created only once by the query processor. Therefore, it is necessary to empty the relation retrieved from the query processor before proceeding.

The function then sends the stream messages `OPEN` and `REQUEST`, to retrieve the first tuple from the stream argument. If (and while) a tuple is received, it gets the tuple from the `actual` variable passed with the `REQUEST` message. The **consume** operator appends the tuple to the result relation, tries to delete it, and asks for the next one from the stream argument. After retrieving all tuples from the stream argument and appending them to the result relation, it closes the stream, and returns 0, meaning success. Note that the operator does not directly delete tuple, but calls the function `DeleteIfAllowed`.

This is, because some other operators may still hold references on certain tuples. These tuples will be finally deleted from memory, when their reference counters have been decreased to 0 and `DeleteIfAllowed` is called once more for them. If you keep references on tuples within an operator, please ensure that you call `IncReference()` exactly once for each copy, and release the reference when you do not need then any more by calling `DecReference()` and `DeleteIfAllowed()`.

The value mapping reflects the „tuple sink“ nature of the consume operator: It will consume the complete tuple stream in order to calculate its result.

Project Operator Value Mapping Function

Let us now take a closer look on the value mapping function of the **project** operator, in order to show how attributes are copied.

```
int
Project(Word* args, Word& result, int message,
        Word& local, Supplier s)
{
  switch (message)
  {
    case OPEN :
    {
      ListExpr resultType = GetTupleResultType( s );
      TupleType *tupleType = new TupleType(nl->Second(resultType));
      local.addr = tupleType;
      qp->Open(args[0].addr);
      return 0;
    }
    case REQUEST :
    {
      Word elem1, elem2;
      int noOfAttrs, index;
      Supplier son;

      qp->Request(args[0].addr, elem1);
      if (qp->Received(args[0].addr))
      {
        TupleType *tupleType = (TupleType *)local.addr;
        Tuple *t = new Tuple( tupleType );

        noOfAttrs = ((CcInt*)args[2].addr)->GetIntval();
        assert( t->GetNoAttributes() == noOfAttrs );
```

```
          for( int i = 0; i < noOfAttrs; i++)
          {
            son = qp->GetSupplier(args[3].addr, i);
            qp->Request(son, elem2);
            index = ((CcInt*)elem2.addr)->GetIntval();
            t->CopyAttribute(index-1, (Tuple*)elem1.addr, i);
          }
          ((Tuple*)elem1.addr)->DeleteIfAllowed();
          result = SetWord(t);
          return YIELD;
        }
        else return CANCEL;
      }
      case CLOSE :
      {
        ((TupleType *)local.addr)->DeleteIfAllowed();
        qp->Close(args[0].addr);
        return 0;
      }
    }
    return 0;
  }
```

The operator starts only by retrieving the result tuple type from the query processor, in the `OPEN` message. For every tuple it receives in the `REQUEST` message, a new resulting tuple is created and every attribute is copied with the help of `CopyAttribute`, and finally the tuple is (possibly) deleted with `DeleteIfAllowed`. Note that the operator does not need to know whether the tuple is fresh or solid, since this is encapsulated inside the copy and delete functions. Finally, in the `CLOSE` message, the argument stream is closed.

3 DBArray - An Abstraction to Manage Data of Widely Varying Size

3.1 Overview

The `DBArray` class provides an abstract mechanism supporting instances of data types of varying size. It implements an array, whose slot size is fixed, however, the number of slots may grow dynamically. `DBArray` is implemented as a template class and provides the following interface:

Creation/Removal	**Access**	**Restructure**	**Other**
`DBArray`	`Append`	`Resize`	`Size`
`~DBArray`	`Get`	`Clear`	
`Destroy`	`Put`	`Sort`	
	`Find`	`Restrict`	
		`TrimToSize`	

The class is derived from the class `FLOB` with the purpose of hiding the complexity of managing large objects. A detailed description about the interface can be found in the file `DBArray.h`.

A `FLOB` (Faked Large Object) decides whether an object has to be loaded or stored on disk and how it is distributed over different record files. `FLOBs` were invented to improve the storage process of tuples potentially containing large objects into records of a storage management system. Based on a threshold value, small objects are stored within tuple records, whereas large objects are swapped out into separate records; for details refer to [DG98]. Hence, the design of the `DBArray`, `FLOB` and the tuple representation within the relational algebra was coordinated to support a simple integration of new data types with varying size into the relational data model with a general mechanism for achieving persistence.

3.2 Example

As an example for the usage of class `DBArray`, the implementation of the `PolygonAlgebra` [Poly02] can be studied. It uses a class `Polygon` having a private member `vertices` of type `DBArray`:

```
struct Vertex {
  ...
  int x;
  int y;
};
```

```
class Polygon {
  ...
  void Append( const Vertex& v );
  void Complete(); // set state = complete
  void Destroy();  // destroy the DBArray vertices
  ...
  private:
    int noVertices;
    int maxVertices;
    DBArray<Vertex> vertices;
    PolygonState state;
};

void Polygon::Append( Vertex& v )
{
  assert( state == partial );
  vertices.Append( v );
}

void Polygon::Destroy()
{
  assert( state == complete );
  vertices.Destroy();
}
...
```

These are only some incomplete code fragments, but they demonstrate the usage of `DBArray`. Since `DBArray` is a template class it is type safe and instantiated with parameter type `Vertex`, a simple struct representing (x, y) coordinates. The assertion `state == partial` is just a helpful instrument to locate errors during the development, a polygon may be in state `partial` or `complete`.

The algebra's `In`-function expects to receive a list of structure ((x_1 y_1) ... (x_n y_n)) and will create a new instance of class `Polygon.` Therefore it iterate over the nested list, which is passed to it by an update, let or restore command. During the iteration process the `Polygon::Append` function is used to fill the `DBArray` member `vertices` with `Vertex`-values. This is demonstrated in the example below.

```
Word
InPolygon( const ListExpr typeInfo, const ListExpr instance,
           const int errorPos, ListExpr& errorInfo, bool& correct )
{
  Polygon* polygon = new Polygon( 0 );

  ListExpr first = nl->Empty();
  ListExpr rest = instance;
  while( !nl->IsEmpty( rest ) )
  {
    first = nl->First( rest );
    rest = nl->Rest( rest );
```

```
        if( nl->ListLength( first ) == 2 &&
            nl->IsAtom( nl->First( first ) ) &&
              nl->AtomType( nl->First( first ) ) == IntType &&
            nl->IsAtom( nl->Second( first ) ) &&
              nl->AtomType( nl->Second( first ) ) == IntType )
        {
          Vertex v( nl->IntValue( nl->First( first ) ),
            nl->IntValue( nl->Second( first ) ) );
          polygon->Append( v );
        }
        else
        {
          correct = false;
          return SetWord( Address(0) );
        }
      }
      polygon->Complete();
      correct = true;
      return SetWord( polygon );
    }
```

The function `Clear` deletes the current content of a FLOB, but the instance can be reused. Usage of `Destroy` completely removes all underlying records. The latter is typically done in a data type's delete function.

```
    void DeletePolygon(Word& w)
    {
      Polygon* polygon = (Polygon*)w.addr;

      polygon->Destroy();
      delete polygon;
    }
```

Here `polygon->Destroy` calls `vertices->Destroy`. The `get` and `put` methods which are not used in the examples above simply return or set the value for a given array index, which are numbered from 0 to `Size()` – 1. If the data type of the array is ordered, one could provide a comparison function. Then the `find` method can be used to do a binary search on the persistent representation, and the `sort` method will re-organize the array in sorted order.

3.3 Accessing FLOBs Directly

The `DBArray` class provides a nice and clean abstraction mechanism to support arrays of varying size, containing elements of fixed size. But, if the data type is not well organized in arrays, one can use directly an object of class `FLOB`. This would be important for a type constructor which needs to store long and variable sized texts, for example. Texts can be viewed as an array of characters, but it would be better to store them directly into a FLOB to avoid lots of calls to `Put` and `Get` functions. The `FLOB` class provides an interface similar to the `DBArray` class, but the `Put` and `Get` functions are different. Their interface is shown below:

```
void Put( size_t offset, size_t length, const void *source );
void Get( size_t offset, const char **target );
```

They read (write) into `source` (`target`), from an `offset` until `length` in bytes. Let us see an example of the `BinaryFileAlgebra`, which provides the type constructor *binfile*. This algebra stores in a SECONDO object the bytes sequence of a file. The storage will be provided by a member variable of type FLOB in the `BinaryFile` class sketched below:

```
#include “Base64.h“

class BinaryFile : public StandardAttribute
{
  public:
  ...

    void Encode( string& textBytes );
    void Decode( string& textBytes );
    bool SaveToFile( char *fileName );

  private:

    FLOB binData;
    bool canDelete;
};
```

Since SECONDO needs the ability of textual representation of data, binary data must be encoded in printable characters. Base 64 [BF93] is a widely used encoding format for this purpose. The `Encode` and `Decode` functions convert binary to textual representation and vice versa. The examples below demonstrate access to a `FLOB` objects, i.e., how to use the `Put` and `Get` functions.

```
void BinaryFile::Encode( string& textBytes )
{
  Base64 b;
  const char *bytes;
  binData.Get( 0, &bytes );
  b.encode( bytes, binData.Size(), textBytes );
}
```

Both functions use the `Base64` class that provides functions for encoding binary data and decoding Base 64 data. The `Encode` function first allocates some bytes for reading and then calls the function `Get` to read in the bytes from the `FLOB`. These bytes are in binary format and they are passed to the `encode` function of the `Base64` class creating a string with characters of the Base 64 alphabet.

```
void BinaryFile::Decode( string& textBytes )
{
  Base64 b;
  int sizeDecoded = b.sizeDecoded( textBytes.size() );
  char *bytes = (char *)malloc( sizeDecoded );

  int result = b.decode( textBytes, bytes );

  assert( result <= sizeDecoded );

  binData.Resize( result );
  binData.Put( 0, result, bytes );
  free( bytes );
}
```

The `Decode` function does the contrary, it decodes a Base64 text string into a block of binary bytes and stores it in the FLOB using the function `Put`. Note that before copying data into the FLOB its capacity must be adjusted with the `Resize` function.

3.4 Interaction with the Relational Algebra

The requirements for data types to be used as relation attributes are discussed in detail in the next section, but when DBArrays or FLOBs are used, two functions inherited from class `Standardattribute` must be implemented. An example of these functions for the *binfile* type constructor is shown below.

```
int BinaryFile::NumOfFLOBs()
{
  return 1;
}

FLOB *BinaryFile::GetFLOB(const int i)
{
  assert( i >= 0 && i < NumOfFLOBs() );
  return &binData;
}
```

As data types can have more than one FLOB, it is necessary to implement a mechanism to have access to them. In this way, the function `NumOfFLOBs` must return how many FLOBs the data type has, and the function `GetFLOB` must return the one identified by the requested index `i`.

4 Kind DATA: Attribute Types for Relations

When data types shall be used as attributes of a relation object, every C++ class implementing such a type has to implement a special set of functions, which are used by operators of the relational algebra. These functions include methods to compare object values, load and save them to disk, and print them. These functions have been encapsulated into a C++ class named `Attribute`. Any attribute type must inherit from this class and additionally register for the kind `DATA`.

The subclass `StandardAttribute` adds functions to compute hashvalues and copy values. Additional functions needed to use indexes on attribute types (as e.g. implemented by the BTree Algebra [BTree02]) have been encapsulated into another class called `IndexableStandardAttribute`. Datatypes inheriting from this class may be register for kind `INDEXABLE`. The class hierarchy is:

`Attribute` → `StandardAttribute` → `IndexableStandardAttribute`.

The functions of each class are explained in more detail in the tables below:

Class `Attribute` (1)	
Function	**Description**
`NumOfFLOBs`	Returns the number of `FLOBs` (`DBArrays`) used in the class. Default implementation returns zero.
`GetFLOB`	Returns a pointer to a `FLOB`[1] object.
`Open`	May be used as standard method for opening a persistent object. Can be useful in the implementation of type constructors.
`Save`	The complementary function to `Open`. Makes an object persistent.
`Initialize`	Used only by JNI-Algebras, which must provide a pointer to an object in the Java Virtual Machine.
`Finalize`	Used only by JNI-Algebras.
`Print`	Should be implemented to print out useful information on an `ostream`, mainly for debugging. But the function is also used in several generic output operators (like the **printstream** operator).
`operator<<`	Should be implemented to write an instance of this class into an `ostream` object.

The `open` and `save` functions should not be overwritten. They define useful standard implementations for creating or restoring persistent versions of an object with `FLOB`[1] members, hiding the complexity of reading and writing data from memory into records. If the data type does not contain members of type `FLOB`, the functions `NumOfFLOBs` and `GetFLOB` do not need to be refined. The functions `Initialize` and `Finalize` are only needed for Algebras written in Java and employing the JNI (Java Native Interface) for integration into SECONDO. Note in contrast to the following functions, the ones presented above are not pure virtual functions, hence they have a default implementation. Thus

1. FLOBs are explained in the Chapter on DBArrays.

when you need to overwrite one of the above functions, be careful to use the correct name and signature. Otherwise at runtime the default implementation will be used. For all other functions the compiler will help you, since pure virtual functions need to be implemented in the child class.

Class `Attribute` (2)	
Function	**Description**
`SizeOf`	Returns the memory size needed to store an instance of the class. This information is needed for calculating the size of a tuple (e.g. to calculate how many tuples fit into a buffer). A typical implementation is to `return sizeof(`<*class name*>`)`.
`Compare`	Defines a total order. This is, for example, used by the **sortby** operator of the relational algebra. Returns an integer value *V*. If the the current instance (`this`) is „smaller“ than the function's argument, *V* should be negative (usually -1). If both objects are „equal“, *V* becomes 0, if the argument is smaller than `this`, *V* becomes positive (usually +1). Note, that there is a convention on how to treat undefined values (see file include/Attribute.h).
`Adjacent`	Decides whether the current instance (`this`) is adjacent to an instance of the same type passed as argument. Examples: 2 and 3 are adjacent integers; “abc“ and “abd“ are adjacent strings. This is used by operations merging intervals.
`Clone`	This is needed by several operators of the relational algebra. Clone will depth copy the object. The result is a completely new object, with its own storage and reference counter.
`IsDefined`	This function is used to indicate if the object represents a valid value or not; for example, a division by zero results into an integer object with status “not defined”.
`SetDefined`	Used to set the objects “defined” status.

Any datatype, that inherits from `Attribute` may be registered with kind `DATA` and used as an attribute within a relation.

Class `StandardAttribute`	
Function	**Description**
`HashValue`	This is a function which maps values to integers used by hash-functions. The **hash-join** operator of the relational algebra needs this information.
`CopyFrom`	Works like a copy constructor. It copies the value of a referenced attribute into this object. This is used by several operators of the relational algebra. Remember, any changes will be transparent for all references created using `Attribute::Copy()`, but not those, that have been created using `Attribute::Clone()`. As only a limited number of references is possible for each object, sometimes, an object will be cloned when you intend to copy it. As a result, you should never rely on transparent changes to references to an attribute.

Class `IndexableStandardAttribute`	
Function	**Description**
`WriteTo`	Converts the value of an object into an uniqe (order preserving) key value.
`SizeOfChars`	Length of the key value.
`ReadFrom`	Restores an object value from its key.

Note: The data structure of any data type intended to be suitable as an attribute type is strictly required to have the following structure: It consists of a single block of memory which may contain a fixed number of FLOBs or DBArrays. Hence all attributes of such a class must themselves have fixed size (so the compiler embeds their memory blocks into the block of the class instance). In other words, the only types that can be used as class members except for FLOBs and DBArrays are scalar types like `int`, `float`, `char[]`, etc. It is not possible to use pointer structures or classes from libraries, e.g. a binary tree or a string, as type of member variables. If dynamic data structures like trees are needed they have to be embedded into DBArrays, with array indices serving as pointers.

The reason for this restriction is that the mechanism for building tuple data structures with the automatic placement of FLOBs is based on this assumption. In addition, all “pointers” are automatically stable regardless of how values are placed in memory.

At first glance, an algebra implementor may not understand the need for all of these functions. However, the relational algebra makes use of all of them. So, the implementation of these functions is essential.

Refer to the `DateTimeAlgebra` [Date04] as an example for an algebra whose types are made available as attribute types in relations. The example below describes some parts of the C++ class `DateTime`.

```
class DateTime : public IndexableStandardAttribute  {

...

/*

The next functions are needed for the DateTime class to act as
an attribute of a relation.

*/
     int        Compare(Attribute* arg);
     bool       Adjacent(Attribute*);
     int        Sizeof();
     bool       IsDefined() const;
     void       SetDefined( bool defined );
     size_t     HashValue();
     void       CopyFrom(StandardAttribute* arg);
     DateTime*  Clone();
     void       WriteTo( char *dest ) const;
     void       ReadFrom( const char *src );
     SmiSize    SizeOfChars() const;

...
}
```

In the implementation of its `open` and `save` function one can see how the standard `open` and `save` methods of class `Attribute` are reused. Of course, class `DateTime` has no FLOB members but it would work the same way if it had some of them.

```
bool OpenDateTime( SmiRecord& valueRecord,
                   size_t& offset,
                   const ListExpr typeInfo,
                   Word& value ){
 DateTime *dt =
      (DateTime*)Attribute::Open( valueRecord, offset, typeInfo );
 value = SetWord( dt );
 return true;
}

bool SaveDateTime( SmiRecord& valueRecord,
                   size_t& offset,
                   const ListExpr typeInfo,
                   Word& value ){
 DateTime *dt = (DateTime *)value.addr;
 Attribute::Save( valueRecord, offset, typeInfo, dt );
 return true;
}
```

5 SMI - The Storage Management Interface

The SMI is a general interface for reading and writing data to disk. Although this sounds simple, implementing concepts for locking, buffering, and transaction management is a highly complex task. Therefore, SECONDO does not have its own concepts for this purpose but uses already existing database libraries. Hence, the SMI provides a collection of classes which are used as a general interface within SECONDO to access the API of other database libraries. Currently, code for two implementations of the SMI is present; the first one is based on the Open Source project Berkeley-DB and the second one uses the Oracle-DBMS. The latest SECONDO sources can only be compiled with the Berkeley-DB version of the SMI, since only this version is maintained in the further development. However, the interface was designed to separate the code of the SECONDO core system and the algebra modules from interfaces of libraries which support storage management facilities.

5.1 Retrieving and Updating Records

In this introduction an overview about the SMI classes and their interdependencies is presented. The SMI uses the two concepts *records* and *files*. In a record a sequence of bytes can be stored. It has a unique ID and each file can hold many records. Basically, the SMI offers operations on files and records. An overview about all SMI classes is presented below:

Classes	**Description**
`SmiEnvironment`	This class provides static member functions for the startup and initialization of the storage management environment.
`SmiFile`	Base class, a container for records.
`SmiRecordFile`	Records are selected by their IDs.
`SmiKeyedFile`	Records are accessed by a key value.
`SmiFileIterator`	Base class, supports scanning of files.
`SmiRecordFileIterator`	Iterator for files with access via record-IDs.
`SmiKeyedFileIterator`	Iterator for files with access via keys.
`SmiRecord`	A handle for processing records. It can be used to access all or partial data of records.
`SmiKey`	A generalization for different types of keys, such as integer, string, etc.
`PrefetchingIterator`	Efficient read-only iteration reducing I/O activity.

More technical information about functions and their signatures is described in the file `SecondoSMI.h`. The following code examples demonstrate the usage of the `SmiRecordFile` and `SmiRecord` classes.

```
bool makefixed = true;
string filename = (makeFixed) ? "testfile_fix" : "testfile_var";
SmiSize reclen = (makeFixed) ? 20 : 0;
SmiRecordFile rf( makeFixed, reclen );
if ( rf.Open( filename ) )
{
  cout << "RecordFile successfully created/opened: "
      << rf.GetFileId() << endl;
  cout << "RecordFile name   =" << rf.GetName() << endl;
  cout << "RecordFile context=" << rf.GetContext() << endl;
  cout << "(Returncodes: 1 = ok, 0 = error )" << endl;
  SmiRecord r;
  SmiRecordId rid, rid1, rid2;
  rf.AppendRecord( rid1, r );
  r.Write( "Emilio", 7 );
  rf.AppendRecord( rid2, r );
  r.Write( "Juan", 5 );
  char buffer[30];
  rf.SelectRecord( rid1, r, SmiFile::Update );
  r.Read( buffer, 20 );
  cout << "buffer = " << buffer << endl;
  cout << "Write " << r.Write( " Carlos ", 8, 4 );
  cout << "Read " << r.Read( buffer, 20 ) << endl;
  cout << "buffer = " << buffer << endl;
  r.Truncate( 3 );
  int len = r.Read( buffer, 20 );
  cout << "Read " << len << endl;
  buffer[len] = '\0';
  cout << "buffer = " << buffer << endl;
}
```

A `RecordFile` object can either contain records of fixed length or of variable length. This is controlled by a boolean parameter passed to the constructor. Note that a new record first has to be appended to the `RecordFile`; afterwards a value can be assigned using the `write` operation. The `write` and `read` operations on records may have up to three parameters. The first parameter defines a storage buffer for the data which is transferred into memory or to the record on disk. The second parameter holds the number of bytes to transfer. Finally, the (optional) third parameter defines an offset relative to the starting position of the record on disk.

5.2 The SMI Environment

At the startup process of SECONDO, an instance of class `SmiEnvironment` is created. Then if, during the work with SECONDO, a database is opened, `SmiFile` objects can be used. Whenever new objects in a SECONDO database are created or destroyed using algebra operations, information about the involved `SmiFiles` and the objects types and values has to be maintained in the database catalog. Hence the catalog does many SMI-Operations. Moreover, the query processor opens, closes, creates and deletes objects.

Most data types have quite simple representations and can be stored in files using default persistence mechanisms, but more complex data types, e.g. relations, have to organize data in their own files and thus use SMI operations.

In order to get familiar with the SMI it is recommended to create small test programs independent from the main SECONDO system. A framework for the startup and shutdown of the storage manager is shown below:

```
SmiError rc;
bool ok;

string configFile = "SecondoConfig.ini"
rc = SmiEnvironment::StartUp( SmiEnvironment::MultiUser,
                              configFile, cerr );
cout << "StartUp rc=" << rc << endl;
if ( rc == 1 )
{
    string dbname="test";
    ok = SmiEnvironment::CreateDatabase( dbname );
    if ( ok ) {
      cout << "CreateDatabase ok." << endl;
    } else {
      cout << "CreateDatabase failed." << endl;
    }
    if ( ok = SmiEnvironment::OpenDatabase( dbname ) ) {
      cout << "OpenDatabase ok." << endl;
    } else {
      cout << "OpenDatabase failed." << endl;
    }
  }
  if ( ok )
  {
    cout << "Begin Transaction: "
   << SmiEnvironment::BeginTransaction() << endl;

    /* SMI code */

    cout << "Commit: "
   << SmiEnvironment::CommitTransaction() << endl;

    if ( SmiEnvironment::CloseDatabase() ) {
      cout << "CloseDatabase ok." << endl;
    } else {
      cout << "CloseDatabase failed." << endl;
    }
    if ( SmiEnvironment::EraseDatabase( dbname ) ) {
      cout << "EraseDatabase ok." << endl;
    } else {
      cout << "EraseDatabase failed." << endl;
    }
  }
}
rc = SmiEnvironment::ShutDown();
cout << "ShutDown rc=" << rc << endl;
```

For building an executable program which offers a runtime environment for working with `SmiFiles`, this code has to be linked together with the SECONDO SMI library and Berkeley-DB library. Please refer to the file `./Tests/makefile` which contains rules for linking against these libraries.

6 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 6.1. We then explain in Section 6.2 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.

6.1 How the Optimizer Works

6.1.1 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
pdview optimizer
```

After the second call of `pdview` also the table of contents has been generated and included. 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].

6.1.2 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 5.

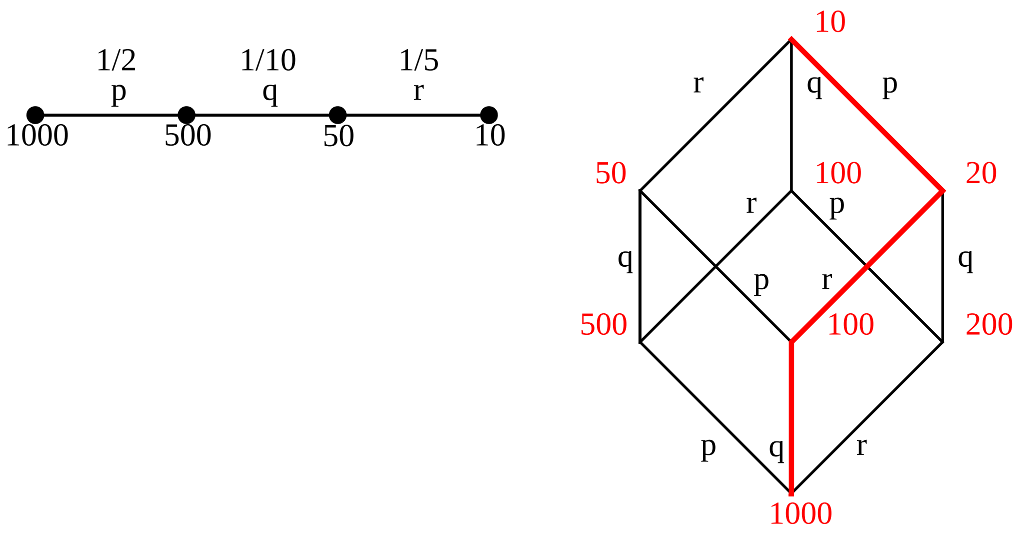


Figure 5: A predicate order graph for 3 predicates

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 6 shows an example of a POG annotated with evaluation methods.

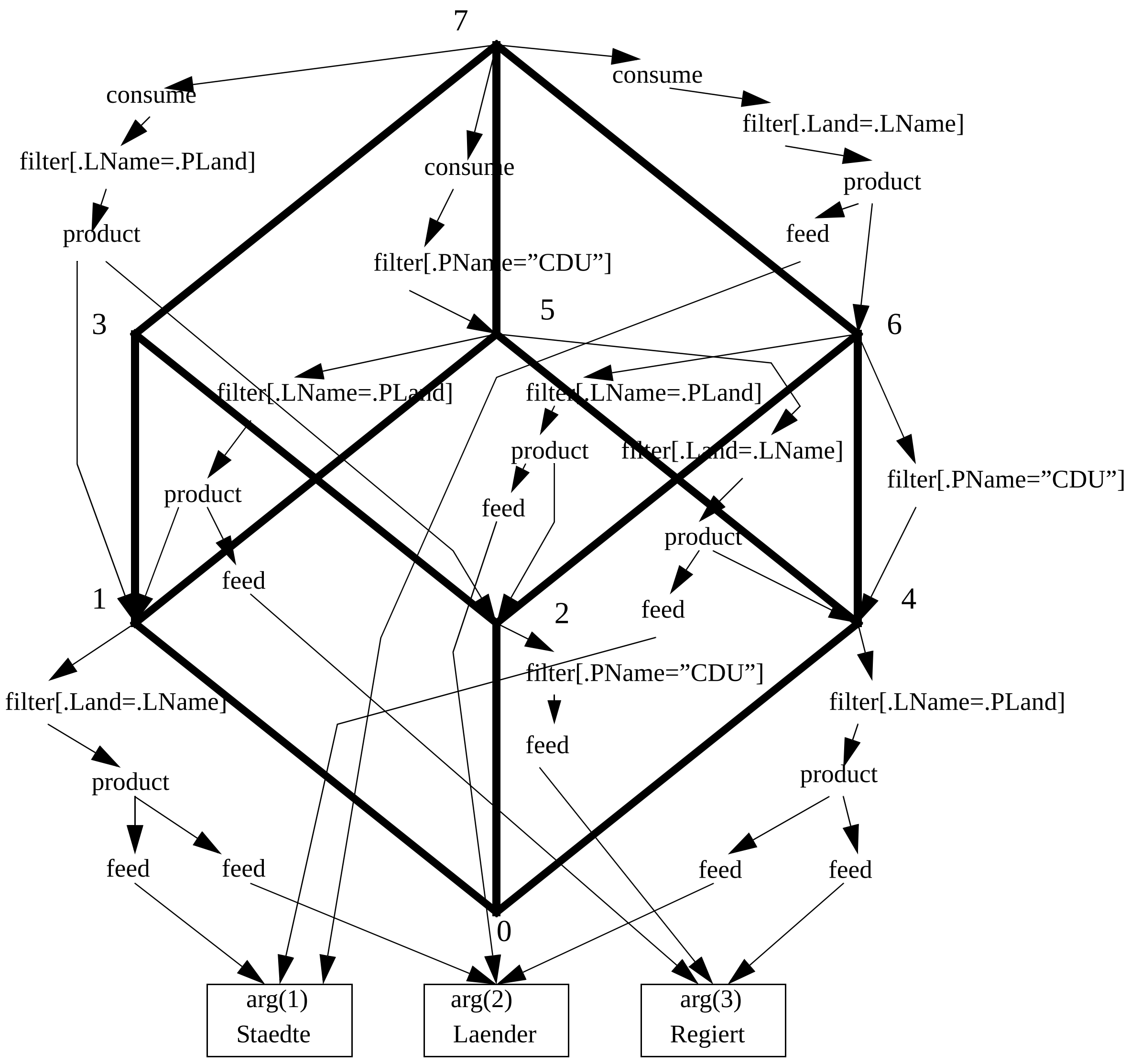


Figure 6: A POG annotated with evaluation methods

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:

```
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).

6.2 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.

1. In the standard 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*.

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

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

6.2.1 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` (Section 1.1.4). 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.

6.2.2 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 `adjacent` is an infix operator, we write:

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

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

6.2.3 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 <, `adjacent`, `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`, `adjacent`, and `translate` with 1, 2, and 3 arguments, respectively, are automatically written as:

```
length(x), x adjacent 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.

6.2.4 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 6.1.2. Before we can formulate translation rules, we need to understand 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)) :-
  isStarQuery,
  argument(N, rel(Name, *, Case)), !.

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

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

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

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 first rule above says that a term of the form `res(N)` is not changed by translation.

For `arg(N)`, which is a stored relation, we check whether the query is of the form `select * from ...` In this case, in the analysis of the query a dynamic predicate `isStarQuery` was asserted. For such a query we look up the relation name and then apply a `feed` and possibly an additional `rename` to convert it into a stream of tuples.

If the query has a projection list of attributes, then in the analysis of the query `isStarQuery` was not asserted and a list of attributes that are needed from this relation in query processing was computed. This list can be retrieved via the predicate `usedAttrList`. So in addition to `feed` and possibly `rename`, a `project` operator is applied to the base relation.

PROLOG is a wonderful environment for testing and debugging. We execute `example5` once more after asserting `isStarQuery` (because translations are done together with the construction of the predicate order graph, and they depend on `isStarQuery`). We can then directly see how things translate.

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

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

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

Yes
20 ?-
```

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:

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

X = Staedte  feed {s}

Yes
21 ?-
```

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 second rule covers the case that the predicate `pr` is a join predicate. Such `select` edges on join predicates are constructed if the query contains two or more join predicates on the same pair of relations. That is, for this `select` edge, the two relations have already been joined on previous edges.

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

A selection can also be translated into an `exactmatch` operation on a B-tree under certain conditions. This is specified by the following four rules (we here treat only the simpler case that no projection is needed after index access, i.e., `isStarQuery` holds):

```
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).
```

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 which is a B-tree. Finally, the fourth rule is only a variant of the third for the case that the relation has a variable assigned in the query and needs to be renamed.

Translating Join Predicates

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

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

Every join can be translated into a so-called `symmjoin`.[1]

```
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), 99997)   :-
  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).[2] 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 99997 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. The symmjoin considers all pairs of tuples from its two argument streams, evaluates the predicate for each pair and puts matching pairs of tuples into the result stream. Hence it works pretty much like a nested loop join, except that it reads tuples from its two argument streams in an alternating way, joining such a tuple with a buffer from the other stream, hence, in a symmetric manner.
2. 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:

```
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 ?-
```

6.2.5 Writing Cost Functions

The next step in the optimization algorithm described in Section 6.1.2, 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` (`delete-Sizes` to remove them again) and see the result by `writeSizes`. Continuing with `example5`, we have: [1]

```
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 ?-
```

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:

```
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
```

1. Note that the numbers occurring in these examples may differ on your system, since selectivities are determined by sampling, and samples may be different.

```
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 ?-
```

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 = symmjoin(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(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
```

For the `symmjoin` operator, the size of the result is given by the size of the Cartesian product times the selectivity (as for any join operator, in fact). The cost is the sum of the costs of producing the arguments plus some cost proportional to the total number of pairs of tuples considered plus some cost proportional to the size of the result. The latter two terms are weighted by the two constants for `symmjoin`, here retrieved in `A` and `B`.

We also consider the cost for a simple filter operator application:

```
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.[1] 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.

Another important aspect that has been neglected in these simple cost formulas is the use of buffers in some operators, especially the join operators. The cost functions change dramatically when buffers overflow and buffer contents need to be saved to disk.

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.

1. This holds for the standard version of the optimizer considered in this manual. More sophisticated cost estimation exists in other versions of the optimizer. However, it is also more difficult to explain and extend.

7 Integrating New Types into User Interfaces

7.1 Introduction

SECONDO can be extended with new algebras. Therefore, a user interface should be able to integrate new display functions for the new data types defined in the newly introduced algebras. In the following sections we show how to extend Javagui to be able to display the new data types. Afterwards, the appropriate extension for `SecondoTTY` is described.

7.2 Extending the Javagui

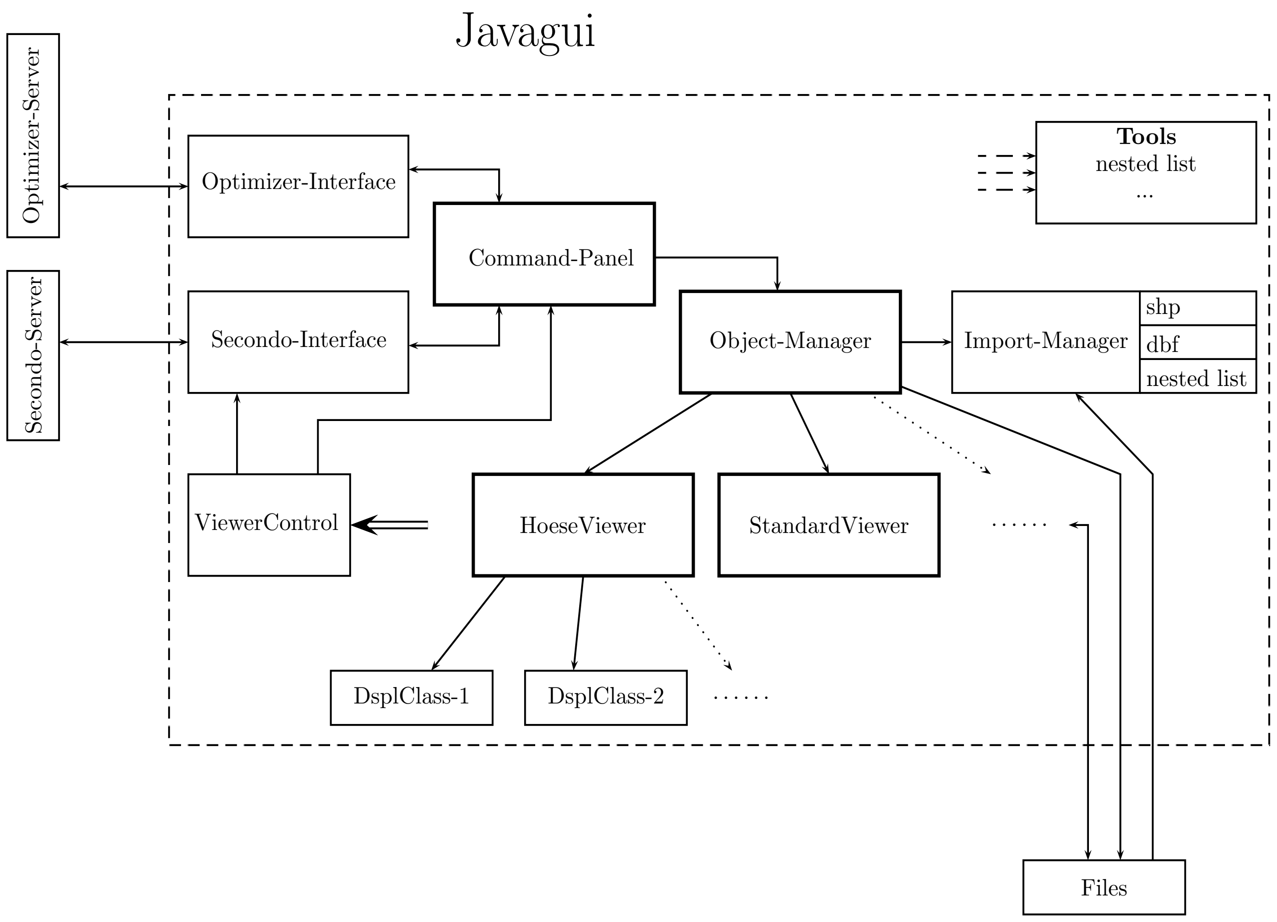


Figure 7 Overview of Javagui

Data are exchanged between SECONDO and Javagui via TCP. In general, a main part of these data are SECONDO objects, which are encoded in nested list format. Nested lists of SECONDO objects are sent to the viewer in the format `(<type> <value>)`.

There are two ways to integrate a new data type into Javagui: to write a new viewer or to extend the HoeseViewer. Both ways have advantages and also disadvantages. When extending the HoeseViewer, the developer only needs to implement how to draw or print the new object. The functionalities provided by the HoeseViewer (zoom etc.) can be used without writing additional code. Unfortunately, the HoeseViewer cannot display all kinds of SECONDO objects. For instance, the developer cannot define the drawing order for multiple objects, and it is not possible to display three-dimensional objects. Therefore, in a lot of cases it is more reasonable to write a new viewer. Since the HoeseViewer uses only one display function for a SECONDO object, it is not possible to have alternative representations for objects. By writing a new viewer, the developer can decide how, when and where an object is drawn.

The complete interface documentation of all classes can be obtained by entering `make doc` in the Javagui directory. Javagui must have been compiled before this command is executed because some classes are generated in the Javagui make process. This creates a new directory `doc` containing the result of the `javadoc` tool.

7.2.1 Writing a New Viewer

Every viewer must be part of the `viewer` package. If more than one class is needed to implement a viewer, only the main class should be in the `viewer` package. All other classes should be placed in a new package. Although Javagui consists of a lot of Java classes, the developer of a new viewer only needs to know seven of them, namely `SecondoViewer, SecondoObject`, `ListExpr`, `MenuVector`, `ID, IDManager, and Reporter`. Furthermore, the developer should know the standard Java classes (especially the Java Swing components). The following sections describe these seven classes. Before a new viewer is integrated, the `makefile` in the `viewer` directory should be changed. The `makefile` in the `Javagui/viewer` directory has to be extended by adding the name of the viewer to the variable `VIEWER_CLASSES` and by adding the names of the packages to the `VIEWER_DIRS` variable.

The ID Class

This class is used to distinguish different SECONDO objects, even though these objects may have the same value. For instance in a viewer, this class can be used to check whether an object is already displayed. To do this, the `equals` method of the class is used.

The IDManager Class

A viewer can create new SECONDO objects. For instance, the HoeseViewer can load object values from files and create SECONDO objects from them. Each SECONDO object must have an own `id.` To get an unused `id`, use the `getNextID` method of this class.

The MenuVector Class

Each viewer can extend the main menu of Javagui with its own entries. The `MenuVector` class is used for this purpose. Normally, a viewer developer should only use the `addMenu(JMenu)` method to extend the menu. The created `MenuVector` is the return value of the `getMenuVector` method (see Table 1).

The ListExpr Class

All objects resulting from requests to SECONDO are in nested list format. The `ListExpr` class is the Java representation of such lists. A viewer should extract the desired information from the nested list. To display a SECONDO object, the viewer needs to derive the desired data from a nested list and create Java objects from it.

The SecondoObject Class

An instance of the `SecondoObject` class consists of an `id`, a `name`, a `value`, and some methods to access these data. The `id` is used for internal identification, whereas the `name` identifies the object for the user of Javagui. The `value` is the nested list representation of the object.

The Reporter Class

This class should be used to inform the user about errors and success of operations. All messages should be shown using this class. This concerns both, outputs on the console and pop up windows. Such windows are switched off automatically when Javagui runs in one of its test modes (see SECONDO's User Manual).

The SecondoViewer Class

Every viewer is a subclass of `SecondoViewer`. All abstract methods have to be implemented. Table 1 describes all important methods in this class.

`String`
`getName()`
	Gets the name of this viewer. This name is displayed in the `Viewers` menu of Javagui.

`boolean`
`addObject(SecondoObject o)`
	Adds a SECONDO object. In this method the viewer must analyse the value of `o` (using `o`'s `toListExpr` method) and display it.

`void`
`removeObject (SecondoObject o)`
	Removes `o` from this viewer.

Table 1: Methods of SecondoViewer

`void`
`  removeAll()`
	Removes all objects from this viewer.

`boolean`
`  canDisplay(SecondoObject o)`
	Normally, a viewer cannot display all objects resulting from requests to SECONDO. This method returns `true` if this viewer is able to display the given object.

`boolean`
`  isDisplayed(SecondoObject o)`
	Returns `true` if `o` is contained in this viewer. Note, that the result of this function can be `true` while `o` is not visible, e.g. in the `StandardViewer` only the currently selected of possibly many objects is visible.

`boolean`
`  selectObject(SecondoObject O)`
	Selects `o` in this viewer. How an object is selected can be specified by the viewer implementor.

`MenuVector`
`  getMenuVector()`
	Returns the menu extension for this viewer. If no menu extension exists, `null` is returned.

`double`
`  getDisplayQuality(SecondoObject SO)`
	This method is not abstract. This means, that a viewer implementor may but does not need to implement this method. The result of this method must be in the range [0,1]. 0 means, that the viewer can’t display this object. 1 means, that this is the best viewer to display the given object. This feature is used when a viewer is selected for displaying an object (see SECONDO User Manual).

`void`
`  enableTestmode(boolean on)`
	If the argument is true, all automatical user interaction has to be disabled.

Table 1: Methods of SecondoViewer

Example

In this example a viewer is described, which can display results of inquiries to SECONDO.

List Format

All results of such inquiries to SECONDO are nested lists with two elements. The first element is a symbol atom containing the value `inquiry`. The second element is again a list with two elements. The first element of this list is a symbol atom describing the kind of the inquiry. Possible values are `databases`, `types`, `objects`, `constructors`, `operators`, `algebras` or `algebra`. The structure of the second element depends on this value.

The lists for `databases` and `algebras` have the same structure. They consist of symbol atoms describing the names of databases and algebras. Example lists are:

- `(inquiry (databases (GEO OPT EUROPE)))`

- `(inquiry (algebras (StandardAlgebra RelationAlgebra SpatialAlgebra)))`

The list for `constructors` consists of lists of three elements. Each of these lists consists of a symbol describing the constructor name, a list with property names and a list with property values.

`(inquiry (constructors ( <constructor`$_1$`>...<constructor`$_n$`>)))` where
`constructor`$_i$` := ( name <property names> <property values>)`

The lists for property names and for property values should have the same length. The element at position `x` in the value list is the value for the name at the same position in the list of names. All elements in these lists are atomic (mostly string or text atoms).

The list structure for `operators` is the same as for `constructors`. Only the keyword is different.

The value list for `algebra` has two elements. The first element is a symbol atom describing the name of the requested algebra. The second element is a list of two elements describing the type constructors and the operators of this algebra. The format of these lists is the same as in the lists above.

The structure for `types` is the following:

- `(inquiry (types (TYPES <type`$_1$`> ... <type`$_n$`>)))`

Each `<type`$_i$`>` is in format:

- `(TYPE <name> <value>)`

where `TYPE` is a keyword, `<name>` is a symbol and `<value>` describes the type. Since the possible types depend on the currently used algebras, `<value>` has no fixed structure.

The lists for objects are similiar to the lists for types:

- `(inquiry (objects (OBJECTS <object`$_1$`> ... <object`$_n$`>)))`

where `<object`$_i$`>` is a list built as follows:

- `(OBJECT <name> <typename> <type>)`

where `OBJECT` is a keyword, `<name>` a symbol containing the name of the object, `<typename>` is a list containing the user defined name of the type as a symbol or an empty list if no name exists, and `<type>` is a list describing the type.

Building the Viewer

The name for this viewer is “InquiryViewer”. All objects are formatted with help of `html` code. The layout of this viewer is very simple. At the top, there is an option bar for selecting an object. Located at the bottom, there is a field and a button supporting searching within the text. In the remaining area the formatted textual representation of the selected object is shown. The package of this viewer is `viewer`. Since graphical elements are used, a few imports are needed. In addition, the access to `ListExpr` and `SecondoObject` is required. (See the source code in Appendix A.)

The viewer has three components to manage SecondoObjects: a ComboBox containing the names of SecondoObjects, a Vector containing the SecondoObjects and another Vector containing the html code for SecondoObjects. The connection between the different representations of an object is given by the indices of the objects in these containers. To extend the main menu of Javagui the viewer has a MenuVector MV. For displaying objects a JEditorPane is used.

The getName method just returns the string “InquiryViewer”. The isDisplayed method checks whether the given object is contained in the vector SecondoObjects or not. The method removeObject removes a given object from all of its representations. All representations can be emptied with the removeAll method. After an object is selected from a combobox, the position of this object in the SecondoObjects vector is determined, and then the object can be displayed. The MenuVector built in the constructor is returned using the getMenuVector method. The code for these methods and the code for html formatting is given in Appendix A. The remaining methods are described here in detail.

The canDisplay method checks whether this viewer can display a given SecondoObject. This is done by checking the list format described above. The first element of the list must be a symbol atom with the content “inquiry”. The second element has to be a list of two elements. The first element of this list must be again a symbol atom. The value of this symbol must be an element of the set {“databases”, “constructors”, “operators”, “algebras”, “algebra”, “types”, “objects”}.

```
public boolean canDisplay(SecondoObject o){
  ListExpr LE = o.toListExpr(); // get the nested list of o
  if(LE.listLength()!=2) // the length must be two
    return false;
  // the first element must be an symbol atom with content "inquiry"
  if(LE.first().atomType()!=ListExpr.SYMBOL_ATOM ||
      !LE.first().symbolValue().equals("inquiry"))
    return false;
  ListExpr VL = LE.second();
  // the length of the second element must again be two
  if(VL.listLength()!=2)
    return false;
  ListExpr SubTypeList = VL.first();
  // the first element of this list must be a symbol atom
  if(SubTypeList.atomType()!=ListExpr.SYMBOL_ATOM)
    return false;
  String SubType = SubTypeList.symbolValue();
  // check for supported "sub types"
  // the used constants just contain the appropriate String
  if(SubType.equals(DATABASES) || SubType.equals(CONSTRUCTORS) ||
      SubType.equals(OPERATORS) || SubType.equals(ALGEBRA) ||
      SubType.equals(ALGEBRAS) || SubType.equals(OBJECTS) ||
      SubType.equals(TYPES))
    return true;
  return false;
}
```

Because this viewer is very good for displaying objects resulting from inquiries, the `getDisplayQuality` method is overwritten.

```
public double getDisplayQuality(SecondoObject SO){
    if(canDisplay(SO))
       return 0.9;
    else
       return 0;
 }
```

The constructor of this class builds the graphical components and initializes the objects managing the SECONDO objects. Besides, the menu is extended.

```
public InquiryViewer(){
  // add the components
  setLayout(new BorderLayout());
  add(BorderLayout.NORTH,ComboBox);
  add(BorderLayout.CENTER,ScrollPane);
  HTMLArea.setContentType("text/html");
  HTMLArea.setEditable(false);
  ScrollPane.setViewportView(HTMLArea);

  // build the panel for search within the text
  JPanel BottomPanel = new JPanel();
  BottomPanel.add(CaseSensitive);
  BottomPanel.add(SearchField);
  BottomPanel.add(SearchButton);
  add(BottomPanel,BorderLayout.SOUTH);
  CaseSensitive.setSelected(true);

  // register functions to the components
  ComboBox.addActionListener(new ActionListener(){
  public void actionPerformed(ActionEvent evt){
    showObject();
  }});

  SearchField.addKeyListener(new KeyAdapter(){
  public void keyPressed(KeyEvent evt){
    if( evt.getKeyCode() == KeyEvent.VK_ENTER )
      searchText();
  }});

  SearchButton.addActionListener(new ActionListener(){
    public void actionPerformed(ActionEvent evt){
      searchText();
  }});

  // build the MenuExtension
  JMenu SettingsMenu = new JMenu("Settings");
  JMenu HeaderColorMenu = new JMenu("header color");
  JMenu CellColorMenu = new JMenu("cell color");
  SettingsMenu.add(HeaderColorMenu);
  SettingsMenu.add(CellColorMenu);
  ActionListener HeaderColorChanger = new ActionListener(){
    public void actionPerformed(ActionEvent evt){
      JMenuItem S = (JMenuItem) evt.getSource();
```

```
        HeaderColor = S.getText().trim();
        reformat();
      }
    };
    ActionListener CellColorChanger = new ActionListener(){
      public void actionPerformed(ActionEvent evt){
        JMenuItem S = (JMenuItem) evt.getSource();
        CellColor = S.getText().trim();
        reformat();
      }
    };
    // some colors for the menuextension
    HeaderColorMenu.add("white").addActionListener(HeaderColorChanger);
    HeaderColorMenu.add("silver").addActionListener(HeaderColorChanger);
    CellColorMenu.add("yellow").addActionListener(CellColorChanger);
    CellColorMenu.add("aqua").addActionListener(CellColorChanger);

    MV.addMenu(SettingsMenu);
    }
```

The `addObject` method constructs the html-code for a new object and appends this object to all components managing `SecondoObject` representations.

```
  public boolean addObject(SecondoObject o){
    // check if object is correct
    if(!canDisplay(o))
      return false;
    // already displayed => only select
    if (isDisplayed(o))
      selectObject(o);
    else{
      // build the text and append object to all representations
      ListExpr VL = o.toListExpr().second();
      ObjectTexts.add(getHTMLCode(VL));
      ComboBox.addItem(o.getName());
      SecondoObjects.add(o);
      try{
        // ensure to diplay the new object
        ComboBox.setSelectedIndex(ComboBox.getItemCount()-1);
        showObject();
      }
      catch(Exception e){
        if(DEBUG_MODE)
          e.printStackTrace();
      }
    }
    return true;
  }
```

Don't forget to extend the `makefile` in the `Javagui/viewer` directory:

```
...
VIEWER_CLASSES := \
    FormattedViewer.class \
    InquiryViewer.class \
    QueryViewer.class \
    ...
```

Because this viewer never expects user interaction, the method `enableTestMode` can remain unchanged.

7.2.2 Extension of the HoeseViewer

This viewer can display textual, graphical and temporal objects, but it cannot display composite types except from the `relation` type. This means that in the list representation of a SECONDO object `(<type> <value>)`, `<type>` must be a single symbol atom or a description of a relation. To add a new object type to the HoeseViewer, a new class in the package `viewer.hoese.algebras` must be implemented. The name of this class must be `Dspl<type>`, where `<type>` is the symbol value of `<type>` in the list above. If another class name is chosen, the viewer can't find this class. If no new packages are included, the makefiles don't need to be changed. Depending on the kind of the new object type (textual, graphical or temporal), a class implementing different interfaces should be developed. For an easy extension, existing adapter classes can be used.

Creating Textual Objects

In the class for a new textual object, the `DsplBase` interface should be implemented. There is an adapter class `DsplGeneric`, which implements all methods from the `DsplBase` interface by default. The easiest way to add a new textual type is to extend this class. Then only the `init` method has to be overwritten. For a formatted output within relations, the additional Interface `DsplSimple` containing another `init` method has to be implemented.

Example: Inserting Rational Numbers

In this example, rational numbers will be displayed. The nested list structure for such objects looks as follows:

```
(rational (<sign> <intpart> <numDecimal> / <denomDecimal>))
```

where

- `<sign>` can be either the symbol "+" (positive number), "-" (negative number) or nothing (interpreted as positive)
- `<intpart>`, `<numDecimal>` and `<denomDecimal>` are non negative integer values with `<numDecimal> < <denomDecimal>`

The output format should be `rat: <sign> <numDecimal> / <demonDecimal>`, where the `<intpart>` from the nested list is integrated in this representation. This class is named `Dsplrational`.

The display class is shown below:

```
package viewer.hoese.algebras;

import sj.lang.ListExpr;
import viewer.hoese.*;

public class Dsplrational extends DsplGeneric implements DsplSimple{
  /* returns a string representing this rational or "ERROR" if
   *the format of the list is wrong
   */
  private String getValueString(ListExpr value){
    int len = value.listLength();
    if(len!=4 && len !=5)
      return "ERROR";
    String result="";
    if(value.listLength()==5){ // with sign
      ListExpr SignList = value.first();
      if(SignList.atomType()!=ListExpr.SYMBOL_ATOM)
        return "ERROR";
      String sign = SignList.symbolValue();
      if(sign.equals("-")) // ignore other values
        result += sign +" ";
      value = value.rest(); // skip the signum
    }
    // check the types
    if( value.first().atomType()!=ListExpr.INT_ATOM ||
      value.second().atomType()!=ListExpr.INT_ATOM ||
      value.fourth().atomType()!=ListExpr.INT_ATOM)
      return "ERROR";
    int intPart = value.first().intValue();
    int numDecimal = value.second().intValue();
    int denomDecimal = value.fourth().intValue();
    result += ""+(denomDecimal*intPart+numDecimal) + " / " + denomDecimal;
    return result;
  }

  public void init(ListExpr type, ListExpr value, QueryResult qr){
    qr.addEntry("rat : " + getValueString(value));
  }
  public void init (ListExpr type,int typewidth,ListExpr value,
                    int valuewidth, QueryResult qr){
    String T = new String(type.symbolValue());
    String V = getValueString(value);
    T=extendString(T,typewidth);
```

```
        V=extendString(V,valuewidth);
        qr.addEntry(T + " : " + V);
        return;
      }
    }
```

Inserting a Graphical Object

In a new display class for a graphical object, the interface `DsplGraph` has to be implemented. The adapter class is named `DisplayGraph`. The basic idea is to convert the nested list representation into a set of Java `Shape` Objects. Each element of this set has some properties which are used to display the object. The methods of interest are:

```
    int numberOfShapes()
```

This methods returns the number of how many Java `Shape` Objects are used to represent this SECONDO object. In the most cases, a single `Shape` is sufficient, and so the return value is set to `1` in these cases.

```
    boolean isPointType(int num)
    boolean isLineType(int num)
```

The HoeseViewer uses different methods to draw lines, points, and areas. To inform the viewer about the kind of a `Shape`, these functions must be overwritten (both methods return `false` by default). The arguments refers to the element `num` within the set of `Shapes`.

```
    Shape getRenderObject(int num, AffineTransform at)
```

This methods returns the element at position `num` within the set of `Shapes`. Because this function is called very often when temporal objects are animated, all `Shapes` should be precomputed if possible. The additional argument `at` of type `AffineTransform` can be used to show an object in fixed size independent of the currently used zoom factor.

```
    void init(ListExpr type, ListExpr value, QueryResult qr)
```

The `init` method converts the nested list passed by the parameter `value` into the set of Java `Shapes`. To be able to handle objects with geographic coordinates (longitude, latitude), the class `Projection` must be used. In contrast to textual objects, the instance of the display class itself has to be added to the query result. The textual representation comes from the `toString()` method of this class.

Example: Inserting a Rectangle Type

The nested list format for a Rectangle is given by the following format:

(rect (<x_1> <y_1> <x_2> <y_2>))

where x_i and y_i are numeric values. Because in Java a class `Rectangle2D.Double` implementing the `Shape` interface already exists, only the `init` method has to be overwritten. Here, the implementation of the class `Dsplrect` is shown:

```
package  viewer.hoese.algebras;

import java.awt.geom.*;
import java.awt.*;
import sj.lang.ListExpr;
import java.util.*;
import viewer.*;
import viewer.hoese.*;
import tools.Reporter;



/**
 * The displayclass for rectangles
 */
public class Dsplrect extends DisplayGraph {
  /** The internal datatype representation */
  Rectangle2D.Double rect;
  /**
    * Scans the numeric representation of a rectangle
    */
  private void ScanValue (ListExpr v) {
    if (v.listLength() != 4){
      Reporter.writeError("Error: No correct rectangle expression:"+
                          " 4 elements needed");
      err = true;
      return;
    }
    Double X1 = LEUtils.readNumeric(v.first());
    Double X2 = LEUtils.readNumeric(v.second());
    Double Y1 = LEUtils.readNumeric(v.third());
    Double Y2 = LEUtils.readNumeric(v.fourth());
    if(X1==null || X2==null || Y1==null | Y2==null){
      Reporter.writeError("Error: no correct rectangle "+
                          "expression (not a numeric)");
      err =true;
      return;
    }
    try{
      double tx1 = X1.doubleValue();
      double tx2 = X2.doubleValue();
      double ty1 = Y1.doubleValue();
      double ty2 = Y2.doubleValue();
      if(!ProjectionManager.project(tx1,ty1,aPoint)){
        err = true;
      } else{
        double x1 = aPoint.x;
        double y1 = aPoint.y;
        if(!ProjectionManager.project(tx2,ty2,aPoint)){
          err=true;
        }  else{
          double x2 = aPoint.x;
          double y2 = aPoint.y;
          double x = Math.min(x1,x2);
          double w = Math.abs(x2-x1);
          double y = Math.min(y1,y2);
```

```
              double h = Math.abs(y2-y1);
              rect = new Rectangle2D.Double(x,y,w,h);
            }
          }
        }catch(Exception e){
          err = true;
        }
      }

      public int numberOfShapes(){
        return 1;
      }

      /** Returns the rectangle to display **/
      public Shape getRenderObject(int num,AffineTransform at){
        if(num<1){
          return rect;
        } else{
          return null;
        }
      }

      /**
        * Init. the Dsplrect instance.
        * @param type The symbol rect
        * @param value The 4 Numeric  of a rectangle x1 x2 y1 y2
        * @param qr queryresult to display output.
        * @see sj.lang.ListExpr
        */
      public void init (ListExpr type, ListExpr value, QueryResult qr) {
        AttrName = type.symbolValue();
        ScanValue(value);
        if (err) {
          Reporter.writeError("Error in ListExpr :parsing aborted");
          qr.addEntry(new String("(" + AttrName + ": GA(rectangle))"));
          return;
        }
        else
          qr.addEntry(this);
      }
    }
```

Including a Graphical Temporal Type

In a display class for a graphical temporal type the `Timed` interface and the `DsplGraph` interface have to be implemented. To do that, only the `DisplayTimeGraph` class has to be extended.

Example: Including a Moving Point

A moving point is a point which changes its position over the time. The point is defined during a set of disjoint time intervals.

The list representation for a moving point is:

```
(mpoint (<unit1>...<unitn>))
```

In a unit, the point moves from a start point to an end point. The two positions may be the same. In this case the point is staying at its position during that time interval. A unit has the following structure:

```
unit := ( <interval> (x1 y1 x2 y2)),
```

where

```
<interval> := (<start> <end> <leftclosed><rightclosed>)
```

`<leftclosed>` and `<rightclosed>` are boolean atoms, describing whether the interval contains the appropriate end points.

`<start>` and `<end>` are of type `instant`, which is defined as a string in format:

```
year-month-day[-hour:minute[:second[.millisecond]]]
```

where the square brackets delimit optional values.

`(x1, y1)` defines the location of the moving point at the beginning of the time interval. `(x2, y2)` is the position of the moving point at the end of the time interval. The moving point moves linearly from `(x1, y1)` to `(x2, y2)` during the given time interval.

In the `init` method, the list is converted to an internal representation of a moving point. The bounding box is computed as the minimum rectangle containing all endpoints from the included units.

The `getRenderObject` checks whether the moving point is defined at a given time instant, which means that a unit whose interval contains the given time instant exists. If no unit is found, `getRenderObject` returns `null`. Otherwise the position of this moving point is computed. Around this position a rectangle or a circle is constructed to display this point. The complete source code is given in Appendix B. The class in the appendix additionally supports an old nested list format for moving points. Furthermore it can be used for labeling and manipulating the appearance of graphical objects which is described in the next sections.

Using Objects as a Label

An object can be used as a label of another object or of itself. This feature is available if the display class of this object implements the `LabelAttribute` interface. This interface contains only a single method `getLabel`. The parameter `time` is used only if the object state changes in time. Otherwise, this argument is ignored. This methods returns a string which is used as a label for the other object.

Using Objects for Manipulating the Appearance of Other Objects

In the HoeseViewer, objects can manipulate a predefined set of properties (e.g. color) of graphical objects. To do so, the display class has to implement the `RenderAttribute` interface which is described in this section. The basic idea is to convert the current value of the object into a real (`double`) value. From this number, the appropriate value of the property is derived.

Some of the methods of the `RenderAttribute` interface have an argument denoting the time currently displayed. If the value of the object does not depend on time, just ignore it. The `isDefined` method checks whether the object has a defined value at the given point in time. The `getRenderValue` method computes the value of the object and converts it into a `double` value. `mayBeDefined` checks if the object is defined at any point in time. The methods `getMinRenderValue` and `getMaxRendervalue` compute the minimum (maximum) value of the object in the `double` representation for all points in time.

Drawing Complex Objects

Sometimes, the classes implementing the `Shape` interface of Java are not able to display an object in the desired form. An example is a label class which should draw a string at given position and a given angle. Such display classes must be derived from the class `DisplayComplex`. Its `draw` method must be implemented. This function is called if the object is drawn to the screen.

Adding Special Views for Objects

The display of some objects requires a lot of space on the screen. Because the HoeseViewer has only a small area for displaying texts, sometimes a new window is needed to display objects in a nice way. For such objects, the interface `ExternDisplay` has to be implemented by the display class. This interface contains two methods `displayExtern` and `isExternDisplayed`. The `displayExtern` method is called if the user double clicks on the textual representation of the object. The reaction of calling this function is not restricted. Usually, a new window is opened and the object is displayed in this window. To avoid too many instances of such windows, the window should be a static instance of the display class. Thus all objects are displayed in the same window. The method `isExternDisplayed` returns `true` if the object is already presented in such a window.

7.3 Writing New Display Functions for SecondoTTY and SecondoTTYCS

The text based user interfaces of SECONDO (SecondoTTYBDB and SecondoTTYCS) can display objects in a formatted manner. In order to do so, display functions have to be defined. If no display function exists for a type, the result will be printed out in a nested list format. In this section we describe how to write and register new display functions.

To define a display function for a new type, two files, `DisplayTTY.h` (located in the `include` directory) and `DisplayTTY.cpp` (in the `UserInterfaces` directory) have to be changed. In `DisplayTTY.h` you have to add an entry for the new display function with the following format:

```
static void DisplayType( ListExpr type, ListExpr numType, ListExpr value )
```

where:

- `type` contains the type in the familiar manner (e.g. `(int)` or `(rel (tuple( ...)))`),
- `numType` contains also the type description, but here the types are encoded by the according algebra number and the number of the used type constructor in this algebra. (e.g. `(4 3)` or `((2 0)((3 1)(...)))`),
- `value` contains the value list.

7.3.1 Display Functions for Simple Types

Writing a display function for a simple (non-composite) type is very easy. Only the value list has to be analyzed and then written to the standard output in the desired format.

Example: Display Function for the Point Type

```
void
DisplayTTY::DisplayPoint( ListExpr type, ListExpr numType, ListExpr value)
{
  if(nl->ListLength(value)!=2)
    cout << "Incorrect Data Format";
  else{
    bool err;
    double x = getNumeric(nl->First(value),err);
    if(err){
      cout << "Incorrect Data Format";
      return;
    }
    double y = getNumeric(nl->Second(value),err);
    if(err){
      cout << "Incorrect Data Format";
      return;
    }
    cout << "point: (" << x << "," << y << ")";
  }
}
```

The `getNumeric` function has been defined in `DisplayTTY.` It returns the `double` value of a list containing an integer, a real, or a rational number. If the list does not contain a value of these types, the `err` parameter will be set to `true`. A short output should not end with a `newline`, because this can lead to conflicts when formatting composite types like relations or arrays which contain this simple type.

7.3.2 Display Functions for Composite Types

For a composite type (e.g. relation or array), the display function is defined by recursively calling `CallDisplayFunction` for the embedded types. The function `CallDisplayFunction` has four parameters. The last three parameters (`type`, `numType` and `value`) correspond to the parameters of display functions. The first parameter (`idPair`) is used to identify the correct display function. For a simple type, `numType` and `idPair` are equal, but for a composite type, `idPair` only contains the "main type" of the `numType` list.

Example: Display Function for an Array Type

The type description of an array is given as`(array <arraytype>)`. The list `<arraytype>` is a description of the embedded type, which may be simple or composite. To find the main type of the embedded type, the list <arraytype> is traversed using depth-first-search until an integer value is found. This integer represents the algebra number of the embedded type. The next element is an integer representing the embedded type's constructor id. For every element of the value list, `CallDisplayFunction` is invoked.

```
void
DisplayTTY::DisplayArray( ListExpr type, ListExpr numType, ListExpr value)
{
  if(nl->ListLength(value)==0)
    cout << "an empty array";
  else{
    ListExpr AType = nl->Second(type);
    ListExpr ANumType = nl->Second(numType);
    // find the idpair
    ListExpr idpair = ANumType;
    while(nl->AtomType(nl->First(idpair))!=IntType)
      idpair = nl->First(idpair);
    int No = 1;
    cout << "*************** BEGIN ARRAY ***************" << endl;
    while( !nl->IsEmpty(value)){
      cout << "--------------- Field No: " << No++ << " ---------------";
      cout << endl;
      CallDisplayFunction(idpair,AType,ANumType,nl->First(value));
      cout << endl;
      value = nl->Rest(value);
    }
    cout << "***************  END ARRAY  ***************";
  }
}
```

7.3.3 Register Display Functions

To register a new display function, just invoke `InsertDisplayFunction` in the `initialize` function of `DisplayTTY`. The calls for the above described functions are as follows:

```
InsertDisplayFunction( "array", &DisplayArray);
InsertDisplayFunction( "point", &DisplayPoint);
```

8 Query Progress Estimation

8.1 Overview

For some time now SECONDO has been equipped with query progress estimation. The user interfaces show a progress bar, indicating the fraction of the work for this query that has been completed, and the expected remaining time. To implement progress estimation, the query processor at regular time intervals (roughly ten times per second) sends a “progress query” to the operator at the root of the operator tree. An operator supporting progress estimation responds to this by returning some estimated quantities concerning the subtree rooted at this operator, namely:

- the total cardinality (i.e., the number of tuples to be returned)
- the tuple size in bytes
- the total time required to evaluate this subtree, in milliseconds
- the progress achieved, i.e., the fraction of work done, a number between 0 and 1

This set of quantities is not yet complete; it is refined below. To answer the question for the whole subtree, an operator normally asks its predecessors - sons in the operator tree - for their progress information. Based on these it derives its own progress quantities.

It is not required that all operators in SECONDO support query progress estimation. First of all, this concerns essentially operators that produce and/or consume streams of tuples. For example, the *filter* operator supports progress estimation, but the (usually atomic) operators needed to evaluate the filter predicate do not need to support it. Even operators processing tuple streams are free to support or not to support progress estimation. Basically, if all relevant operators in an operator tree support progress, then the query processor will receive a valid progress estimate and report it to the user interface. Otherwise no progress information appears.

For each operator it is individually registered within its algebra whether it supports progress. An operator requests progress information from its predecessor calling a method of the query processor. The query processor checks whether the predecessor operator supports progress estimation. If it does not, the query processor returns a `CANCEL` message to the current operator. Normally an operator receiving a `CANCEL` message from a predecessor cannot compute a reasonable progress estimate and returns `CANCEL` itself. If the query processor receives `CANCEL` at the root of the operator tree, it does not report progress.

Even if an operator supports progress in principle, it is not required to return valid estimates at all times (although this is desired). An operator may return `CANCEL` to a progress query. Again, subsequent operators - higher up in the operator tree - are then likely to report `CANCEL` themselves and no progress will be reported for this particular progress query. However, a bit later this operator may yield progress information and progress will be reported at the user interface.

Assuming now that an operator and its predecessors do support progress, how does the operator compute its progress estimate? First of all, it keeps track of the amount of work it has done such as the number of tuples read from its argument streams or the number of tuples returned. For this purpose, counters are inserted into the original code. Whereas observing the amount of work that has been done seems easy, the central problem in progress estimation (as in query optimization) is to estimate the total amount of work that needs to be done. This depends strongly on the sizes of intermediate results, hence in particular on the selectivities of operations implementing selections or joins.

8.2 Selectivity Estimation

Selectivities can be observed within an operator. For example, the *filter* operator can determine its selectivity as the fraction of returned vs. read tuples. Similarly, join operators such as *sortmergejoin* or *symmjoin* can determine their respective selectivity as the number of tuples returned relative to the size of the Cartesian product of their input streams. Based on the observed selectivity and the sizes of the input streams, the operator can estimate the cardinality of its output stream.

Note that this is a kind of sampling approach. Based on the fraction of qualifying tuples within the subset that has been processed (e.g. the initial part of a stream of tuples) we estimate selectivity for the entire set of tuples. The underlying assumption is that tuples arrive in random order (which is not always true). Further, the sample should be large enough. For this reason, the selectivity within an operator is only trusted after “some time”. More precisely, we assume that an estimate is stable when a sufficient number of positive tuples (i.e., for which the predicate is true) has been seen. This constant is currently set to 50.

Before an operator has seen this number of positive tuples, we call it *in cold state*. Afterwards it is in *warm state*. The question is what estimate of cardinality an operator can return in cold state. There are three possibilities: (1) rely on optimizer estimates, (2) use a default value, and (3) know the precise cardinality because the input stream is exhausted.

Query optimizer selectivity estimates exist if this query has been constructed by the optimizer. An interface exists (described below) for an operator to access the optimizer estimate. One can also access the observed predicate evaluation time and use it in the estimation of the total time needed by this operator and subtree.

However, a query can run without the optimizer and an operator be still in cold state. In that case, there is no other possibility than using a (stupid) default. For selection, a default selectivity of 0.1 is used; for join a selectivity is used such that the cardinality of the smaller relation is returned, guessing the somewhat frequent case that each tuple in the smaller relation connects to one tuple in the larger one.

Finally, if the input stream is exhausted, the cardinality to be reported is precisely the number of tuples returned by this operator.

8.3 Estimation of Tuple and Attribute Sizes

For the bottommost operators in the query tree, average tuple and attribute sizes are obtained from the relations accessed, e.g. by the *feed* operator. They are then propagated through subsequent operators. For most operators, their effect on tuple and attribute sizes is known precisely. Many operators leave the tuple structure unchanged or combine two tuples (join operators). Projection changes the tuple structure, but knows all attribute sizes, hence can determine correct sizes for the output tuples.

There are a few operators that compute new attributes as the result of expressions such as *extend* or *groupby*. For these derived attributes, sizes are initially unknown. However, the sizes of new attribute values can be observed as they are created. Observing sizes over time should yield a reasonable estimate (again assuming uniformity in the stream processed).

On the other hand, measuring attribute sizes induces some overhead. Note that this is done for each tuple in the proper query processing, not in progress queries (of which there are only a few per second). Therefore it is done only on an initial portion of the stream processed.

Furthermore, computing tuple sizes as the sum of attribute sizes in progress queries may also be a bit expensive, especially for relations with very large numbers of attributes. Therefore the following technique is used:

- Together with the sizes reported of tuples and attributes, an operator returns a boolean value `sizesChanged` to indicate whether these sizes have been recomputed for the current progress query.
- Within a progress query, tuple and attribute sizes are recomputed either if the predecessor reports a size change or if within this operator a threshold has been passed so that observed attribute sizes are now assumed to be stable. In this case the value `sizesChanged = true` is passed to the successor.

8.4 Pipelining

For a sequence of non-blocking operators each of which consumes a constant amount of time per tuple, e.g.

```
<rel> feed filter[...] project[...] consume
```

it is obvious that they work in a synchronized manner and their progress is the same. We say they form a pipeline. This is not true, however, when there are blocking operators in the sequence.

In such operators, one can check whether the subtree below has blocking operators (i.e., significant blocking time). If that is not the case, this operator can simply report the progress of its predecessor as its own progress. This strategy is more stable in some cases, if observed selectivity estimates are not correct or operators still in cold state.

Pipelining can be switched on as an option, by setting the constant `pipelinedProgress` to `true` in the file `include/Progress.h`.

8.5 Infrastructure for Progress Implementation

8.5.1 New Messages and Storage Management

Recall that for stream processing the query processor sends messages to operators, namely `OPEN`, `REQUEST`, and `CLOSE`. The standard protocol for processing a stream is

```
OPEN REQUEST* CLOSE
```

Usually in the `OPEN` part some initializations are done and data structures allocated. For each `REQUEST`, a tuple is returned. In `CLOSE`, data structures are deallocated. In fact, it is possible that a stream operator is called in a loop (e.g. embedded in a *loopjoin*), hence the protocol is more completely

```
(OPEN REQUEST* CLOSE)*
```

For progress estimation, two new messages are introduced called `REQUESTPROGRESS` and `CLOSEPROGRESS`. With `REQUESTPROGRESS`, the successor asks for progress information. The `CLOSEPROGRESS` message is used to deallocate data structures after completion of the entire query.

The `REQUESTPROGRESS` messages are sent “asynchronously” at any time within the protocol shown above. In particular, such a message may be sent after an operator has processed its stream completely, that is, after the `CLOSE` message. However, the branch of the operator implementation that answers progress queries still needs access to the operator’s data structure that manages progress information such as counters. It follows that we must not deallocate the data structure in the `CLOSE` branch. On the other hand, it is necessary to deallocate the data structure at some point to release the storage.

For this reason, the protocol is extended to use the `CLOSEPROGRESS` message which is guaranteed to be sent only once after completion of the entire query:

```
(OPEN REQUEST* CLOSE)* CLOSEPROGRESS
```

The allocation and deallocation of the operator’s data structure *D* with the protocol that was used so far can be represented as follows:

```
OPEN            create D
REQUEST
...
REQUEST
CLOSE           delete D

...

OPEN            create D
REQUEST
...
REQUEST
CLOSE           delete D
```

With progress estimation, allocation and deallocation is done as follows:

```
OPEN            if D exists delete D; create D
REQUEST
...
REQUEST
CLOSE

...

OPEN            if D exists delete D; create D
REQUEST
...
REQUEST
CLOSE

CLOSEPROGRESS   delete D
```

In this way it is ensured that the data structure is available for the entire evaluation time of the query. Hence progress queries can be answered at any time. But data structures are also properly released to avoid storage holes.

8.5.2 Data Structures

Two data structures are defined in the file `include/Progress.h` to support the implementation of progress estimation in operators. The first, given in class `ProgressInfo` represents the various quantities of progress information mentioned above. Pointers to instances of this class are passed between operators.

```
class ProgressInfo
{
public:

  ProgressInfo();

  double Card;           //expected cardinality
  double Size;           //expected total tuple size (including FLOBs)
  double SizeExt;        //expected size of tuple root and extension part
                         //   (no FLOBs)
  int noAttrs;           //no of attributes
  double *attrSize;      //for each attribute, the complete size
  double *attrSizeExt;   //for each attribute, the root and extension size
  bool sizesChanged;     //true if sizes have been recomputed in this request

  double Time;           //expected time, in millisecond
  double Progress;       //a number between 0 and 1

  double BTime;          //expected time, in millisecond of blocking ops
  double BProgress;      //a number between 0 and 1


  void CopySizes(ProgressInfo p);           //copy the size fields

  void CopySizes(ProgressLocalInfo* pli);   //copy the size fields
  void CopyBlocking(ProgressInfo p);//copy BTime, BProgress
          //for non blocking unary op.

  void CopyBlocking(ProgressInfo p1,ProgressInfo p2);
                          //copy BTime, BProgress
                          //for non-blocking binary op. (join)

  void Copy(ProgressInfo p);                //copy all fields

};
```

The quantities mentioned above are refined as follows. For tuple size, both the size of the core tuple without FLOBs and the complete tuple size including FLOBs are maintained. Further, the number of attributes and the two respective sizes for each attribute are kept. This is necessary so that operators changing the tuple schema (e.g. projection) can recompute the size for their result tuples.

In addition to `Time` and `Progress`, an operator also estimates the blocking time `BTime` and and blocking progress `BProgress` for the subtree of which it is the root. For example, the *sort* operator first reads the entire input stream before it returns any tuple. During this time it is blocking. The time estimated for this stage is the blocking time, and the fraction of work done within the blocking stage is the blocking progress.

The class offers some additional methods to easily copy some of the fields.

The second data structure, called `ProgressLocalInfo`, provides some counters and other fields that can be used to augment an operator’s local data structure.

```
class ProgressLocalInfo
{
public:

  ProgressLocalInfo();

  ~ProgressLocalInfo();

  int returned;          //current number of tuples returned
  int read;              //no of tuples read from arg stream
  int readFirst;         //no of tuples read from first arg stream
  int readSecond;        //no of tuples read from second argument stream
  int total;             //total number of tuples in argument relation
  int defaultValue;      //default assumption of result size, needed for
                         //some operators
  int state;             //to keep state info if needed
  int memoryFirst,
    memorySecond;        //size of buffers for first and second argument

  void* firstLocalInfo; //pointers to localinfos of first and second arg
  void* secondLocalInfo;

  bool sizesInitialized;//size fields only defined if sizesInitialized;
                         //initialized means data structures are allocated
                         //and fields are filled
  bool sizesChanged;     //sizes were recomputed in last call
```

```
      double Size;          //total tuplesize
      double SizeExt;       //size of root and extension part of tuple
      int noAttrs;          //no of attributes
      double *attrSize;     //full size of each attribute
      double *attrSizeExt;  //size of root and ext. part of each attribute

      void SetJoinSizes( ProgressInfo& p1, ProgressInfo& p2 ) ;

          //set the sizes for a join of first and second argument
          //only done when sizes are initialized or have changed

};
```

Not all of these fields are used by all operators. Also, it is not mandatory to use this data structure. In fact, some operators were equipped with progress estimation before this data structure was defined. For an operator that needs to manage its own data for regular execution, a data structure is often introduced as a subclass of `ProgressLocalInfo`, for example:

```
    class XLocalInfo: public ProgressLocalInfo
    {
      <further fields and methods for operator X>
    }
```

8.5.3 Interface to Request Progress Information From a Predecessor

An operator can request progress information from one of its arguments using a method of the query processor (defined in `include/QueryProcessor.h`):

```
    bool RequestProgress( const Supplier s, ProgressInfo* p );
```

This evaluates the subtree *s* for a `PROGRESS` message. It returns true iff a progress info has been received. In *p* the address of a `ProgressInfo` must be passed.

8.5.4 Interface to Access Optimizer Selectivity Estimate and Predicate Cost

If the query was produced by the optimizer, for each predicate its selectivity and predicate evaluation cost was determined in a query on a sample. These results are stored in the operator tree at the node of the respective filter or join operator. The implementation of such an operator can access these quantities as follows (defined in `include/QueryProcessor.h`):

```
    double GetSelectivity( const Supplier s);
```

From a given supplier *s* get the selectivity at this node.

```
    double GetPredCost( const Supplier s);
```

From a given supplier *s* get the predicate cost at this node.

8.6 Some Example Operators

In this section we look at the code of some operators that provide progress estimation.

8.6.1 Rename

This is the most simple operator because it essentially does nothing, just passes a tuple from predecessor to successor. In a sense, this is the “Hello, World” of progress estimation.

```
int
Rename(Word* args, Word& result, int message,
       Word& local, Supplier s)
{
  Word t; Tuple* tuple;

  switch (message)
  {
    case OPEN :
      qp->Open(args[0].addr);
      return 0;

    case REQUEST :
      qp->Request(args[0].addr,t);
      if (qp->Received(args[0].addr))
      {
        tuple = (Tuple*)t.addr;
        result.setAddr(tuple);
        return YIELD;
      }
      else return CANCEL;

    case CLOSE :
      qp->Close(args[0].addr);
      return 0;

    case CLOSEPROGRESS:
      return 0;

    case REQUESTPROGRESS:
      ProgressInfo p1;
      ProgressInfo *pRes;

      pRes = (ProgressInfo*) result.addr;

      if ( qp->RequestProgress(args[0].addr, &p1) )
      {
        pRes->Copy(p1);
        return YIELD;
      }
      else return CANCEL;
  }
  return 0;
}
```

Besides the usual three branches OPEN, REQUEST, and CLOSE of a stream processing operator, this operator has branches CLOSEPROGRESS and REQUESTPROGRESS. The operator does not allocate any data structures, therefore nothing happens in CLOSEPROGRESS.

In REQUESTPROGRESS a local variable p1 for ProgressInfo is declared as well as a pointer to such a variable pRes. The latter is set to result.addr which means it is assigned the address of a ProgressInfo variable in the successor operator (by some slight misuse of the result parameter).

The operator then gets progress information from its predecessor args[0]. That operator will either write such information into p1 and return YIELD which results in the query processor method RequestProgress returning TRUE. Or it will return CANCEL upon which RequestProgress returns FALSE.

This operator uses only a negligible amount of time itself. It also does not change tuple size or expected cardinality. Therefore it simply copies the progress information it has received from its predecessor into the structure of its successor, provided that the predecessor has delivered progress information. In that case it returns YIELD itself. Otherwise it cannot provide progress information and returns CANCEL.

8.6.2 Project

The original code of the project operator is shown below:

```
int
Project(Word* args, Word& result, int message,
        Word& local, Supplier s)
{
  switch (message)
  {
    case OPEN :
    {
      ListExpr resultType = GetTupleResultType( s );
      TupleType *tupleType = new TupleType(nl->Second(resultType));
      local.addr = tupleType;

      qp->Open(args[0].addr);
      return 0;
    }
    case REQUEST :
    {
      Word elem1, elem2;
      int noOfAttrs, index;
      Supplier son;

      qp->Request(args[0].addr, elem1);
      if (qp->Received(args[0].addr))
      {
        TupleType *tupleType = (TupleType *)local.addr;
        Tuple *t = new Tuple( tupleType );
        noOfAttrs = ((CcInt*)args[2].addr)->GetIntval();
        assert( t->GetNoAttributes() == noOfAttrs );

        for( int i = 0; i < noOfAttrs; i++)
        {
          son = qp->GetSupplier(args[3].addr, i);
          qp->Request(son, elem2);
          index = ((CcInt*)elem2.addr)->GetIntval();
          t->CopyAttribute(index-1, (Tuple*)elem1.addr, i);
        }
        ((Tuple*)elem1.addr)->DeleteIfAllowed();
        result.setAddr(t);
        return YIELD;
      }
      else return CANCEL;
    }
    case CLOSE :
    {
      qp->Close(args[0].addr);
      if(local.addr)
      {
        ((TupleType *)local.addr)->DeleteIfAllowed();
        local.setAddr(0);
      }
      return 0;
    }
  }
  return 0;
}
```

We now discuss step by step the changes needed for the progress version.

```
class ProjectLocalInfo: public ProgressLocalInfo
{
public:
  ProjectLocalInfo() {
    tupleType = 0;
    read = 0;
}

  ~ProjectLocalInfo() {
    tupleType->DeleteIfAllowed();
    tupleType = 0;
  }

  TupleType *tupleType;
};
```

First a local data structure `ProjectLocalInfo` is declared which inherits the fields from `Progress-LocalInfo`. Note that in the original version a tuple type is maintained between calls. Hence a field `tupleType` is defined in this class together with constructor and destructor methods.

```
16   int
17   Project(Word* args, Word& result, int message,
18           Word& local, Supplier s)
19   {
20     ProjectLocalInfo *pli=0;
21     Word elem1(Address(0));
22     Word elem2(Address(0));
23     int noOfAttrs= 0;
24     int index= 0;
25     Supplier son;
26
27     switch (message)
28     {
29       case OPEN:{
30
31         pli = (ProjectLocalInfo*) local.addr;
32         if ( pli ) delete pli;
33
34         pli = new ProjectLocalInfo();
35         pli->tupleType = new TupleType(nl->Second(GetTupleResultType(s)));
36         local.setAddr(pli);
37
38         qp->Open(args[0].addr);
39         return 0;
40       }
```

In the `OPEN` branch, the `ProjectLocalInfo` data structure is allocated and stored in the local variable (line 34). Observe that it is first deleted if present (line 32) as explained in Section 8.5.1.

```
41   case REQUEST:{
42
43         pli = (ProjectLocalInfo*) local.addr;
44
45         qp->Request(args[0].addr, elem1);
46         if (qp->Received(args[0].addr))
47         {
48           pli->read++;
49           Tuple *t = new Tuple( pli->tupleType );
50
51           noOfAttrs = ((CcInt*)args[2].addr)->GetIntval();
52           assert( t->GetNoAttributes() == noOfAttrs );
53
54           for( int i = 0; i < noOfAttrs; i++)
55           {
56             son = qp->GetSupplier(args[3].addr, i);
57             qp->Request(son, elem2);
58             index = ((CcInt*)elem2.addr)->GetIntval();
59             t->CopyAttribute(index-1, (Tuple*)elem1.addr, i);
60           }
61           ((Tuple*)elem1.addr)->DeleteIfAllowed();
62           result.setAddr(t);
63           return YIELD;
64         }
65         else return CANCEL;
66       }
```

The `REQUEST` branch is as before except that a counter for the read tuples has been inserted (line 48).

```
    case CLOSE: {

      // Note: object deletion is done in repeated OPEN or CLOSEPROGRESS
      qp->Close(args[0].addr);
      return 0;
    }
```

Nothing is deleted in the `CLOSE` branch.

```
    case CLOSEPROGRESS:{
      pli = (ProjectLocalInfo*) local.addr;
      if ( pli ){
         delete pli;
         local.setAddr(0);
      }
      return 0;
    }
```

Instead, this is done in `CLOSEPROGRESS`.

```
case REQUESTPROGRESS:{

      ProgressInfo p1;
      ProgressInfo *pRes;
      const double uProject = 0.00073;  //millisecs per tuple
      const double vProject = 0.0004;   //millisecs per tuple and attribute

      pRes = (ProgressInfo*) result.addr;
      pli = (ProjectLocalInfo*) local.addr;

      if ( !pli ) return CANCEL;

      if ( qp->RequestProgress(args[0].addr, &p1) )
      {
        pli->sizesChanged = false;

        if ( !pli->sizesInitialized )
        {
          pli->noAttrs = ((CcInt*)args[2].addr)->GetIntval();
          pli->attrSize = new double[pli->noAttrs];
          pli->attrSizeExt = new double[pli->noAttrs];
        }

        if ( !pli->sizesInitialized || p1.sizesChanged )
        {
          pli->Size = 0;
          pli->SizeExt = 0;

          for( int i = 0; i < pli->noAttrs; i++)
          {
            son = qp->GetSupplier(args[3].addr, i);
            qp->Request(son, elem2);
            index = ((CcInt*)elem2.addr)->GetIntval();
            pli->attrSize[i] = p1.attrSize[index-1];
            pli->attrSizeExt[i] = p1.attrSizeExt[index-1];
            pli->Size += pli->attrSize[i];
            pli->SizeExt += pli->attrSizeExt[i];
118           }
119           pli->sizesInitialized = true;
120           pli->sizesChanged = true;
121         }
122
123         pRes->Card = p1.Card;
124         pRes->CopySizes(pli);
125
126         pRes->Time = p1.Time + p1.Card *
127           (uProject + pli->noAttrs * vProject);
128
129         //only pointers are copied; therefore the tuple sizes do not
130         //matter
131
132         if ( p1.BTime < 0.1 && pipelinedProgress )    //non-blocking,
133                                                       //use pipelining
134           pRes->Progress = p1.Progress;
135         else
136           pRes->Progress =
137             (p1.Progress * p1.Time +
138               pli->read * (uProject + pli->noAttrs * vProject))
139             / pRes->Time;
140
141         pRes->CopyBlocking(p1);    //non-blocking operator
142         return YIELD;
143       }
144       else return CANCEL;
145     }
146     return 0;
147   }
148   return 0;
149 }
```

Lines 83, 84, and 88 are as in the previous example. In lines 85-86 two constants are defined to be used later in time estimations. They have been obtained in experiments.

Line 91 checks whether the local data structure has been allocated and otherwise returns `CANCEL`. Remember that a progress query may come at any time, possibly before the execution of this operator (i.e. the `OPEN` branch) has been started.

In line 93 progress information is requested from the predecessor `args[0]`. Again, if it is not available, `CANCEL` is returned (line 144).

In lines 95-121 the size fields (tuple and attribute sizes) for the result tuples are set in the local data structure `pli`. Lines 97-101 allocate space and are executed only once, because in line 119 `sizesInitialized` is set to true. In the following part attribute and tuple sizes are (re)computed. This is done either for initialization or if the predecessor reports a change of sizes.

In lines 123-141 the various quantities for progress are computed and written into the `ProgressInfo` data structure of the successor. As *project* does not change the number of tuples, it passes the cardinality obtained from the predecessor to the successor (line 123). It copies the sizes of tuples and attributes from the local data structure `pli` to the data structure of the successor (line 124).

The total time needed for this subtree (lines 126-127) consists of the time estimated by the predecessor plus the contribution of this operator. The time needed for *project* itself is proportional to the number of tuples received `p1.Card`. For each tuple there is a constant amount of time needed, represented by `uProject`, and some work required per attribute, `vProject`.

The progress is determined either by pipelining (see Section 8.4) or by the fraction of the time obtained by replacing the total cardinality to be processed (`p1.Card`) by the number of tuples read (`pli->read`) in the formula for total time, divided by the total time (lines 136-139).

As this operator is non-blocking, blocking time and progress are just passed from the predecessor to the successor (line 136).

8.6.3 Filter

Here is the original code of the filter operator:

```
int
Filter(Word* args, Word& result, int message,
       Word& local, Supplier s)
{
  bool found = false;
  Word elem, funresult;
  ArgVectorPointer funargs;
  Tuple* tuple = 0;

  switch ( message )
  {

    case OPEN:

      qp->Open (args[0].addr);
      return 0;

    case REQUEST:

      funargs = qp->Argument(args[1].addr);
      qp->Request(args[0].addr, elem);
      found = false;
      while (qp->Received(args[0].addr) && !found)
      {
        tuple = (Tuple*)elem.addr;
        (*funargs)[0] = elem;
        qp->Request(args[1].addr, funresult);
        if (((StandardAttribute*)funresult.addr)->IsDefined())
        {
          found = ((CcBool*)funresult.addr)->GetBoolval();
        }
        if (!found)
        {
          tuple->DeleteIfAllowed();
          qp->Request(args[0].addr, elem);
        }
      }
      if (found)
      {
        result.setAddr(tuple);
        return YIELD;
      }
      else
        return CANCEL;

    case CLOSE:

      qp->Close(args[0].addr);
      return 0;
  }
  return 0;
}
```

Again, we discuss the changes needed to support progress estimation.

```
struct FilterLocalInfo
{
  int current;    //tuples read
  int returned;    //tuples returned
  bool done;      //arg stream exhausted
};

```

A simple local data structure is defined.

```
int
Filter(Word* args, Word& result, int message,
       Word& local, Supplier s)
{
  bool found = false;
  Word elem, funresult;
  ArgVectorPointer funargs;
  Tuple* tuple = 0;
  FilterLocalInfo* fli;

  switch ( message )
  {
    case OPEN:

      fli = (FilterLocalInfo*) local.addr;
      if ( fli ) delete fli;

      fli = new FilterLocalInfo;
        fli->current = 0;
        fli->returned = 0;
        fli->done = false;
      local.setAddr(fli);

      qp->Open (args[0].addr);
      return 0;

```

The local data structure is (re)allocated and initialized.

```
34       case REQUEST:
35
36         fli = (FilterLocalInfo*) local.addr;
37
38         funargs = qp->Argument(args[1].addr);
39         qp->Request(args[0].addr, elem);
40         found = false;
41         while (qp->Received(args[0].addr) && !found)
42         {
43           fli->current++;
44           tuple = (Tuple*)elem.addr;
45           (*funargs)[0] = elem;
46           qp->Request(args[1].addr, funresult);
47           if (((StandardAttribute*)funresult.addr)->IsDefined())
48           {
49             found = ((CcBool*)funresult.addr)->GetBoolval();
50           }
51           if (!found)
52           {
53             tuple->DeleteIfAllowed();
54             qp->Request(args[0].addr, elem);
55           }
56         }
57         if (found)
58         {
59           fli->returned++;
60           result.setAddr(tuple);
61           return YIELD;
62         }
63         else
64         {
65           fli->done = true;
66           return CANCEL;
67         }
68
```

The code of the `REQUEST` branch is extended to count the numbers of tuples read and returned (lines 43 and 59) and to note when the input stream is exhausted (line 65).

```
69       case CLOSE:
70         qp->Close(args[0].addr);
71         return 0;
72
73       case CLOSEPROGRESS:
74         fli = (FilterLocalInfo*) local.addr;
75         if ( fli )
76         {
77            delete fli;
78            local.setAddr(0);
79         }
80         return 0;
81
```

Deallocation is not done in `CLOSE` but in `CLOSEPROGRESS` (and in `OPEN`, line 23).

```
      case REQUESTPROGRESS:

        ProgressInfo p1;
        ProgressInfo* pRes;
        const double uFilter = 0.01;

        pRes = (ProgressInfo*) result.addr;
        fli = (FilterLocalInfo*) local.addr;

        if ( qp->RequestProgress(args[0].addr, &p1) )
        {
          pRes->CopySizes(p1);

          if ( fli )    //filter was started
          {
            if ( fli->done )      //arg stream exhausted, all known
            {
              pRes->Card = (double) fli->returned;
              pRes->Time = p1.Time + (double) fli->current
                * qp->GetPredCost(s) * uFilter;
              pRes->Progress = 1.0;
              pRes->CopyBlocking(p1);
              return YIELD;
            }

            if ( fli->returned >= enoughSuccessesSelection )
              //stable state assumed now
            {
              pRes->Card =  p1.Card *
                ( (double) fli->returned / (double) (fli->current));
              pRes->Time = p1.Time + p1.Card * qp->GetPredCost(s) * uFilter;

              if ( p1.BTime < 0.1 && pipelinedProgress )   //non-blocking,
                                                            //use pipelining
                pRes->Progress = p1.Progress;
              else
                pRes->Progress = (p1.Progress * p1.Time
                  + fli->current * qp->GetPredCost(s) * uFilter) / pRes->Time;

              pRes->CopyBlocking(p1);
              return YIELD;
            }
          }
          //filter not yet started or not enough seen

          pRes->Card = p1.Card * qp->GetSelectivity(s);
          pRes->Time = p1.Time + p1.Card * qp->GetPredCost(s) * uFilter;

          if ( p1.BTime < 0.1 && pipelinedProgress )   //non-blocking,
                                                        //use pipelining
            pRes->Progress = p1.Progress;
          else
            pRes->Progress = (p1.Progress * p1.Time) / pRes->Time;
            pRes->CopyBlocking(p1);
          return YIELD;
        }
```

```
138      else return CANCEL;
139    }
140    return 0;
141  }
```

Lines 82-92 have no surprises. The filter operator does not change the tuple structure, hence it just copies sizes from the predecessor (line 93).

Lines 95-124 treat the case that the *filter* operator has executed its `OPEN` method and the local data structure `fli` exists.

Within this case, lines 97-105 handle the case that *filter* has also finished its work, i.e. the input stream is exhausted. In this case, the result cardinality is just the number of tuples returned. The total time needed for the subtree is the time needed for the predecessor subtree plus the contribution of the *filter* operator itself. This operator needs time proportional to the number of tuples read. For each tuple, the cost is the cost of predicate evaluation obtained from the optimizer times a constant `uFil-ter`. (If the query did no come from the optimizer, a default cost for predicate evaluation is stored in the operator tree.) Since in this case we are done, progress is 1.0. The *filter* operator does not have blocking time, hence it just copies blocking quantities from the predecessor.

Lines 107-123 handle the case that the operator is in warm state. The result cardinality is the fraction of returned vs. read tuples. Result time, progress, and blocking information are computed in the obvious way.

Lines 125-137 treat the case that either the `OPEN` method of filter was not yet executed (hence the `fli` data structure is not available) or the operator is not yet in warm state. In this case, selectivity is obtained from the query tree, which may be either an optimizer estimate or a default. Progress is determined by the progress of the predecessor as this operator has not yet processed any tuples.

8.7 Registering Progress Operators

Once progress estimation has been implemented for an operator, one needs to register this operator as supporting progress by calling a method `EnableProgress()`. For example at the end of the file `RelationAlgebra.cpp`, the operator *project* is registered by

```
relalgproject.EnableProgress();
```

8.8 Testing Progress Implementations

There are two main techniques to test whether operator implementations determine and propagate progress quantities in the correct way. The first is to switch on a trace mode in the query processor. The second is to look at the protocol files written during query processing.

8.8.1 Tracing

To switch on tracing, one has to modify a line in the file `QueryProcessor/QueryProcessor.cpp`. In the method `RequestProgress` one needs to modify the line

```
bool trace = false;  //set to true for tracing
```

setting variable `trace` to `true`. Obviously one then also needs to recompile SECONDO.

As a result, for every `RequestProgress` message that is sent from operator *x* to one of its predecessors *y*, one can see the resulting quantities delivered by *y*. Here is an example.

```
Secondo => query plz feed filter[.Ort contains "x"] Orte feed {o} hash-
join[Ort, Ort_o, 99997] count
Secondo ->
RequestProgress called with Supplier = 0xbb3f870  ProgressInfo* = 0x22e490
RequestProgress called with Supplier = 0xbb3f9a8  ProgressInfo* = 0x22e2b0
RequestProgress called with Supplier = 0xbb3fae0  ProgressInfo* = 0x22e0c0
RequestProgress called with Supplier = 0xbb3fc18  ProgressInfo* = 0x22dd90
Return from supplier 0xbb3fc18
Cardinality = 41267
Size = 20
SizeExt = 20
noAttrs = 2
attrSize[i] = 5 15
attrSizeExt[i] = 5 15
sizesChanged = 1
BlockingTime = 0.001
BlockingProgress = 1
Time = 96.243
Progress = 0.0836726
=================
Return from supplier 0xbb3fae0
Cardinality = 4126.7
Size = 20
SizeExt = 20
noAttrs = 2
attrSize[i] = 5 15
attrSizeExt[i] = 5 15
sizesChanged = 1
BlockingTime = 0.001
BlockingProgress = 1
Time = 137.51
Progress = 0.0836726
=================
RequestProgress called with Supplier = 0xb80b960  ProgressInfo* = 0x22e070
RequestProgress called with Supplier = 0xb80ba98  ProgressInfo* = 0x22ddb0
Return from supplier 0xb80ba98
Cardinality = 506
Size = 42
SizeExt = 42
noAttrs = 4
attrSize[i] = 8 18 11 5
attrSizeExt[i] = 8 18 11 5
sizesChanged = 1
BlockingTime = 0.001
```

```
BlockingProgress = 1
Time = 1.40298
Progress = 0.998028
=================
Return from supplier 0xb80b960
Cardinality = 506
Size = 42
SizeExt = 42
noAttrs = 4
attrSize[i] = 8 18 11 5
attrSizeExt[i] = 8 18 11 5
sizesChanged = 1
BlockingTime = 0.001
BlockingProgress = 1
Time = 1.40298
Progress = 0.998028
=================
Return from supplier 0xbb3f9a8
Cardinality = 506
Size = 62
SizeExt = 62
noAttrs = 6
attrSize[i] = 5 15 8 18 11 5
attrSizeExt[i] = 5 15 8 18 11 5
sizesChanged = 1
BlockingTime = 4.79418
BlockingProgress = 1
Time = 238.482
Progress = 0.0699726
=================
Return from supplier 0xbb3f870
Cardinality = 506
Size = 62
SizeExt = 62
noAttrs = 6
attrSize[i] = 5 15 8 18 11 5
attrSizeExt[i] = 5 15 8 18 11 5
sizesChanged = 1
BlockingTime = 4.79418
BlockingProgress = 1
Time = 238.482
Progress = 0.0699726
=================
...
```

In the listing one can see the calls to operator nodes (identified by supplier addresses) and the responses. The root of the tree is supplier `0xbb3f870` and the reply from it is returned as the last one (in this round). The order in which operator nodes reply corresponds to a postorder traversal of the operator tree. Hence the first set of answers is from the *feed* operator applied to `plz`, the next from *filter*. After that, the *hashjoin* operator sends a request to its second argument; then replies come from *feed* on `Orte`, then from *rename*, *hashjoin*, and *count*.

8.8.2 Looking at Protocol Files

During the execution of a query, for each progress query executed, a line is written to a protocol file containing the following fields:

- *CurrentTime* - the system time passed since the start of this query, in milliseconds
- *Card* - the estimated cardinality
- *Time* - the estimated total time for the query
- *Progress* - the estimated progress in percent

The protocol file is called `proglogt.csv` and it is located in the `secondo/bin` directory. New protocol lines are always appended; to reinitialize one can simply delete the file.

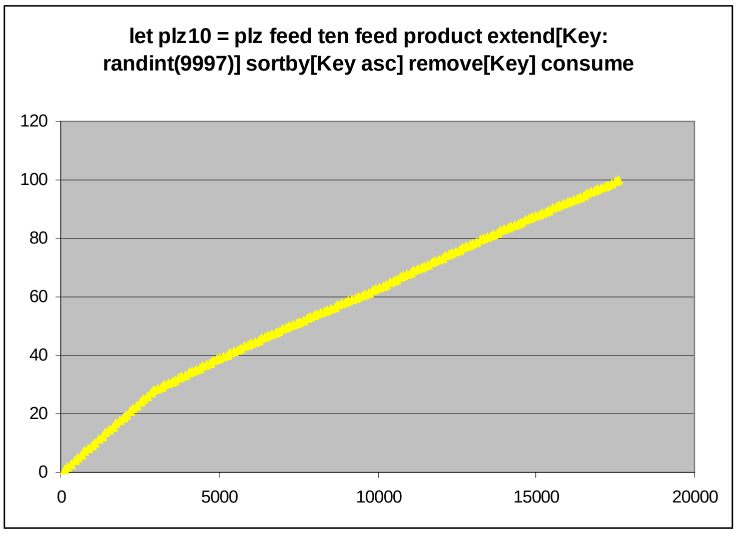
To study the behaviour of progress estimation, one can load this file into an EXCEL sheet and create xy-diagrams for the listed quantities. For example, for the query

```
let plz10 = plz feed ten feed product extend[Key: randint(9997)] sortby[Key
asc] remove[Key] consume
```

which creates a larger version of plz in randomized order, the protocol file starts

```
109;412670;22478;  0,86;
219;412670;22478;  1,87;
328;412670;22478;  2,91;
437;412670;22478;  3,97;
547;412670;22478;  4,99;
656;412670;22478;  6,05;
766;412670;22478;  7,12;
875;412670;22478;  8,20;
984;412670;22478;  9,14;
1094;412670;22478; 10,22;
...
```

One can see that there is roughly one progress query every 100 milliseconds. From the protocol one can create e.g. the *Progress* diagram shown below. Diagrams for *Card* and *Time* are not interesting as these numbers do not change in this query.



8.9 Implementation Techniques for Blocking Operators

As a final issue we consider the implementation of blocking operators. For example, *sortby* first consumes its entire argument stream before it returns any tuples.

Without progress estimation, a natural implementation strategy for such operators is to consume the entire argument stream within the `OPEN` branch, organizing tuples within some data or file structure. Later, for each `REQUEST` message, one tuple is returned.

However, with progress estimation this strategy does not work well. The reason is that in the `OPEN` branches data structures for all operators are set up, including the structures needed for progress estimation. If one operator spends a lot of time in its `OPEN` branch as with the strategy above, progress estimation will only start to report results once that operator has finished its blocking phase.

Therefore it is mandatory that in the `OPEN` branch of any operator only a small, constant amount of work is done. For a blocking operator this means that consuming the input stream(s) must be moved into the `REQUEST` branch. One can do that on the first `REQUEST` message, remembering in a local variable whether this is the first `REQUEST`.

A related problem is how one can convert an implementation of a blocking operator to support progress estimation if this operator does a lot of work within the constructor of its local data structure. For example, the *sortby* operator in its standard implementation defines a class `SortByLocalInfo` and constructs an instance of that class within its `OPEN` branch:

```
SortByLocalInfo* li = new SortByLocalInfo( args[0], lexicographically,
                                           tupleCmp );
```

We can move this statement into the `REQUEST` branch to execute with the first call as discussed. However, then we need a second data structure to hold progress information, to be initialized within the `OPEN` branch. Furthermore, within the constructor of `SortByLocalInfo` one needs to maintain counters and therefore to access the other data structure containing the progress fields.

A solution for this problem is provided within the file `Progress.h` (in `secondo/include`) in the form of two classes `LocalInfo and ProgressWrapper`.

```
template<class T>
class LocalInfo : public ProgressLocalInfo {

  public:
    LocalInfo() : ProgressLocalInfo(), ptr(0) {}
    ~LocalInfo() { if (ptr) delete ptr; }

    inline void deletePtr() { if (ptr) {delete ptr; ptr = 0; } }

    T* ptr;
};
```

The class `LocalInfo` provides the `ProgressLocalInfo` structure (Section 8.5.2) together with a pointer to an instance of the argument class `T`. As an argument class one uses the existing local data structure. Now in the `OPEN` branch this structure can be created:

```
li = new LocalInfo<SortByLocalInfo>;
```

Class `LocalInfo` has the fields of a `ProgressLocalInfo` and can be used from now on. This statement terminates quickly. However, the time consuming construction of `SortByLocalInfo` has been postponed. It can later be done in the first `REQUEST`.

In addition, within class `SortByLocalInfo` one needs access to `li`. To enable this, class `SortByLocalInfo` is declared to be a subclass of a class `ProgressWrapper`:

```
class SortByLocalInfo : protected ProgressWrapper {...}
```

with

```
class ProgressWrapper {

public:
  ProgressWrapper(ProgressLocalInfo* p) : progress(p) {}
  ~ProgressWrapper(){}

protected:
  // the pointer address can only be assigned once, but
  // the object pointed to may be modified.
  ProgressLocalInfo* const progress;
};
```

This just adds a pointer to a `ProgressLocalInfo` to the existing `SortByLocalInfo` structure. Within the `REQUEST` branch, on the first `REQUEST`, one can then call the time consuming constructor for `SortByLocalInfo`, passing to it the pointer to the `LocalInfo` structure (here `li`):

```
li->ptr = new SortByLocalInfo( args[0], lexicographically, tupleCmp, li );
```

See the implementation of *sortby* in the file `Algebras/ExtRelation-C++/ExtRelAlgPersistent.cpp` as an example.

References

[Alm03] V. T. de Almeida, Object State Diagram in the Secondo System. Fernuniversität Hagen, Praktische Informatik IV, Secondo System, Directory “Documents”, file “ObjectStateDiagram.pdf”, Sept. 2003.

[BF93] N. Borenstein and N. Freed. MIME (Multipurpose Internet Mail Extensions) Part One: Mechanisms for Specifying and Describing the Format of Internet Message Bodies. Chapter 5.2: Base64 Content-Transfer-Encoding. Also known as RFC 1521, published Online, e.g. http://www.freesoft.org/CIE/RFC/1521/index.htm. September 1993.

[BTree02] Algebra ModuleBTreeAlgebra. Fernuniversität Hagen, Praktische Informatik IV, Secondo System, Directory “Algebras/BTree”, file “BTreeAlgebra.cpp”, since Dec. 2002.

[Date04] Algebra Module DateTimeAlgebra. Fernuniversität Hagen, Praktische Informatik IV, Secondo System, Directory “Algebras/DateTime”, file “DateAlgebra.cpp”, since April 2004.

[DG98] S. Dieker and R.H. Güting, Efficient Handling of Tuples with Embedded Large Objects. *Data & Knowledge Engineering 32 (2000)*, 247-269.

[GBA+04] Güting, R.H., T. Behr, V.T. de Almeida, Z. Ding, F. Hoffmann, and M. Spiekermann, SECONDO: An Extensible DBMS Architecture and Prototype. Fernuniversität Hagen, Informatik-Report 313, 2004.

[GFB+97] R.H. Güting, C. Freundorfer, L. Becker, S. Dieker, H. Schenk: Secondo/QP: Implementation of a Generic Query Processor. 10th Int. Conf. on Database and Expert System Applications (DEXA'99), LNCS 1677, Springer Verlag, 66-87, 1999.

[Güt02] R.H. Güting, A Query Optimizer for Secondo. Description and PROLOG Source Code. Fernuniversität Hagen, Praktische Informatik IV, Secondo System, Directory “Optimizer”, file “optimizer”, from 2002 on.

[Güt95] R.H. Güting, Integrating Programs and Documentation. Informatik Berichte 182 - 5 / 1995.

[Poly02] Algebra Module PolygonAlgebra. Fernuniversität Hagen, Praktische Informatik IV, Secondo System, Directory “Algebras/Polygon”, file “PolygonAlgebra.cpp”, since Sept. 2002.

A The Source for the InquiryViewer

```
package viewer;

import javax.swing.*;
import javax.swing.text.*;
import java.util.Vector;
import java.awt.*;
import java.awt.event.*;
import gui.SecondoObject;
import sj.lang.*;
import tools.Reporter;

public class InquiryViewer extends SecondoViewer{

  // define supported subtypes
  private static final String DATABASES = "databases";
  private static final String CONSTRUCTORS="constructors";
  private static final String OPERATORS = "operators";
  private static final String ALGEBRAS = "algebras";
  private static final String ALGEBRA = "algebra";
  private static final String TYPES = "types";
  private static final String OBJECTS ="objects";

  private JScrollPane ScrollPane = new JScrollPane();
  private JEditorPane  HTMLArea = new JEditorPane();
  private JComboBox ComboBox = new JComboBox();
  private Vector ObjectTexts = new Vector(10,5);
  private Vector SecondoObjects = new Vector(10,5);
  private SecondoObject CurrentObject=null;

  private String HeaderColor = "silver";
  private String CellColor ="white";
  private MenuVector MV = new MenuVector();
  private JTextField SearchField = new JTextField(20);
  private JButton SearchButton = new JButton("Search");
  private int LastSearchPos =0;
  private JCheckBox CaseSensitive = new JCheckBox("Case Sensitive");

  /* create a new InquiryViewer */
  public InquiryViewer(){
    setLayout(new BorderLayout());
    add(BorderLayout.NORTH,ComboBox);
    add(BorderLayout.CENTER,ScrollPane);
    HTMLArea.setContentType("text/html");
    HTMLArea.setEditable(false);
    ScrollPane.setViewportView(HTMLArea);

    JPanel BottomPanel = new JPanel();
    BottomPanel.add(CaseSensitive);
    BottomPanel.add(SearchField);
    BottomPanel.add(SearchButton);
    add(BottomPanel,BorderLayout.SOUTH);
    CaseSensitive.setSelected(true);
    ComboBox.addActionListener(new ActionListener(){
      public void actionPerformed(ActionEvent evt){
        showObject();
      }});

    SearchField.addKeyListener(new KeyAdapter(){
      public void keyPressed(KeyEvent evt){
        if( evt.getKeyCode() == KeyEvent.VK_ENTER )
          searchText();
      }});

    SearchButton.addActionListener(new ActionListener(){
      public void actionPerformed(ActionEvent evt){
        searchText();
      }});


    JMenu SettingsMenu = new JMenu("Settings");
    JMenu HeaderColorMenu = new JMenu("header color");
    JMenu CellColorMenu = new JMenu("cell color");
    SettingsMenu.add(HeaderColorMenu);
    SettingsMenu.add(CellColorMenu);

    ActionListener HeaderColorChanger = new ActionListener(){
      public void actionPerformed(ActionEvent evt){
        JMenuItem S = (JMenuItem) evt.getSource();
        HeaderColor = S.getText().trim();
        reformat();
      }
    };
    ActionListener CellColorChanger = new ActionListener(){
      public void actionPerformed(ActionEvent evt){
        JMenuItem S = (JMenuItem) evt.getSource();
        CellColor = S.getText().trim();
        reformat();
      }
    };

    HeaderColorMenu.add("white").addActionListener(HeaderColorChanger);
    HeaderColorMenu.add("silver").addActionListener(HeaderColorChanger);
    HeaderColorMenu.add("gray").addActionListener(HeaderColorChanger);
    HeaderColorMenu.add("aqua").addActionListener(HeaderColorChanger);
    HeaderColorMenu.add("blue").addActionListener(HeaderColorChanger);
    HeaderColorMenu.add("black").addActionListener(HeaderColorChanger);
    CellColorMenu.add("white").addActionListener(CellColorChanger);
    CellColorMenu.add("yellow").addActionListener(CellColorChanger);
    CellColorMenu.add("aqua").addActionListener(CellColorChanger);
    CellColorMenu.add("lime").addActionListener(CellColorChanger);
    CellColorMenu.add("silver").addActionListener(CellColorChanger);

    MV.addMenu(SettingsMenu);
  }

  /** returns the html formatted string representation for an atomic list */
  private String getStringValue(ListExpr atom){
    int at = atom.atomType();
```

```
    String res = "";
    switch(at){
      case ListExpr.NO_ATOM : return "";
      case ListExpr.INT_ATOM : return ""+atom.intValue();
      case ListExpr.BOOL_ATOM : return atom.boolValue()?"TRUE":"FALSE";
      case ListExpr.REAL_ATOM : return ""+atom.realValue();
      case ListExpr.STRING_ATOM: res = atom.stringValue();break;
      case ListExpr.TEXT_ATOM: res = atom.textValue();break;
      case ListExpr.SYMBOL_ATOM: res = atom.symbolValue();break;
      default : return "";
    }

    res = replaceAll("&",res,"&amp");
    res = replaceAll("<",res,"&lt;");
    res = replaceAll(">",res,"&gt;");
    return res;
  }


  /** include for using older Java-versions */
  private static String replaceAll(String what, String where,
                                   String ByWhat){
    StringBuffer res = new StringBuffer();
    int lastpos = 0;
    int len = what.length();
    int index = where.indexOf(what,lastpos);
    while(index>=0){
      if(index>0)
        res.append(where.substring(lastpos,index));
      res.append(ByWhat);
      lastpos = index+len;
      index = where.indexOf(what,lastpos);
    }
    res.append(where.substring(lastpos));
    return res.toString();
  }

  /** searchs the text in the textfield in the document and
    * marks its if found
    */
  private void searchText(){
    String Text = SearchField.getText();
    if(Text.length()==0){
      Reporter.showInfo("no text to search");
      return;
    }
    try{
      Document Doc = HTMLArea.getDocument();
      String DocText = Doc.getText(0,Doc.getLength());
      if(!CaseSensitive.isSelected()){
        DocText = DocText.toUpperCase();
        Text = Text.toUpperCase();
      }
      int pos = DocText.indexOf(Text,LastSearchPos);
      if(pos<0){
        Reporter.showInfo("end of text is reached");
        LastSearchPos=0;
        return;
      }
      pos = pos;
      int i1 = pos;
      int i2 = pos+Text.length();
      LastSearchPos = pos+1;
      HTMLArea.setCaretPosition(i1);
      HTMLArea.moveCaretPosition(i2);
      HTMLArea.getCaret().setSelectionVisible(true);
    } catch(Exception e){
      Reporter.debug(e);
      Reporter.showError("error in searching text");
    }
  }

  /** returns the html string for a single entry for
    * type constructors or operators
    */
  private String formatEntry(ListExpr LE){
    if(LE.listLength()!=3){
      Reporter.writeError("InquiryViewer : error in list"+
                          "(listLength() # 3");
      return "";
    }
    ListExpr Name = LE.first();
    ListExpr Properties = LE.second();
    ListExpr Values = LE.third();
    if(Properties.listLength()!= Values.listLength()){
      Reporter.writeWarning("InquiryViewer : Warning: lists "+
            "have different lengths ("+Name.symbolValue()+")");
    }

    String res =" <tr><td class=\"opname\" colspan=\"2\">" +
    Name.symbolValue() + "</td></tr>\n";
    while( !Properties.isEmpty() & ! Values.isEmpty()){
      res = res + "   <tr><td class=\"prop\">" +
            getStringValue(Properties.first())+"</td>" +
            "<td class=\"value\">" +
            getStringValue(Values.first())+"</td></tr>\n";
      Properties = Properties.rest();
      Values = Values.rest();
    }

    // handle non empty lists
    // if the lists are correct this never should occur
    while( !Properties.isEmpty()){
      res = res + "   <tr><td class=\"prop\">" +
            getStringValue(Properties.first())+"</td>" +
            "<td>   </td></tr>\n";
      Properties = Properties.rest();
    }
    while(!Values.isEmpty()){
        res = res + "   <tr><td> </td>" +
              "<td class=\"value\">" +
              getStringValue(Values.first())+"</td></tr>\n";
      Values = Values.rest();
    }

    return res;
  }


  /** create the html head for text representation
    * including the used style sheet
    */
  private String getHTMLHead(){
    StringBuffer res = new StringBuffer();
    res.append("<html>\n");
    res.append("<head>\n");
    res.append("<title> inquiry viewer </title>\n");
    res.append("<style type=\"text/css\">\n");
    res.append("<!--\n");
    res.append("td.opname { background-color:"+HeaderColor+
                "; font-family:monospace;"+
                "font-weight:bold; "+
                "color:green; font-size:x-large;}\n");
    res.append("td.prop  {background-color:"+CellColor+
          "; font-family:monospace; font-weight:bold; color:blue}\n");
    res.append("td.value {background-color:"+CellColor+
          "; font-family:monospace; color:black;}\n");
    res.append("-->\n");
    res.append("</style>\n");
    res.append("</head>\n");
    return res.toString();
  }


  /** get the html formatted html Code for type contructors
    */
  private String getHTMLCode_Constructors(ListExpr ValueList){
    StringBuffer res = new StringBuffer();
    if(ValueList.isEmpty())
      return "no type constructors are defined <br>";
    res.append("<table border=\"2\">\n");
    while(!ValueList.isEmpty() ){
      res.append(formatEntry(ValueList.first()));
      ValueList = ValueList.rest();
    }
    res.append("</table>\n");
    return res.toString();
  }

  /** returns the html-code for  operators */
  private String getHTMLCode_Operators(ListExpr ValueList){
    if(ValueList.isEmpty())
      return "no operators are defined <br>";
    // the format is the same like for constructors
    return getHTMLCode_Constructors(ValueList);
  }
```

```
/** returns the html for an Algebra List */
private String getHTMLCode_Databases(ListExpr Value){
  // the valuelist for algebras is just a list containing
  // symbols representing the database names
  if(Value.isEmpty())
    return "no database exists <br>";
  StringBuffer res = new StringBuffer();
  res.append("<ul>\n");
  while (!Value.isEmpty()){
    res.append("<li> "+Value.first().symbolValue() + " </li>");
    Value = Value.rest();
  }
  res.append("</ul>");
  return res.toString();
}

/** returns the html code for objects */
private String getHTMLCode_Objects(ListExpr Value){
  ListExpr tmp = Value.rest(); // ignore  "SYMBOLS"
  if(tmp.isEmpty())
    return "no existing objects";
  StringBuffer res = new StringBuffer();
  res.append("<h2> Objects - short list </h2>\n ");
  res.append("<ul>\n");
  while(!tmp.isEmpty()){
    res.append("  <li>"+tmp.first().second().symbolValue()+ " </li> \n");
    tmp = tmp.rest();
  }
  res.append("</ul><br><hr><br>");
  res.append("<h2> Objects - full list </h2>\n");
  res.append("<pre>\n"+Value.rest().writeListExprToString() +"</pre>");
  return res.toString();
}

/** returns the html code for types */
private String getHTMLCode_Types(ListExpr Value){
  ListExpr tmp = Value.rest(); // ignore  "TYPES"
  if(tmp.isEmpty())
    return "no existing type";
  StringBuffer res = new StringBuffer();
  res.append("<h2> Types - short list </h2>\n ");
  res.append("<ul>\n");
  while(!tmp.isEmpty()){
    res.append("  <li>"+tmp.first().second().symbolValue()+ " </li> \n");
    tmp = tmp.rest();
  }
  res.append("</ul><br><hr><br>");
  res.append("<h2> Types - full list </h2>\n");
  res.append("<pre>\n"+Value.rest().writeListExprToString() +"</pre>");
  return res.toString();
}

/** returns a html formatted list for algebras */
private String getHTMLCode_Algebras(ListExpr Value){
  // use the same format like databases
  if(Value.isEmpty())
```

```
    return "no algebra is included <br> please check"+
            " your Secondo installation <br>";
  return getHTMLCode_Databases(Value);
}

/** returns the formatted html code for a algebra inquiry */
private String getHTMLCode_Algebra(ListExpr Value){
  // the format is
  // (name ((constructors) (operators)))
  // where constructors and operators are formatted like in the
  // non algebra version
  StringBuffer res = new StringBuffer();
  res.append("<h1> Algebra "+Value.first().symbolValue()+" </h1>\n");
  res.append("<h2> type constructors of algebra: "+
             Value.first().symbolValue()+" </h2>\n");
  res.append( getHTMLCode_Constructors(Value.second().first()));
  res.append("<br>\n<h2> operators of algebra: "+
             Value.first().symbolValue()+"</h2>\n");
  res.append( getHTMLCode_Operators(Value.second().second()));
  return res.toString();
}

/** returns the html code for a given list */
private String getHTMLCode(ListExpr VL){
  StringBuffer Text = new StringBuffer();
  Text.append(getHTMLHead());
  Text.append("<body>\n");
  String inquiryType = VL.first().symbolValue();
  if (inquiryType.equals(DATABASES)){
    Text.append("<h1> Databases </h1>\n");
    Text.append(getHTMLCode_Algebras(VL.second()));
  }
  else if(inquiryType.equals(ALGEBRAS)){
    Text.append("<h1> Algebras </h1>\n");
    Text.append(getHTMLCode_Algebras(VL.second()));
  } else if(inquiryType.equals(CONSTRUCTORS)){
    Text.append("<h1> Type Constructors </h1>\n");
    Text.append(getHTMLCode_Constructors(VL.second()));
  } else if(inquiryType.equals(OPERATORS)){
    Text.append("<h1> Operators </h1>\n");
    Text.append(getHTMLCode_Operators(VL.second()));
  } else if(inquiryType.equals(ALGEBRA)){
    Text.append(getHTMLCode_Algebra(VL.second()));
  } else if(inquiryType.equals(OBJECTS)){
    Text.append("<h1> Objects </h1>\n");
    Text.append(getHTMLCode_Objects(VL.second()));
  } else if(inquiryType.equals(TYPES)){
    Text.append("<h1> Types </h1>\n");
    Text.append(getHTMLCode_Types(VL.second()));
  }
  Text.append("\n</body>\n</html>\n");
  return Text.toString();
}
```

```
/* adds a new Object to this Viewer and display it */
public boolean addObject(SecondoObject o){
  if(!canDisplay(o))
    return false;
  if (isDisplayed(o))
    selectObject(o);
  else{
    ListExpr VL = o.toListExpr().second();
    ObjectTexts.add(getHTMLCode(VL));
    ComboBox.addItem(o.getName());
    SecondoObjects.add(o);
    try{
      ComboBox.setSelectedIndex(ComboBox.getItemCount()-1);
      showObject();
    } catch(Exception e){
      Reporter.debug(e);
    }
  }
  return true;
}

/** write all htmls texts with a new format */
private void reformat(){
  int index = ComboBox.getSelectedIndex();
  ObjectTexts.removeAllElements();
  for(int i=0;i<SecondoObjects.size();i++){
    SecondoObject o = (SecondoObject) SecondoObjects.get(i);
    ListExpr VL = o.toListExpr().second();
    String inquiryType = VL.first().symbolValue();
    ObjectTexts.add(getHTMLCode(VL));
  }
  if(index>=0)
    ComboBox.setSelectedIndex(index);
}

/* returns true if o a SecondoObject in this viewer */
public boolean isDisplayed(SecondoObject o){
  return SecondoObjects.indexOf(o)>=0;
}

/** remove o from this Viewer */
public void removeObject(SecondoObject o){
  int index = SecondoObjects.indexOf(o);
  if(index>=0){
    ComboBox.removeItem(o.getName());
    SecondoObjects.remove(index);
    ObjectTexts.remove(index);
  }
}

/** remove all containing objects */
public void removeAll(){
  ObjectTexts.removeAllElements();
  ComboBox.removeAllItems();
  SecondoObjects.removeAllElements();
  CurrentObject= null;
  if(VC!=null)
    VC.removeObject(null);
  showObject();
}

/** check if this viewer can display the given object */
public boolean canDisplay(SecondoObject o){
  ListExpr LE = o.toListExpr();
  if(LE.listLength()!=2)
    return false;
  if(LE.first().atomType()!=ListExpr.SYMBOL_ATOM ||
    !LE.first().symbolValue().equals("inquiry"))
    return false;
  ListExpr VL = LE.second();
  if(VL.listLength()!=2)
    return false;
  ListExpr SubTypeList = VL.first();
  if(SubTypeList.atomType()!=ListExpr.SYMBOL_ATOM)
    return false;
  String SubType = SubTypeList.symbolValue();
  if(SubType.equals(DATABASES) || SubType.equals(CONSTRUCTORS) ||
    SubType.equals(OPERATORS) || SubType.equals(ALGEBRA) ||
    SubType.equals(ALGEBRAS)  || SubType.equals(OBJECTS) ||
    SubType.equals(TYPES))
      return true;
  return false;
}

/** returns the Menuextension of this viewer */
public MenuVector getMenuVector(){
  return MV;
}

/** returns InquiryViewer */
public String getName(){
  return "InquiryViewer";
}

public double getDisplayQuality(SecondoObject SO){
  if(canDisplay(SO))
    return 0.9;
  else
    return 0;
}

/* select O */
public boolean selectObject(SecondoObject O){
  int i=SecondoObjects.indexOf(O);
  if (i>=0) {
    ComboBox.setSelectedIndex(i);
    showObject();
    return true;
  }else //object not found
    return false;
}
```

```
  private void showObject(){
    String Text="";
    int index = ComboBox.getSelectedIndex();
    if (index>=0){
      HTMLArea.setText((String)ObjectTexts.get(index));
    } else {
      // set an empty text
      HTMLArea.setText(" <html><head></head><body></body></html>");
    }
    LastSearchPos = 0;
  }
}
```

B The Source for Dsplmovingpoint

```
package  viewer.hoese.algebras;

import java.awt.geom.*;
import java.awt.*;
import viewer.*;
import viewer.hoese.*;
import sj.lang.ListExpr;
import java.util.*;
import gui.Environment;
import tools.Reporter;


/**
  * A displayclass for the movingpoint-type (spatiotemp algebra),
  */
public class Dsplmovingpoint extends DisplayTimeGraph
                              implements LabelAttribute, RenderAttribute {
  Point2D.Double point;
  Vector PointMaps;
  Rectangle2D.Double bounds;
  double minValue = Integer.MAX_VALUE;
  double maxValue = Integer.MIN_VALUE;
  boolean defined;
  static java.text.DecimalFormat format =
                          new java.text.DecimalFormat("#.#####");

  public int numberOfShapes(){
    return 1;
  }

  /** Returns a short text usable as label **/
  public String getLabel(double time){
    if(Intervals==null || PointMaps==null){
      return null;
    }
    int index = IntervalSearch.getTimeIndex(time,Intervals);
    if(index<0){
      return null;
    }
    PointMap pm = (PointMap) PointMaps.get(index);
    Interval in = (Interval)Intervals.get(index);
    double t1 = in.getStart();
    double t2 = in.getEnd();
    double Delta = (time-t1)/(t2-t1);
    double x = pm.x1+Delta*(pm.x2-pm.x1);
    double y = pm.y1+Delta*(pm.y2-pm.y1);
    return "("+format.format(x)+", "+ format.format(y)+")";
  }

  /**
    * Gets the shape of this instance at the ActualTime
    * @param at The actual transformation,
    * used to calculate the correct size.
    * @return Rectangle or Circle Shape if ActualTime is defined
    *otherwise null.
    * @see <a href="Dsplmovingpointsrc.html#getRenderObject">Source</a>
    */
  public Shape getRenderObject (int num,AffineTransform at) {
    if(num!=0){
      return null;
    }
    if(Intervals==null || PointMaps==null){
      return null;
    }
    if(RefLayer==null){
      return null;
    }
    double t = RefLayer.getActualTime();
    int index = IntervalSearch.getTimeIndex(t,Intervals);
    if(index<0){
      return null;
    }

    PointMap pm = (PointMap) PointMaps.get(index);
    Interval in = (Interval)Intervals.get(index);
    double t1 = in.getStart();
    double t2 = in.getEnd();
    double Delta = (t-t1)/(t2-t1);
    double x = pm.x1+Delta*(pm.x2-pm.x1);
    double y = pm.y1+Delta*(pm.y2-pm.y1);

    point = new Point2D.Double(x, y);
    double ps = Cat.getPointSize(renderAttribute,CurrentState.ActualTime);
    double pixy = Math.abs(ps/at.getScaleY());
    double pix = Math.abs(ps/at.getScaleX());
    Shape shp;
    if (Cat.getPointasRect())
      shp = new Rectangle2D.Double(point.getX()- pix/2,
                                    point.getY() - pixy/2, pix, pixy);
    else {
      shp = new Ellipse2D.Double(point.getX()- pix/2,
                                  point.getY() - pixy/2, pix, pixy);
    }
    return  shp;
  }

  /**
    * Reads the coefficients out of ListExpr for a map
    * @param le ListExpr of four reals.
    * @return The PointMap that was read.
    * @see <a href="Dsplmovingpointsrc.html#readPointMap">Source</a>
    */
  private PointMap readPointMap (ListExpr le) {
    Double value[] =  {null, null, null, null};
    if (le.listLength() != 4)
      return  null;
    for (int i = 0; i < 4; i++) {
      value[i] = LEUtils.readNumeric(le.first());
      if (value[i] == null)
        return  null;
      le = le.rest();
    }
    double x1, y1;

    double v0 = value[0].doubleValue();
    double v1 = value[1].doubleValue();
    double v2 = value[2].doubleValue();
    double v3 = value[3].doubleValue();
    if(minValue>v0) minValue=v0;
    if(maxValue<v0) maxValue=v0;
    if(minValue>v2) minValue=v2;
    if(maxValue<v2) maxValue=v2;

    if(!ProjectionManager.project(value[0].doubleValue(),
                  value[1].doubleValue(),aPoint)){
      return null;
    }
    x1 = aPoint.x;
    y1 = aPoint.y;
    if(!ProjectionManager.project(value[2].doubleValue(),
            value[3].doubleValue(),aPoint)){
      return null;
    }
    return  new PointMap(x1,y1,aPoint.x,aPoint.y);
  }

  /**
    * Scans the representation of a movingpoint datatype
    * @param v A list of start and end intervals with ax,bx,ay,by values
    * @see sj.lang.ListExpr
    * @see <a href="Dsplmovingpointsrc.html#ScanValue">Source</a>
    */
  private void ScanValue (ListExpr v) {
    err = true;
    if (v.isEmpty()){ //empty point
      Intervals=null;
      PointMaps=null;
      err=false;
      defined = false;
      return;
    }
    while (!v.isEmpty()) {      // unit While maybe empty
      ListExpr aunit = v.first();
      ListExpr tmp = aunit;
      int L = aunit.listLength();
      if(L!=2 && L!=8){
        Reporter.debug("wrong ListLength in reading moving point unit");
        defined = false;
        return;
      }
      // deprecated version of external representation
      Interval in=null;
      PointMap pm=null;
      if (L == 8){
        Reporter.writeWarning("Warning: using deprecated external"+
                              " representation of a moving point !");
        in = LEUtils.readInterval(ListExpr.fourElemList(aunit.first(),
                      aunit.second(), aunit.third(), aunit.fourth()));
        aunit = aunit.rest().rest().rest().rest();
        pm = readPointMap(ListExpr.fourElemList(aunit.first(),
                                aunit.second(), aunit.third(),
                                aunit.fourth()));
      }
      // the corrected version of external representation
      if(L==2){
        in = LEUtils.readInterval(aunit.first());
        pm = readPointMap(aunit.second());
      }

      if ((in == null) || (pm == null)){
        Reporter.debug("Error in reading Unit");
        Reporter.debug(tmp.writeListExprToString());
        if(in==null){
          Reporter.debug("Error in reading interval");
        }
        if(pm==null){
          Reporter.debug("Error in reading Start and EndPoint");
        }
        defined = false;
        return;
      }
      Intervals.add(in);
      PointMaps.add(pm);
      v = v.rest();
    }
    err = false;
    defined = true;
  }

  public boolean isPointType(int num){
    return true;
  }

  /**
    * Init. the Dsplmovingpoint instance and calculate the overall
    * bounds and Timebounds
    * @param type The symbol movingpoint
    * @param value A list of start and end intervals with ax,bx,ay,by values
    * @param qr queryresult to display output.
    * @see sj.lang.ListExpr
    */
  public void init (ListExpr type, ListExpr value, QueryResult qr) {
    AttrName = type.symbolValue();
    int length = value.listLength();
    Intervals = new Vector(length+2);
    PointMaps = new Vector(length+2);
    ScanValue(value);
    if (err) {
      Reporter.writeError("Dsplmovingpoint Error in ListExpr :"+
          "parsing aborted");
      qr.addEntry(new String("(" + AttrName + ": GTA(mpoint))"));
      return;
    }
    else
      qr.addEntry(this);
    //ListIterator li=iv.listIterator();
    bounds = null;
    TimeBounds = null;
    if(Intervals==null) // empty moving point
      return;
    for (int j = 0; j < Intervals.size(); j++) {
      Interval in = (Interval)Intervals.elementAt(j);
      PointMap pm = (PointMap)PointMaps.elementAt(j);
      Rectangle2D.Double r = new Rectangle2D.Double(pm.x1,pm.y1,0,0);
      r = (Rectangle2D.Double)r.createUnion(
                  new Rectangle2D.Double(pm.x2,pm.y2,0,0));
      if (bounds == null) {
        bounds = r;
        TimeBounds = in;
      } else {
        bounds = (Rectangle2D.Double)bounds.createUnion(r);
        TimeBounds = TimeBounds.union(in);
      }
    }
  }

  /**
    * @return The overall boundingbox of the movingpoint
    * @see <a href="Dsplmovingpointsrc.html#getBounds">Source</a>
    */
  public Rectangle2D.Double getBounds () {
    return  bounds;
  }

  /** returns the minimum x value **/
  public double getMinRenderValue(){
    return minValue;
  }
  /** returns the maximum x value **/
  public double getMaxRenderValue(){
    return maxValue;
  }
  /** returns the current x value **/
  public double getRenderValue(double time){
    if(Intervals==null || PointMaps==null){
      return 0;
    }
    int index = IntervalSearch.getTimeIndex(time,Intervals);
    if(index<0){
      return 0;
    }
    PointMap pm = (PointMap) PointMaps.get(index);
    Interval in = (Interval)Intervals.get(index);
```

```
    double t1 = in.getStart();
    double t2 = in.getEnd();
    double Delta = (time-t1)/(t2-t1);
    double x = pm.x1+Delta*(pm.x2-pm.x1);
    return  x;
  }

  public boolean mayBeDefined(){
    return defined;
  }

  public boolean isDefined(double time){
    if(!defined){
      return false;
    }
    int index = IntervalSearch.getTimeIndex(time,Intervals);
    return index>=0;
  }

  class PointMap {
    double x1,x2,y1,y2;

    public PointMap (double x1, double y1, double x2, double y2) {
      this.x1 = x1;
      this.y1 = y1;
      this.x2 = x2;
      this.y2 = y2;
    }

    public String toString(){
      return ("[x1,y1 | x2,y2] = ["+x1+","+y1+" <> "+x2+","+y2+"]");
    }
  }
}
```

C The Source for Algebra Module PointRectangle