Secondo User Manual*

Version 3, September 21, 2004

Ralf Hartmut Güting, Dirk Ansorge, Thomas Behr, Markus Spiekermann

Praktische Informatik IV, Fernuniversität Hagen

D-58084 Hagen, Germany

(*) This work was partially supported by a grant Gu 293/8-1 from the Deutsche Forschungsgemeinschaft (DFG), project “Datenbanken für bewegte Objekte” (Databases for Moving Objects).

Table of Contents

1 Introduction and Overview 1

2 Command Syntax 5
2.1 Overview 5
2.2 Nested List Syntax 5
2.3 User Level Syntax 6

3 Secondo Commands 10
3.1 Basic Commands 10
3.2 Inquiries 11
3.3 Databases 12
3.4 Transactions 12
3.5 Import and Export 12
3.6 Database States 13

4 Algebra Module Configuration 14
4.1 Activating Algebra Modules 14
4.2 Invoking the TestRunner 14

5 User Interfaces 15
5.1 Overview 15
5.2 Single-Threaded User Interfaces 15
5.3 Multi-User Operation 16

6 Algebra Modules 27
6.1 Overview 27
6.2 Standard Algebra (Standard-C++) 27
6.3 Relation Algebra (Relation-C++) 29

7 Functions and Function Objects 34

8 The Optimizer 36
8.1 Preparations 36
8.2 Using SECONDO in a PROLOG environment 36
8.3 An SQL-like Query Language 38
8.4 Further Ways of Querying 42
8.5 The Optimizer's Knowledge of Databases 43
8.6 Operator Syntax 44

A Operator Syntax 46

B Grammar of the Query Language 47

C References 48

1 Introduction and Overview

The goal of SECONDO is to provide a "generic" database system frame that can be filled with implementations of various DBMS data models. For example, it should be possible to implement relational, object-oriented, temporal, or XML models and to accomodate data types for spatial data, moving objects, chemical formulas, etc. Whereas extensibility by data types is common now (e.g. as data blades, cartridges, etc.), the possibility to change the core data model is rather special to SECONDO.

The strategy to achieve this goal is the following:

- Separate the data model independent components and mechanisms in a database system (the *system frame*) from the data model dependent parts.
- Provide a formalism to describe the implemented data model, in order to be able to provide clean interfaces between system frame and "contents". This formalism is *second-order signature*, explained below.
- Structure the implementation of a data model into a collection of algebra modules, each providing specific data structures and operations.

SECONDO was intended originally as a platform for implementing and experimenting with new kinds of data models, especially to support spatial, spatio-temporal, and graph database models. We now feel, SECONDO has a clean architecture, and it strikes a reasonable balance between simplicity and sophistication. In addition, the central parts are well documented, with a technique developed specifically for this system (the so-called PD system). Since all the source code is accessible and to a large extent comprehensible for students, we believe it is also an excellent tool for teaching database architecture and implementation concepts.

SECONDO uses BerkeleyDB as a storage manager, runs on Windows, Linux, and Solaris platforms, and consists of three major components written in different languages:

- The SECONDO kernel implements specific data models, is extensible by algebra modules, and provides query processing over the implemented algebras. It is implemented on top of BerkeleyDB and written in C++.
- The optimizer provides as its core capability conjunctive query optimization, currently for a relational environment. Conjunctive query optimization is, however, needed for any kind of data models. In addition, it implements the essential part of SQL-like languages, in a notation adapted to PROLOG. The optimizer is written in PROLOG.
- The graphical user interface (GUI) is on the one hand an extensible interface for an extensible DBMS such as SECONDO. It is extensible by *viewers* for new data types or models. On the other hand, there is a specialized viewer available in the GUI for spatial types and moving objects, providing a generic and rather sophisticated spatial database interface, including animation of moving objects. The GUI is written in Java.

The three components can be used together or independently, in several ways. The SECONDO kernel can be used as a single user system or in a client-server mode. As a stand-alone system, it can be linked together with either a simple command interface running in a shell, or with the optimizer. In client-server mode, the kernel can serve clients running the command interface, an optimizer client, or the GUI. The optimizer can be used separately to transform SQL-like queries into

query plans that would be executable in SECONDO. The GUI can be used separately to browse spatial or spatio-temporal data residing in files. All three components can be used together in a configuration shown in Figure 1.

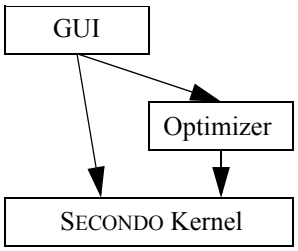

Figure 1: Cooperation of SECONDO Components

In this configuration, the GUI can ask the kernel directly to execute commands and queries (queries written as query plans, i.e., terms of the implemented algebras). Or it can call the optimizer to get a plan for a given SQL query. The optimizer when necessary calls the SECONDO kernel to get information about relation schemas, cardinalities of relations, and selectivity of predicates. Here the optimizer acts as a server for the GUI and as a client to the kernel.

A very rough description of the architecture of the SECONDO kernel is shown in Figure 2. A data model is implemented as a set of data types and operations. These are grouped into *algebras*.

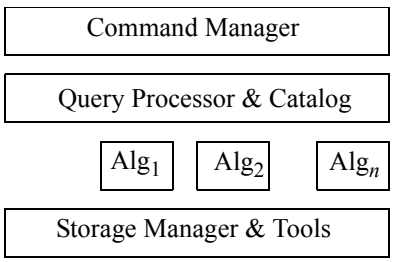

Figure 2: Rough architecture of the kernel

The definition of algebras is based on the concept of *second-order signature* [Gü93]. The idea is to use two coupled signatures. Any signature provides sorts and operations. Here in the first signature the sorts are called *kinds* and represent collections of types. The operations of this signature are *type constructors*. The signature defines how type constructors can be applied to given types. The available types in the system are exactly the terms of this signature.

The second signature defines operations over the types of the first signature.

An algebra module provides a collection of type constructors, implementing a data structure for each of them. A small set of support functions is needed to register a type constructor within an algebra. Similarly, the algebra module offers operators, implementing support functions for them such as type mapping, evaluation, resolution of overloading, etc.

The query processor evaluates queries by building an operator tree and then traversing it, calling operator implementations from the algebras. The framework allows algebra operations to have parameter functions and to handle streams. More details can be found in [DG00].

The SECONDO kernel manages *databases*. A database is a set of SECONDO objects. A SECONDO object is a triple of the form (*name*, *type*, *value*) where *type* is a type term of the implemented

algebras and *value* a value of this type. Databases can be created, deleted, opened, closed, exported to and imported from files. In files they are represented as nested lists (like in LISP) in a text format.

On an open database, there are some basic commands available:

```
type <ident> = <type expr>
delete type <ident>
create <ident>: <type expr>
update <ident> := <value expr>
let <ident> = <value expr>
delete <ident>
query <value expr>
```

Obviously, the type expressions and value expressions are defined over the implemented algebras. Note that *create* creates an object whose value is yet undefined. *Let* creates an object whose type is defined by the given value, so one does not need to specify the type.

The kernel offers further commands for inspection of the implemented system (`list type constructors, list operators, list algebras, list algebra <algebraName>`), the available databases (`list databases`), or the contents of the open database (`list types, list objects`). Objects can also be exported into and restored from files. Finally there are commands for transaction management.

Currently there exist about twenty algebras implemented within SECONDO. All algebras include appropriate operations. Some examples are:

- StandardAlgebra. Provides data types int, real, bool, string.
- RelationAlgebra. Relations with all operations needed to implement an SQL-like relational language.
- BTreeAlgebra. B-Trees.
- RTreeAlgebra. R-Trees.
- SpatialAlgebra. Spatial data types point, points, line, region.
- DateAlgebra. A small algebra providing a *date* type.
- MidiAlgebra. Providing a data type to represent the contents of Midi files including interesting operations like searching for a particular sequence of keys.

The following examples are based on these algebras. Here are some commands:

```
create x: int
update x := 7
let inc = fun(n:int) n + 1
query "secondo" contains "second"
```

A more complex example involves some SECONDO objects in the open database: (i) a relation `Kreis` with type (schema) `rel(tuple([KName: string, ..., Gebiet: region]))` containing the regions of 439 counties (“Kreise”) in Germany, (ii) an object `magdeburg` of type region, containing the geometry of county “Magdeburg”, and (iii) an object `kreis_Gebiet` of type `rtree(tuple([KName: string, ..., Gebiet: region]))` which is an R-tree on the `Gebiet` attribute of relation `Kreis`.

The following query finds neighbour counties of `magdeburg`:

```
query kreis_Gebiet Kreis
  windowintersects[bbox(magdeburg)]
  filter[.Gebiet touches magdeburg]
  filter[not(.KName contains "Magdeburg")]
  project[KName] consume
```

The query uses the R-tree index to find tuples for which the bounding box (MBR) of the `Gebiet` attribute overlaps with the bounding box of the `magdeburg` region. The qualifying stream of tuples is filtered by the condition that the region of the tuple is indeed adjacent (“touches”) the region of `magdeburg` and then by a further condition eliminating the county “Magdeburg” itself. Tuples are then projected on their `KName` attribute and the stream is collected into a result relation.

The following sections of this manual describe the use of SECONDO in detail.

2 Command Syntax

2.1 Overview

SECONDO offers a fixed set of commands for database management, catalog inquiries, access to types and objects, queries, and transaction control. Some of these commands require type expression, value expression, or identifier arguments. Whether a type expression or value expression is valid or not is determined by means of the specifications provided by the active algebra modules, while validity of an identifier depends on the content of the actual database.

SECONDO accepts two different forms of user input: queries in nested list syntax and queries following a syntax defined by the active algebra modules. Both forms have positive and negative aspects. On the one hand, nested list syntax remains the same, regardless of the actual set of operators provided by active algebra modules. On the other hand, queries in nested list syntax tend to contain a lot of parentheses, thereby getting hard to formulate and read. This is the motivation for offering a second level of query syntax with two important features:

- Reading and writing type expressions is simplified.
- For each operator of an algebra module, the algebra implementor can specify syntax properties like infix or postfix notation. If this feature is used carefully, value expressions can be much more understandable.

2.2 Nested List Syntax

Using nested list syntax, each command is a single nested list. For short, the textual representation of a nested list consists of a left parenthesis, followed by an arbitrary number of elements, terminated by a right parenthesis. Elements are either nested lists again or atomic elements like numbers, symbols, etc. The list expression `(a b ((c)(d e)))` represents a nested list of 3 elements: `a` is the first element, `b` is the second one, and `((c)(d e))` is the third one. Thus the third element of the top-level list in turn is a nested list. Its two elements are again nested lists, the first one consisting of the single element `c`, the other one containing the two elements `d` and `e`.

Since a single user command must be given as a single nested list, a command like `list type constructors` has to be transformed to a nested list before it can be passed to the system: `(list type constructors)`. In addition to commands with fixed contents, there are also commands containing identifiers, type expressions, and value expressions. While identifiers are restricted to be atomic symbols, type expressions and value expressions may either be atomic symbols or nested lists again.

For instance, assuming there are type constructors `rel` and `tuple` and attribute data types `int` and `string`, the nested list term `(rel (tuple ((name string)(pop int))))` is a valid type expression, defining a relation type consisting of tuples whose attributes are called `name` of type `string` and `pop` of type `int`. Additionally SECONDO supports the definition of new types. Consider the new type `cityrel` defined as `(rel (tuple ((name string)(pop int))))`: Now the symbol

`cityrel` is a valid type expression, too. Writing `cityrel` has exactly the same effect as writing its complete definition.

Value expressions are *constants*, object *names*, or *terms* of the query algebra defined by the current collection of active algebra modules. Constants, in general, are two-element lists of the form `(<type> <value>)`. For standard data types (`int`, `real`, `bool`, `string`) just giving the value is sufficient:

```
17, 3.14159, TRUE, "Secondo"
```

Thus, `5`, `cities`, `(+ 4 5)`, `(head (feed cities) 4)`, or the constant relation

```
( (rel (tuple ((name string) (pop int)))
  (("New York" 7322000) ("Paris" 2175000) ("Hagen" 212000)))
```

are valid value expressions, provided an object with name `cities` and appropriate operators `feed` and `head` exist. Prefix notation is mandatory for specifying operator application in nested list syntax.

The following value expression demonstrates another general concept for query formulation: anonymous function definition in nested list syntax. Its main purpose is to define predicate functions for operators like `select`, taking a relation and a boolean function as parameter.[1] The function is applied to each tuple and returns `true` if the result of `select` shall contain the tuple, otherwise `false`.

```
(select cities (fun (c city) ( > (attr c pop ) 500000)))
```

An anonymous function definition is a list whose first element is the keyword `fun`. The next elements are lists of two elements, introducing function parameter names and types. The last element of the top-level list is the function body. In the example, the type `city` might be defined as `(tuple (city string)(pop int)))`. The operator `attr` extracts from the tuple passed as first argument the value of the attribute whose name is given in the second argument. Thus, the anonymous function determines for a tuple of type `city` whether its `pop` value is greater than 500000 or not. The `select` operator applies this function to each tuple in `cities` and keeps only those for which `true` is returned.

2.3 User Level Syntax

The user level syntax, also called *text syntax*, is more comfortable to use. It is implemented by a tool called the SECONDO *parser* which just transforms textual commands into the nested list format in which they are then passed to execution. This parser is not aware of the contents of a database; so any errors with respect to a database (e.g. objects referred to do not exist) are only discovered at the next level, when lists are processed. However, the parser knows the SECONDO commands described in Section 2.3; it implements a fixed set of notations for type expressions, and it also implements for each operator of an active algebra a specific syntax defined by the algebra implementor.

1. We mention `select` here because it is a well-known operation of the relational algebra; however it is at the *descriptive* algebra level [Gü93] and not implemented in the current SECONDO system

2.3.1 Commands

Commands can be written without parentheses, for example

```
list type constructors
query cities
```

which is translated to

```
(list type constructors)
(query cities)
```

2.3.2 Constants

Constants are written in text syntax in the form

```
[const <type expression> value <value expression>]
```

This is translated to the list

```
(<type expression> <value expression>)
```

which is the list form of constants explained above. Of course, simple constants for integers etc. can be written directly. For example,

```
[const int value 5]
5
[const rectangle value (12.0 16.0 2.5 50.0)]
```

might be notations for constants.

2.3.3 Type Expressions

Type constructors can be written in prefix notation, that is

```
<type constructor>(<arg_1>, ..., <arg_n>)
```

This is translated into the nested list format

```
(<type constructor> <arg_1> ... <arg_n>)
```

Example type expressions are

```
array(int)
rel(tuple([name: string, pop: int]))
```

The second example uses notations for lists and pairs

```
[elem_1, ..., elem_n]
x: y
```

translated by the parser into

```
(elem_1 ... elem_n)
(x y)
```

So the expression `rel(tuple([name: string, pop: int]))` is transformed into the nested list form shown in Section 2.2. The relation constant can now be written as

```
[const rel(tuple([name: string, pop: int]))
value (("New York" 7322000) ("Paris" 2175000) ("Hagen" 212000))]
```

2.3.4 Value Expressions

Value expressions are terms consisting of operator applications to database objects or constants. For each operator a specific syntax can be defined. For an algebra *Alg* this is done in a file called *Alg.spec* which can be found in the directory of this algebra. The parser is built at compile time taking these specifications into account. The syntax for an operator can be looked up in a running system by one of the commands

```
list operators
list algebra <algebra name>
```

These commands provide further information such as the meaning of the operator. For example, the current last entry appearing on `list operators` is

```
     Name: year
Signature: (date) -> int
   Syntax: year ( _ )
  Meaning: extract the year info. from a date.
  Example: query year ( date1 )
```

This specifies prefix syntax for the operator `year`. The entry in the `DateAlgebra.spec` file is a bit more technical:

```
operator year alias YEAR pattern op ( _ )
```

Be aware that what appears in a listing is a comment written by the algebra implementor which may occasionally be wrong; the parser uses the specification from the `.spec` file.

Some example syntax patterns for the operators mentioned in Section 2.2 are

```
+:          _ # _
>:          _ # _
attr:       # ( _ , _)
feed:       _ #
head:       _ # [ _ ]
select:     _ # [ _ ]
```

where # denotes the operator, _ an argument, and parentheses, comma, or semicolon have to be put as shown. Based on that, we can write queries (value expressions)

```
4 + 5
cities feed head[4]
cities select[fun(c: city) attr(c, pop) > 500000]
```

In Section 6 the syntax for many operations of the standard algebra and the relation algebra is shown.

Abbreviations for Writing Parameter Functions

The expression

```
cities select[fun(c: city) attr(c, pop) > 500000]
```

is a bit lengthy to write. Also, we need to specify the type of the argument of the parameter function which may not always be available as an explicitly defined type name. Here, it would be the type

```
tuple([name: string, pop: int])
```

A first mechanism to make this easier is that an algebra implementor can define a so-called *type operator* which, for example, computes from a given relation type the corresponding tuple type. Suppose such an operator called `TUPLE` exists (by convention, type operators are written in capitals). Then, already at the nested list level one can use this to let the system compute the argument type. Hence one could write

```
(select cities (fun (c TUPLE) ( > (attr c pop ) 500000)))
```

and this is available in text syntax as well:

```
cities select[fun(c: TUPLE) attr(c, pop) > 500000]
```

Second, in the `.spec` file entry for the `select` operator one can tell the parser to infer the function header information, that is, generate a variable name and use a type operator to get the type. For select, such an entry might be

```
operator select alias SELECT pattern _ op [ fun ] implicit parameter tuple
type TUPLE
```

This allows one to write the query above in the form

```
cities select[attr(., pop) > 500000]
```

which would be translated by the parser to

```
(select cities (fun (tuple1 TUPLE) ( > (attr tuple1 pop ) 500000)))
```

Of course, when writing the query one does not know which parameter name is generated by the parser (in particular, the parser assigns distinct numbers); hence there must be another way to refer to it, and that is the "." symbol. The same mechanism is available for operators with parameter functions taking two arguments (e.g. for implementing join operators referring to the tuple types of the first and the second argument separately); in that case the symbol ".." can be used to refer to the second argument. Instead of the symbol "." one can also write `tuple` or `group` as this makes sense for many operators; it is translated by the parser in the same way as the "." symbol.

Finally, attribute access is very frequently needed; therefore notations

```
.<attrname>
..<attrname>
```

are provided equivalent to the expressions

```
attr(., <attrname>)
attr(.., <attrname>)
```

So the use of the `attr` operator has been hard-coded into the parser. Finally we can write our query as

```
cities select[.pop > 500000]
```

3 SECONDO Commands

There is a fixed set of commands implemented at the SECONDO application programming interface. These commands can be called from a program and they are also available in the various interactive user interfaces described in Section 5. An overview is given in Table 1.

Basic Commands	**Inquiries**

`type <identifier> = <type expression>`
`delete type <identifier>`
`create <identifier> : <type expression>`
`update <identifier> := <value expression>`
`let <identifier> = <value expression>`
`derive <identifier> = <value expression>`
`delete <identifier>`
`query <value expression>`
	`list type constructors`
	`list operators`
	`list algebras`
	`list algebra <algebra-name>`
	`list databases`
	`list types`
	`list objects`

Databases	**Transactions**

`create database <identifier>`
`delete database <identifier>`
`open database <identifier>`
`close database`
	`begin transaction`
	`commit transaction`
	`abort transaction`

Import and Export	

`save database to <file>`
`restore database <identifier> from <filename>`
`save <identifier> to <filename>`
`restore <identifier> from <filename>`	

Table 1: SECONDO Commands

An identifier is defined by the regular expression `[a-z,A-Z]([a-z,A-Z]|[0-9]|_)*` with a maximal length of 48 characters, e.g. `Lineitem`, `T_LINEITEM`, `x_04` but not `_Int2` or `1000rows`.

3.1 Basic Commands

These are the fundamental commands executed by SECONDO. They provide creation and manipulation of types and objects as well as querying, within an open database.

- `type <identifier> = <type expression>`
  Defines a type name `identifier` for the type expression. Currently type identifiers can not be used within type expressions but only be as a replacement for a type expression.
- `delete type <identifier>`
  Deletes the type name `identifier`.
- `create <identifier> : <type expression>`
  Creates an object called `identifier` of the type given by `type expression`. The value is still undefined.
- `update <identifier> := <value expression>`
  Assigns the value computed by `value expression` to the object `identifier`.

- `let <identifier> = <value expression>`
  Assign the value resulting from `value expression` to a new object called `identifier`. The object must not exist yet; it is created by this command and its type is defined as the one of the value expression. The main advantage vs. using `create` and `update` is that the type is determined automatically.

- `derive <identifier> = <value expression>`
  This is a variant of the let command which can be useful to construct objects which use other objects as input and have no external list representation, e.g. indexes. Their value expressions are stored in a system table (a relation object) named `SEC_DERIVED_OBJ`. When restoring a database those objects can be automatically reconstructed.

- `delete <identifier>`
  Destroys the object `identifier`.

- `query <value expression>`
  Evaluates the `value expression` and returns the result value as a nested list.

Some example commands:

```
type myrel = rel(tuple([name: string, age: int]))
delete type myrel
create x: int
update x := 5
let place = "Hagen"
let rel_2 = [const myrel value (("peter" 27)("claus" 31))]
derive staedte_Ort = Staedte createbtree[Ort]
delete place
query (3 * 7) + 5
query Staedte feed filter[.Bev > 500000] project[SName, Bev] consume
```

3.2 Inquiries

Inquiry commands are used to inspect the actual system and database configuration.

- `list type constructors`
  Displays all names of type constructors together with their specification and an example in a formatted mode on the screen.

- `list operators`
  Nearly the same as the command above, but information about operations is presented instead.

- `list algebras`
  Returns a nested list containing all names of active algebra modules.

- `list algebra <algebra-name>`
  Displays type constructors and operators of the specified algebra.

- `list databases`
  Returns a nested list of names for all known databases.

- `list types`
  Returns a nested list of type names defined in the currently opened database.
- `list objects`
  Returns a nested list of objects present in the currently opened database.

3.3 Databases

Database commands are used to manage entire databases.

- `create database <identifier>`
  Creates a new database. No distinction between uppercase and lowercase letters is made.
- `delete database <identifier>`
  Destroys the database `identifier`.
- `open database <identifier>`
  Opens the database `identifier`.
- `close database`
  Closes the currently open database.

The state diagram in Section 3.6 shows how database commands are related to the two states OPEN and CLOSED of a database.

3.4 Transactions

Each of the basic commands of SECONDO is encapsulated into its own transaction and committed automatically. If you want to put several commands into one single transaction the following commands have to be used.

- `begin transaction`
  Starts a new transaction; all commands until the next commit command are managed as one common unit of work.
- `commit transaction`
  Commits a running transaction; all changes to the database will be effective.
- `abort transaction`
  Aborts a running transaction; all changes to the database will be revoked.

3.5 Import and Export

An entire database can be exported into an ASCII file and loaded from such a file. Similarly, a single object within a database can be saved to a file or restored from a file.

A database file is a nested list of the following structure:

```
(DATABASE <identifier>
  (DESCRIPTIVE ALGEBRA)
  (TYPES <a sequence of types>)
  (OBJECTS <a sequence of objects>)
  (EXECUTABLE ALGEBRA)
  (TYPES <a sequence of types>)
  (OBJECTS <a sequence of objects>))
```

Each of the mentioned sequences may be empty. Each type is a list of the form

```
(TYPE <identifier> <type expression>)
```

and each object is a list

```
(OBJECT <identifier> (<type identifier>) <type expression> <value expres-
sion> <model>)
```

An object does not need to have a named type. In that case the third element is an empty list.

A file for a single object contains just such a list.

- `save database to <filename>`
  Write the entire contents of the currently open database in nested list format into the file `filename`. If the file exists, it will be overwritten, otherwise it will be created. Note that filenames must not contain periods, e.g. `geo.txt` is not suitable

- `restore database <identifier> from <filename>`
  Read the contents of the file `filename` into the database `identifier`. Previous contents of the database are lost.

- `save <identifier> to <filename>`
  Writes the object list for object `identifier` to file `filename`. The content of an existing file will be deleted.

- `restore <identifier> from <filename>`
  Creates a new object with name `identifier`, possibly replacing a previous definition of `identifier`. Type and value of the object are read from file `filename`.

3.6 Database States

Figure 3 shows how commands depend on and change the state of a database. The commands referred to all have the keyword “database” in their names. All commands accessing objects only work in an open database.

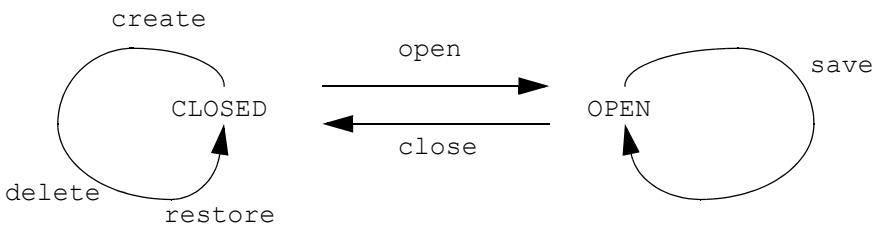

Figure 3: Database Commands and States

4 Algebra Module Configuration

As described in Section 1, a running SECONDO system consists of the kernel extended by several algebra modules. These algebra modules can arbitrarily be included or excluded when compiling and linking the system by running the `make` utility. There is also a program `TestRunner` which can be used to create comprehensive automated tests in order to verify the correctness of algebra implementations.

4.1 Activating Algebra Modules

The file `makefile.algebras` 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 += Polygon
ALGEBRAS     += PolygonAlgebra

ALGEBRA_DIRS += BTree
ALGEBRAS     += BTreeAlgebra
...
```

Just comment or uncomment some of these algebras using the # symbol at the beginning of a line. Finally recompile with the command

```
make alg=auto
```

This command automatically changes the file `Algebras/Managements/AlgebraList.i` which defines the active modules and is processed by the compiler. Their interrelationship is more precisely explained in the SECONDO Programmers Guide.

Currently, there is no mechanism which detects dependencies between algebra modules. Hence read the comments in the file. In case of trouble you have to switch on or off more algebras than the single one you wanted to change.

4.2 Invoking the TestRunner

During compilation the `TestRunner` is created in the `./bin` directory. Testfiles should be created in the `./Tests` directory. The file `./Tests/relalgtest` contains a sequence of SECONDO commands and their assumed results. When you start the `TestRunner` the standard input has to be redirected as follows:

```
TestRunner < ./Tests/relalgtest
```

Further information about creation of test files can be derived from the files `./Tests/exampletest` and `relalgtest`.

5 User Interfaces

5.1 Overview

SECONDO comes with five different user interfaces, SecondoTTYBDB, SecondoTTYCS, SecondoPL, SecondoPLCS and Javagui. The shell-based interfaces without optimizer support can be found in the bin directory. All programs related to the optimizer are in the Optimizer directory. Javagui is located in the Javagui directory.

SecondoTTYBDB is a simple single-user textual interface, implemented in C++. It is linked with the system frame. Its most interesting field of application is debugging and testing the system without relying on client-server communication. SecondoPL is the single-user version of the optimizer. SecondoTTYCS, SecondoPLCS and Javagui are multi-user client-server interfaces. They exchange messages with the system frame via TCP/IP. Provided that the database server process has been started, multiple user interface clients can access a SECONDO database concurrently.

5.2 Single-Threaded User Interfaces

5.2.1 SecondoTTYBDB

SecondoTTYBDB is a straightforward interface implementation. Both input and output are textual. Since SecondoTTYBDB materializes in the shell window from which it has been started, existence and usage of features like scrolling, cut, copy and paste etc. depend on the shell and window manager environment. A command ends with a "`;`" or an empty line. So multi-line commands are possible. SecondoTTYBDB has additional commands described in the following table:

Command	Description
`? or HELP`	Displays all user interface commands.
`@<filename>`	Starts batch processing of the specified file. The file must be in the bin directory. Each line of the input file is supposed to contain a SECONDO command. Empty lines are ignored. All lines are subsequently passed to the system, just as if they were typed in manually at the user prompt. After execution of the last command line, SecondoTTYBDB returns to interactive mode.
`D, E, H`	Set the command level (D = descriptive, E = executable, H = hybrid; currently only executable is used).
`SHOW <option>`	Shows system information. The valid option is LEVEL showing the current command level.
`DEBUG {0|1|2}`	Sets the debug mode. See online help for further information.

Table 2: Commands of SecondoTTYBDB

Command	Description
`q or quit`	Quits the session. A final `abort transaction` is executed automatically. After that, `SecondoTTYBDB` is terminated.

Table 2: Commands of SecondoTTYBDB

For the algebra modules, `SecondoTTYBDB` is extended by support functions for pretty printed output of tuples and relations. Notice that an algebra implementor is not obliged to provide these functions. As a consequence, there might be active algebra modules having types for which no pretty printing can be performed. If no functions for pretty printing of an object are defined, this object is displayed in nested list format.

If you have enabled the readline functionality (see Installation Guide), you will have some additional features. The command history is available by pressing the `cursor-up` and `cursor-down` keys, respectively. The history remains available even after termination of SECONDO. By pressing the `tab` key, the input is extended to the next matching keyword. Keywords are all words from the SECONDO commands (`list`, `database`, etc.) and some frequently used operators (`feed`, `consume`). A double `tab` prints out all possible extensions of the current word.

5.2.2 SecondoPL

`SecondoPL` is the text-based interface of the optimizer of SECONDO. To start this interface, go into the `Optimizer` directory of SECONDO and enter `SecondoPL`. Some messages are printed out. At the first run of `SecondoPL` some error messages pointing to non-existing files will occur. You can ignore them. On Linux machines you will have the advantages of the readline library if it is installed.

5.3 Multi-User Operation

5.3.1 SecondoMonitor and SecondoListener

Before the client-server user interfaces can be used, `SecondoListener`, the database server process waiting for client requests, must be started. The host name and the port address can be changed in the file `SecondoConfig.ini`. To start the `SecondoListener` type `SecondoMonitorBDB`. After some messages you should enter `startup`. `SecondoListener` is started and waiting for requests from clients. Additional commands are listed using the command `HELP`.

5.3.2 OptimizerServer

If you want to use the Optimizer within Javagui, you must also start an optimizer server. Because this server acts as a client for SECONDO, the `SecondoListener` must be running before. To start the optimizer server, go into the `Optimizer` directory of SECONDO and enter `StartOptServer [Port]`. Without any argument, the default port 1235 is taken. The available commands of the optimizer server are:

Command	Description
`help`	Prints out the available commands.
`quit`	Quits the server.
`clients`	Prints out the number of connected clients.
`trace-on`, `trace-off`	Enables (diables) tracing. If tracing is enabled, the query, the used database and the computed query plan are printed out.

Table 3: Commands of the optimizer server

5.3.3 SecondoTTYCS

`SecondoTTYCS` is a client version of the single-threaded `SecondoTTYBDB` described in Section 5.2. The main difference is that all user queries are shipped to the database server via TCP/IP, which is capable to serve multiple clients simultaneously, rather than calling frame procedures directly. For the user of a `SecondoTTYCS` client, appearance and functionality are pretty much the same as those of `SecondoTTYBDB`. All commands work in the same way as with the single-threaded user interface.

To start `SecondoTTYCS`, change to the directory `bin`, and type `SecondoTTYCS`. Remember that `SecondoListener` (Section 5.3.1) must be running.

5.3.4 SecondoPLCS

The text-based client version of the optimizer of SECONDO is `SecondoPLCS`. You can start it by entering `SecondoPLCS` in the `Optimizer` directory of SECONDO. The functionality is the same as in `SecondoPL`. Because the optimization is done within this client, the optimizer server is not required for using `SecondoPLCS`. However, the `SecondoListener` (Section 5.3.1) must be running.

5.3.5 Javagui

`Javagui` is a user-friendly, window-oriented user interface implemented in Java. Among its main features are:

- Javagui can be executed in any system in which a Java virtual machine (Ver. 1.4.2 or higher) is installed.
- Javagui has a command history for easy editing of previous commands.
- They are several viewers to display a lot of different types (e.g. spatial data types).
- There is the possibility to import files in different formats.
- Query results can be saved in a file.
- New viewers can be added.
- Javagui supports the optimizer of SECONDO.

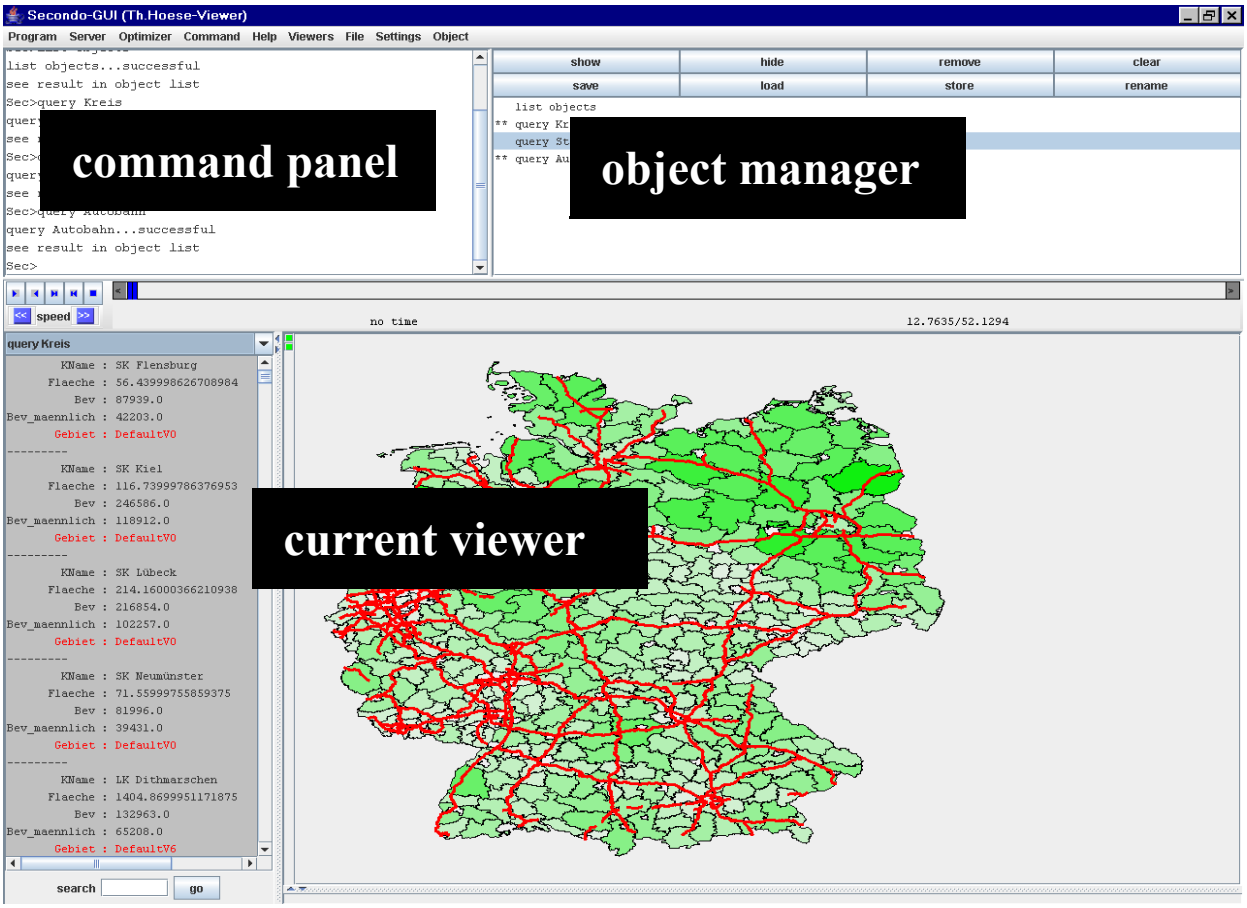

Figure 4: The three Parts of Javagui

The Configuration File

Before `Javagui` can be executed, the user has to make a few settings in the `gui.cfg` file. In this file, variables used by `Javagui` can be defined.

Configuration File Format

Because empty lines are ignored, the user can insert such lines to make the file more readable. Lines beginning with a `#` are comments. All other lines consist of a name and a value list. All entries in a line are separated by one or more spaces. A line ending with a backslash (\) together with the next line is read as single line.

Used Variables in gui.cfg

Set the host address and port number entries, `SERVERNAME` and `SERVERPORT`, to the correct values (see Section 5.3.1). The `Javagui` can be extended by new viewers. All viewers that should be automatically available at the start of a session must be listed in the configuration file as `KNOWN_VIEWERS`. Rarely used viewers can also be loaded at runtime. To use this user interface without SECONDO (e.g. to browse file contents), `START_CONNECTION` can be set to `false`. To avoid debug messages, set `DEBUG_MODE` to `false`. They are two possibilities to receive a nested list from the SecondoServer, textual and binary respectively. In most cases, the binary transfer of

nested lists is faster and more robust against keywords used by the client server protocol of SECONDO within such lists. Ensure to enable the runtime flag `Server:BinaryTransfer` in the `SecondoConfig.ini` file (Section 5.3.1) when this variable is set to `true` (disable if `false`). For using the optimizer, you can change the values of the hostname (`OPTIMIZER_HOST`) and the port (`OPTIMIZER_PORT`) of the OptimizerServer. If `ENABLE_OPTIMIZER` is set to the value `true`, `Javagui` tries to connect to an OptimizerServer at start. Furthermore, several properties can be set to private values, but this is not needed in a standard SECONDO installation.
After that, `Javagui` is ready to run.

Starting the Javagui

The easiest way to start `Javagui` is to call the `sgui` script. Remember to start the `SecondoListener` process before executing the script (see Section 5.3.1). For optimizer functionality, ensure that the OptimizerServer is also running (see Section 5.3.2). If very large objects are to be loaded into the user interface, a lot of main memory is required and the memory setting in this script should be changed.

On calling `sgui`, a window will appear on the screen. This window will have three parts: the current viewer, the object manager, and the command panel (see Figure 4).

The Menubar

The menubar of Javagui consists of a viewer independent part and a viewer depending one. The following description includes only viewer-independent parts.

Menu	Submenu / Menuitem	Description
`Program`	`New`	Clears the history and removes all objects from `Javagui`. The state of SECONDO (opened databases etc.) is not changed.
	`Fontsize`	Here you can change the fontsize of the command panel and object manager.
	`Execute File`	Opens a file input dialog to choose a file. Then the batch mode is started to process the content of the selected file. You can choose how to handle errors.
	`History`	In this menu you can load, save or clear the current history in the command panel.
	`Exit`	Closes the connection to SECONDO and quits `Javagui`.
`Server`	`connect`	Connects `Javagui` with SECONDO.
	`disconnect`	Disconnects the user inferface from SECONDO.

Table 4: Menubar of Javagui

Menu	Submenu / Menuitem	Description
	`Settings`	Shows a dialog to change the address and port used for communication with SECONDO.
`Optimizer`	`Enable`	Connects to an OptimizerServer.
	`Disable`	Closes the connection to an OptimizerServer.
	`Command`	In this menu the update functions of the optimizer for relations and indexes can be called.
	`Settings`	Opens a dialog to change the settings of hostname and portnumber of the OptimizerServer.
`Command`		This menu contains all available Secondo commands. Menu entries beginning with a ~ require additional information. If such an entry is selected, a template of the command is printed out to the command panel. Other commands are processed directly without further user inputs.
`Help`	`Show gui commands`	Opens a new window containing all gui commands (see Table 5).
	`Show secondo commands`	Shows a list of all known SECONDO commands.
`Viewer`	`<name list>`	All known viewers are listed here. By choosing a new viewer the old viewer is replaced.
	`Set priorities`	Opens a dialog to define priorities for the loaded viewers (see Figure 5).
	`Add Viewer`	Opens a file input dialog for adding a new viewer at runtime.
	`Show only viewer`	Blends out the command panel and the object manager to have more space to display objects. The menu entry is replaced by `show all`, which displays all hidden components.

Table 4: Menubar of Javagui

Setting Viewer Priorities

In the priority dialog you can choose your preferred viewer. The viewer at the top has the highest priority. To change the position of a viewer, select it and use the `up` or `down` button. Information

about the display capabilities of a specific object is given by the viewer. This feature is used when

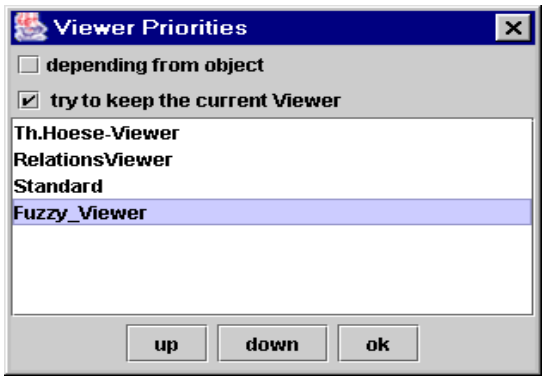

Figure 5: The Priority Dialog

`depending from object` is selected. The Viewer with the highest value is used to display an object. If you have selected the box `try to keep the current viewer` the current viewer is only replaced by another one when it cannot display the object.

The Command Panel

With the command panel the user can communicate with the `SecondoServer`. After the prompt `Sec>` you can enter a command and terminate it with `return`. The command is stored in the history. A history entry can be selected by `cursor-up` and `cursor-down` keys. All SECONDO commands are available. Additionally, some gui commands exist to control the behaviour of `Javagui`:

Command	Description
`gui exit`	Closes the connection to SECONDO and quits `Javagui`.
`gui clearAll`	Removes all objects from `Javagui` and clears the history.
`gui addViewer <viewer name>`	Adds a new viewer at runtime. If the viewer was already loaded, then the current viewer is replaced by this viewer.
`gui selectViewer <viewer name>`	Replaces the current viewer by the viewer with the given name.
`gui clearHistory`	Removes all entries from the history.
`gui loadHistory [-r]`	Shows a file input dialog and reads the history from this file. Used with the -r option this command replaces the current history with the file content. Without the -r option this command appends the file content to the current history.

Table 5: Gui Commands

Command	Description
`gui saveHistory`	Opens a file dialog to save the content of the current history.
`gui showObject <ObjectName>`	Shows an object from the object manager in a viewer. The viewer is determined by the priority settings.
`gui showAll`	Shows all objects in the object manager in the current viewer, whose type is supported by this viewer.
`gui hideObject <ObjectName>`	Removes the object with the specified name from the current viewer.
`gui hideAll`	Removes all objects from the current viewer.
`gui removeObject <ObjectName>`	Removes the object with the given name from the object manager and from all viewers.
`gui clearObjectList`	Removes all objects in the object manager.
`gui saveObject <ObjectName>`	Opens a file dialog to save the object with the given object name.
`gui loadObject`	Opens a file dialog to load an object.
`gui setObjectDirectory <directory>`	Sets the object directory. This directory is initially shown when a `load` or `save` command is executed.
`gui loadObjectFrom <Filename>`	Loads the object with the specified filename. The file must be in the `objectdirectory`.
`gui storeObject <ObjectName>`	Stores an object in the currently opened database. The object name can't contain spaces.
`gui connect`	Connects `Javagui` to SECONDO.
`gui disconnect`	Disconnects `Javagui` from SECONDO.
`gui serverSettings`	Opens the server setting dialog to change the default settings for host name and port.
`gui renameObject <old name> -> <new name>`	Renames an object.
`gui onlyViewer`	Blends out the command panel and the object manager. To show the hidden components use the `Viewers` entry in the menubar.

Table 5: Gui Commands

Command	Description
`gui executeFile [-i] <filename>`	Batch processing of the file. If -i is set then file processing is continued when an error occurs

Table 5: Gui Commands

Each non-empty query result requested in the command panel is moved to the object manager and showed in the viewer determined by priority settings. If no viewer is found, which is capable to display the requested object, a message is shown to inform the user about this. If the `StandardViewer` is loaded, this should never occur.

If the optimizer is enabled, you can also input queries in SQL syntax. All queries beginning with `select` or `sql` are shipped to the OptimizerServer to get a query plan. You can also use embedded SQL queries for a postprocessing of the result of such a query. An example is:

```
query (select sname from staedte where bev>500000) feed head[3] consume
```

Each select-clause in brackets is optimized separately. Note that the result of an optimized query is always a relation. For this reason, the `feed` operator is required in the example. In contrast to an embedded select-clause, a single select-clause adds the leading `query` automatically.

The Object Manager

This window manages all objects resulting from queries or file input operations. The manager can be controlled using a set of buttons described below:

Button	Consequence
`show`	Shows the selected object in the viewer depending on priority settings.
`hide`	Removes the selected object from the current viewer.
`remove`	Removes the selected object from all viewers and from the object manager.
`clear`	Removes all objects from all viewers and from the object manager.
`save`	Opens a file dialog to save the selected object to a file. If the selected object is a valid SECONDO object [consisting of a list (type value)] and the chosen file name ends with `obj` then the object is saved as a SECONDO object.
`load`	Opens a file dialog to load an object. Supported file formats are nested list files, shape files or dbase3 files. In the current version restrictions in shape or dbf files exists.
`store`	Stores the selected object in the currently opened database.

Button	Consequence
`rename`	Replaces the object manager with a dialog to rename the selected object.

The Viewers

The Standard Viewer

The `StandardViewer` simply shows a SECONDO object as a string representing the nested list of this object. In the text area there is only one object at the same time. To show another object in this viewer it must be selected in the combobox at the top of this viewer. You can remove the current (or all) object(s) in the extension of the menubar. Make sure to load the `StandardViewer` by default to be able to display any SECONDO object.

The Relation Viewer

This viewer displays SECONDO relations as a table. The object to be displayed can be selected in the combobox at the top of this viewer. The viewer is not suitable for displaying relations with many attributes or relations containing large objects.

The Hoese Viewer

This viewer is very powerful and can display a lot of SECONDO objects. The configuration file of this viewer `GBS.cfg` contains a few settings. It should not be needed to change this file in a standard SECONDO installation.

The viewer consists of several different parts to display textual, graphical and temporal data. If an object in the textual part is selected, then the corresponding graphical representation is also selected (if it exists) and vice versa.

The Textual Representation of an Object

Using the combobox at the top of the text panel you can choose another object to display. You can search a string in the selected object by entering the search string in the field at the bottom of the text panel and clicking on the `go` button. If the end of text is reached, the search continues at the beginning of the text.

The Graphical Representation of Objects

The graphic panel contains geometric/spatial objects. You can zoom in by drawing a rectangle with a pressed right mouse key. Stepwise zoom in (zoom out) is possible in the `Settings` menu or by pressing `Alt + (Alt -)`. To view all contained objects click on `Zoom out` in the `Settings` menu or press `Alt z`.

Each query result is displayed in a single layer. To hide/show a layer use the green/gray buttons on the left of the graphic panel. The order of the layers can be set in the layer management located

in the `Settings` menu. A selected object can be moved to another layer in the `Object` menu. Here the user also can change the display settings for a single selected object.

If data containing geographical coordinates (longitude, latitude) should be displayed, the menu `Settings->Projections` offers the possibility to enable one of a set of projections. The familiar view of such data is obtained using the Mercator projection.

Sessions

A session is a snapshot from the state of this viewer. It contains all objects including the display settings. You can save, load or start an empty session in the `File` menu.

Categories

A category contains information about how to display an object. Such information is color or texture of the interior, color and thickness of the borderline, size and form of a point. Categories can

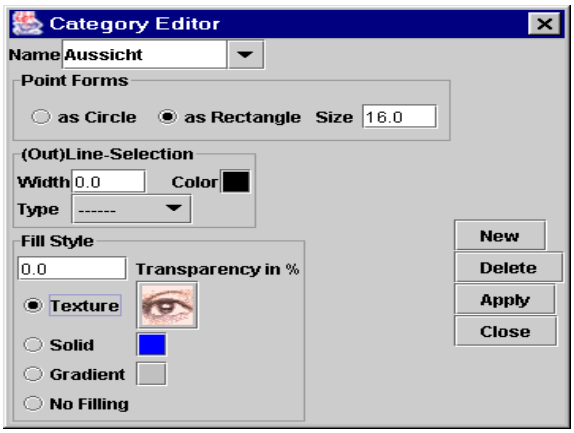

Figure 6: Category Editor

be loaded and saved in the `File` menu. To edit an exististing category you must invoke the category editor available in the `Settings` menu.

References

A reference is a link between an attribute value of a relation and a category. So you can display objects depending on a given value. You can load and save references in the `File` menu. Creating and editing references is possible in the query representation dialog. This dialog is displayed when a new object is inserted in the viewer or by selecting it in the `Settings` menu.

Query Representation

In this window the user can make settings for displaying a query result with graphical content. This can be a single graphical object or a relation with one or more graphical attributes. At the top the user can choose an existing category for all graphical objects in this query. The button `...` invokes the category editor to create or change categories. A graphical object can have a label. The label content can be entered as `Label Text`. If the object is part of a relation, the value of

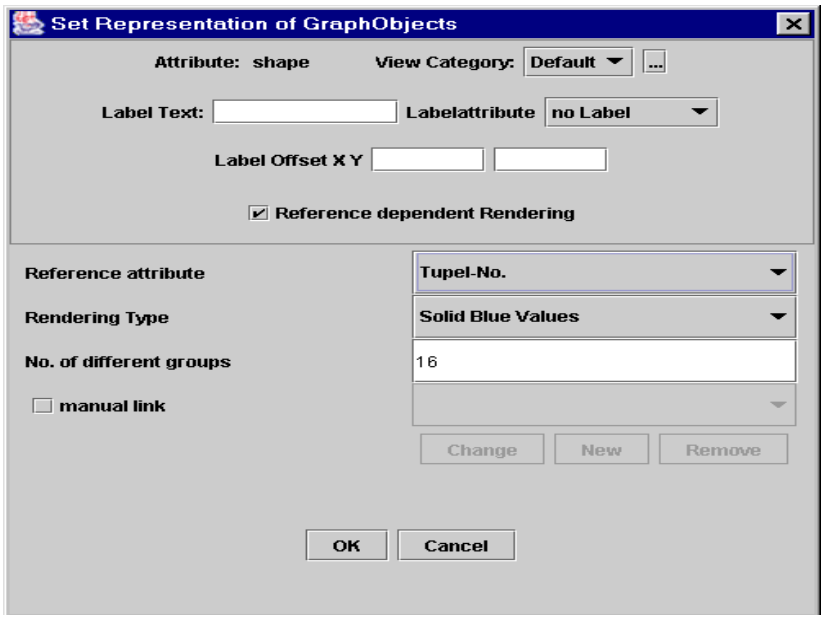

Figure 7: Query Representation

another attribute can be used as label. This feature is available in the `Labelattribute` combobox. In this case the user can also make graphical settings for objects contained in the relation. If `Single Tuple` is selected, for each single tuple in the relation an own category can be chosen. Another possibility is to choose the category dependent from an attribute in the relation. For numeric attributes categories can be computed automatically. E.g. the size of a point representing a city can be computed depending on the number of people living in this city. You can also make a manual reference between attribute values and categories.

Animating Temporal Objects

If a spatial-temporal object is loaded you can start the animation by clicking on the `play` button left of the time line. The speed can be chosen in the `Settings` menu. The speed can also be halved (doubled) by clicking on the `[<<]speed[>>]` buttons. The other buttons are `play`, `play backwards`, `go start`, `go end` and `stop`. You can also use the time scrollbar to select a desired time.

6 Algebra Modules

6.1 Overview

Included in the full SECONDO release is a large set of different algebra modules. Two of these algebras are somehow fundamental and therefore are described here in detail. They are:

- a standard algebra named Standard-C++
- a relation algebra named Relation-C++

Both algebras are located in the *Algebras*-directory of the SECONDO installation. Algebra modules can be activated and deactivated by changing the configuration of SECONDO. However the standard algebra and relation algebra are activated by default.

6.2 Standard Algebra (Standard-C++)

6.2.1 Standard Type Constructors

The standard algebra module provides four constant type constructors (thus four types) for standard data types:

- *int* (for integer values): The domain is that of the *int* type implemented by the C++ compiler, typically -2147483648 to 2147483647.
- *real* (for floating point values): The domain is that of the *float* type implemented by the C++ compiler.
- *bool*: The value is either TRUE or FALSE.
- *string*: A value consisting of a sequence of up to 48 ASCII characters.

Each standard data type contains an additional flag determining whether the respective value is *defined* or not. Thus, for instance, integer division by zero does not cause a runtime error, but the result is an undefined integer value.

6.2.2 Standard Operators

Table 6 shows the most important operators provided by the standard algebra module. Most of the operations (like +, -, *, /) are overloaded and work for both types, *int* and *real*. For more information about specific operations type `list operators` in your SECONDO interface.

In Table 6 we use the following notations. For signatures, `int || real` means that either of the types *int* or *real* can be used as an argument. In the *syntax* column, "_" denotes an argument, and "`#`" the operator; parentheses have to be put as shown.

Operator	Signature	Syntax	Semantics

`+, -, *`	`int x int -> int`
	`int x real -> real`
	`real x int -> real`
	`real x real -> real`
		`_ # _`	addition, subtraction, multiplication

`/`	`(int || real) x (int || real) -> real`	`_ # _`	division
`div, mod`	`int x int -> int`	`_ # _`	integer division and modulo operation
`<, <=, >, >=, =, #`	`(int || real) x (int || real) -> bool`	`_ # _`	comparison operators
`starts`	`string x string -> bool`	`_ # _`	TRUE if arg_1 begins with arg_2
`contains`	`string x string -> bool`	`_ # _`	TRUE if arg_1 contains arg_2
`not`	`bool -> bool`	`# ( _ )`	logical not
`and, or`	`bool x bool -> bool`	`_ # _`	logical and

Table 6: Standard Operators

6.2.3 Query Examples (Standard Algebra)

Notice that queries can only be processed after a database was opened by the user. In the query command

```
query <value expression>
```

`value expression` is a term over the active algebra(s). Some examples for queries using the standard algebra are given in Table 7. Note that no operator takes priority over another operator. Therefore a multiplication is not computed before an addition as the examples 3 and 4 show. Parentheses must be used in this case.

Text Syntax	Nested List Syntax
`query 5;`	`(query 5);`
`query 5.0 + 7;`	`(query (+ 5.0 7));`
`query 3 * 4 + 9;`	`(query (* 3 (+ 4 9)));`
`query (3 * 4) + 9;`	`(query (+ (* 3 4) 9));`
`query 6 < 8;`	`(query (< 6 8));`

`query ("Secondo" contains "cond") and TRUE;`	`(query`
	`  (and`
	`  (contains "Secondo" "cond")`
	`  TRUE));`

Table 7: Query Examples (Standard Algebra)

6.3 Relation Algebra (Relation-C++)

6.3.1 Relation Algebra Type Constructors

The relational algebra module 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

	→ DATA	*int*, *real*, *string*, *bool* (from standard algebra)
(IDENT × DATA)$^+$	→ TUPLE	*tuple*
TUPLE	→ REL	*rel*

Therefore a tuple is a list of one or more pairs (identifier, attribute type). A relation is built from such a tuple type. For instance

```
rel(tuple([name: string, pop: int]))
```

is the type of a relation containing tuples consisting of two attribute values, namely `name` of type `string` and `pop` of type `int`. A valid value of this type in nested list representation is a list containing lists of attributes of values, e.g.

```
(
  ("New York" 732200)
  ("Paris" 2175000)
  ("Hagen" 212000)
)
```

6.3.2 Relation Operators

Table 8 shows a selection of the operators provided by the relational algebra module. Some of the operators are overloaded. For more information about the operators and a full list of operators type `list operators` in the SECONDO user interface.

Here in the description of signatures, type constructors are denoted in lower case whereas words starting with a capital denote type variables. Note that type variables occurring several times in a signature must be instantiated with the same type.

Operator	Signature	Syntax	Semantics
`feed`	`rel(Tuple) -> stream(Tuple)`	`_ #`	Produces a stream of tuples from a relation.
`consume`	`stream(Tuple) -> rel(Tuple)`	`_ #`	Produces a relation from a stream of tuples.
`filter`	`stream(Tuple) x (Tuple -> bool) -> stream(Tuple)`	`_ # [ _ ]`	Lets pass those input tuples for which the parameter function evaluates to TRUE.

Table 8: Relation Operators

attr	tuple([a1:t1, ..., an:tn]) x ai -> ti	# (_ , _)	retrieves an attribute value from a tuple
project	stream(Tuple1) x attrname+ -> stream(Tuple2)	_ # [_]	relational projection operator (on streams); no duplicate removal
product	stream(Tup1) x stream(Tup2) -> stream(Tup3)	_ _ #	relational Cartesian product operator on streams
count	stream(Tuple) -> int rel(Tuple) -> int	_ #	Counts the number of tuples in a stream or a relation.
extract	stream(tuple([a1:t1, ..., an:tn])) x ai -> ti	_ # [_]	Returns the value of a specified attribute of the first tuple in the input stream.
extend	stream(tuple([a1:t1, ..., an:tn]) x [(b1 x (Tuple -> u1)) ... (bj x (Tuple -> uj))] -> stream(tuple([a1:t1, ..., an:tn, b1:u1, ..., bj:uj]))	_ # [_]	Extends each input tuple by new attributes. The second argument is a list of pairs; each pair consists of a name for a new attribute and an expression to compute the value of that attribute. Result tuples contain the original attributes and the new attributes. See the example in Table 9.
loopjoin	stream(Tuple1) x (Tuple1 -> stream(Tuple2))) -> stream(Tuple3)	_ # [_]	Join operator performing a nested loop join. Each tuple of the outer stream is passed as an argument to the second argument function which computes an inner stream of tuples. The operator returns the concatenation of each tuple of the outer stream with each tuple produced in the inner stream.
mergejoin	stream(Tuple1) x stream(Tuple2) x attr1 x attr2 -> stream(Tuple3)	_ _ # [_ , _]	Join operator performing merge join on two streams w.r.t. attr1 of the first and attr2 of the second stream. Each argument stream must be ordered (ascending) by the respective attribute.
concat	stream(Tuple) x stream(Tuple) -> stream(Tuple)	_ _ #	Concatenates two streams. Can be used to implement relational union (without duplicate removal).
mergesec	stream(Tuple) x stream(Tuple) -> stream(Tuple)	_ _ #	Intersection. Both streams must be ordered (lexicographically by all attributes, achievd by applying a sort operator before).
mergediff	stream(Tuple) x stream(Tuple) -> stream(Tuple)	_ _ #	Difference on two ordered streams.

Table 8: Relation Operators

groupby	stream(Tuple) x [a1 ... ai] x [(b1 x (Tuple -> u1)) ... (bj x (Tuple -> uj))] -> stream(tuple([a1:t1, ..., ai:ti, b1:u1, ..., bj:uj]))	_ # [_ ; _]	Groups a stream by the attributes given in the second argument. The third argument is a list of pairs; each pair consists of a name for a new attribute and an expression to compute the value of that attribute. Result tuples contain the grouping attributes and the new attributes. See the example in Table 9.
sortby	stream(Tuple) x (attr_i x dir_i)+ -> stream(Tuple)	_ # [_]	Operator for sorting a stream lexicographically by one or more attributes. For each attribute, the "direction" of sorting can be specified as either ascending (asc) or descending (desc).
sort	stream(Tuple) -> stream(Tuple)	_ #	Sorts the input tuple stream lexicographically by all attributes.
rdup	stream(Tuple) -> stream(Tuple)	_ #	Removes duplicates from a totally ordered input stream.
min, max, sum	stream(Tuple) x intattrname -> int stream(Tup) x realattrname -> real	_ # [_]	Returns the minimum, maximum, or sum value of the specified column. The second argument must be the name of an integer or real attribute.
avg	stream(Tuple) x numattrname -> real	_ # [_]	Returns the average value of the specified column. The second argument must be the name of an integer or real attribute.
rename	stream(Tuple1) x id -> stream(Tuple2)	_ { _ } (special syntax, operator name not needed)	Changes only the type, not the value of a stream by appending the characters supplied in arg_2 to each attribute name. The first character of arg_2 must be a letter. Used to avoid name conflicts, e.g. in joins.

Table 8: Relation Operators

6.3.3 Query Examples (Relation Algebra)

To make it easier to understand how to use the operations of the relational algebra a small database is available for this purpose. It is called testqueries and can be created and loaded as follows:

```
create database testqueries;
restore database testqueries from testqueries;
```

Now the database is loaded. It provides several (simple) relations:

```
tenTest:        rel(tuple([no: int]))
```

```
twentyTest:     rel(tuple([no: int]))
EmployeeTest:   rel(tuple([EName: string, EmpNr: int, DeptNr: int]))
DeptTest:       rel(tuple([Leader: string, DeptNr: int]))
StaedteTest:    rel(tuple([SName: string, Bev: int,
                  PLZ: int, Vorwahl: string, Kennzeichen: string]))
```

A first query example (in text syntax) is the following:

```
query tenTest feed filter [.no > 5] consume count;
```

`query` is the keyword to start a query, but 'does' nothing. Next, `tenTest` selects the relation used in the query. `feed` and `consume` are operators to produce a stream of tuples from a relation and produce a relation from a stream of tuples resp. The `filter` operation has a function `[.no > 5]` as input and removes all tuples from the stream which evaluate to `FALSE`. `count` finally counts the number of tuples in the resulting relation.

As we can see, the query in text syntax can be read from the beginning to the end: First, the relation is selected and a tuple stream is generated. For every single tuple the `filter` operation is evaluated. After that, the remaining tuples are collected and a new relation is formed. Finally, the tuples of the resulting relation are counted. For this query the proper result is 5.

As a second example we take

```
query tenTest feed twentyTest feed {A} product project[no_A] sort
  rdup count;
```

Now we use two different relations (`tenTest` and `twentyTest`). Since the attribute names of both relations are the same (they both have the attribute `no`) we have to use the `rename` operation, which renames the attribute in the second relation by appending the attribute's name with the passed character string. After that the `product` of both relations is computed. An attribute name (the new name `no_A`) is passed to the `project` operator; after the projection to this attribute the tuples of the stream are sorted and duplicates are removed. Finally the number of tuples is counted. The proper result is 20.

In contrast to the first example where the number of tuples of a relation was counted, we counted the number of tuples in the stream here. Both options are available in SECONDO.

Some more example queries are given in Table 9.

Text Syntax	Nested List Syntax

query StaedteTest feed
  avg[Bev];
	(query
	  (avg
	  (feed StaedteTest)
	  Bev));

query tenTest feed
  extend[mod2: .no mod 2]
  head[3]
  consume;
	(query
	  (consume
	  (head
	  (extend
	  (feed tenTest)
	  ((mod2
	  (fun (tuple1 TUPLE)
	  (mod
	  (attr tuple1 no)
	  2)))))
	  3)));

query StaedteTest feed
  extract[Bev];
	(query
	  (extract
	  (feed StaedteTest)
	  Bev));

query EmployeeTest feed
  sortby[DeptNr asc]
  groupby[DeptNr;
  anz: group feed count]
  consume;
	(query
	  (consume
	  (groupby
	  (sortby
	  (feed EmployeeTest)
	  ((DeptNr asc)))
	  (DeptNr)
	  ((anz
	  (fun (group1 GROUP)
	  (count
	  (feed group1))))))));

query StaedteTest feed
  max[SName];
	(query
	  (max
	  (feed StaedteTest)
	  SName));

Table 9: Query Examples (Relation Algebra)

7 Functions and Function Objects

A fundamental facility in SECONDO is the possibility to treat functions as values. We have already seen anonymous parameter functions to operators such as `select` or `filter`. The type of a function

```
fun (<arg_1>: <type_1>, ..., <arg_n>: <type_n>) <expr>
```

is

```
map(<type1>, ..., <type_n>, <resulttype>)
```

where `<resulttype>` is the type of `<expr>`. For example, the type of the function

```
fun (n: int) n + 1
```

is

```
map(int, int)
```

It is possible to create named SECONDO objects whose values are functions; technically the type constructor `map` is provided by the `FunctionAlgebra`. Hence we can say:

```
create double: map(int,int)
update double := fun (n: int) n + n
```

It is easier to create a function object through the `let` command:

```
let prod = fun (n: int, m: int) m * n
```

We can ask for the value of a function object:

```
query prod
```

As a result, the type and the value of the function are displayed in nested list syntax:

```
Function type:
(map int int int)
Function value:
(fun
    (n int)
    (m int)
    (* m n))
```

Function objects can be applied to arguments in the usual syntax and mixed with other operations:

```
query prod(5, double(7 * 6)) + 50
```

It is also possible to apply anonymous functions to arguments (this works only in nested list syntax):

```
(<anonymous function> <arg1> ... <arg_n>)
```

For example, we can write

```
(query ((fun (n int) (+ n 1)) 70))
```

and get as a result 71. It is allowed to define function objects with zero arguments; this can be used to define views. For example given a relation `Staedte` similar to `StaedteTest` from Section 6.3 we can define a view to get cities with more than 500000 inhabitants:

```
let Grossstaedte = fun () Staedte feed filter[.Bev > 500000] consume
```

Then the command

```
query Grossstaedte
```

yields the definition of the function:

```
Function type:
(map
    (rel
        (tuple
            (
                (SName string)
                (Bev int)
                (PLZ int)
                (Vorwahl string)
                (Kennzeichen string)))))
Function value:
(fun
    (consume
        (filter
            (feed Staedte)
            (fun
                (tuple1 TUPLE)
                (>
                    (attr tuple1 Bev)
                    500000)))))
```

Observe that the `map` constructor has only one argument, the result type. The function is applied (evaluated) by writing

```
query Grossstaedte()
```

Such a view can be used in further processing, for example:

```
query Grossstaedte() feed project[SName] consume
```

and it can even be used in further views:

```
let Grosse = fun () Grossstaedte() feed project[SName] consume
```

As a final example, let us define a function that for a given string argument returns the number of cities starting with that string:

```
let Staedte_with = fun (s: string) Staedte feed filter[.SName starts s]
count
```

Then

```
query Staedte_with("B")
```

returns on our little example relation the value 7.

8 The Optimizer

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

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

8.1 Preparations

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

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

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

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

Furthermore, for each of these relations called `X` there is another one with the same schema called `X-sample`. Finally, there are the two indexes `plz_Ort` and `plz_PLZ` which index on the `plz` relation the attributes `Ort` and `PLZ`, respectively. All relations are small except for `plz` which is a bit larger, having 41267 tuples.

8.2 Using SECONDO in a PROLOG environment

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

```
SecondoPL
```

After some messages, there appears a PROLOG prompt:

```
1 ?-
```

1. There is also a further relation SEC_DERIVED_OBJ used internally to restore indexes and other derived objects.

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

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

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

So at the command line, one can type:

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

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

```
Res = []
```

This is the empty list that SECONDO returns on executing successfully such a command, converted to a PROLOG list. As usual with a PROLOG interpreter we can type `<return>` to see more solutions, to which the interpreter responds

```
Yes
2 ?-
```

Let us try another command:

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

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

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

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

In the sequel, we omit the PROLOG prompt in the examples.

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

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

Hence we can say:

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

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

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

```
open
create
update
```

```
let
delete
query
```

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

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

8.3 An SQL-like Query Language

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

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

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

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

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

The best plan is:

Staedte feed filter[.Bev > 500000] consume

Estimated Cost: 120.64
```

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

```
make
pdview optimizer
pdprint optimizer
```

In the following, we describe the currently implemented query language in detail. Whereas the syntax resembles SQL, no attempt is made to be consistent with any particular SQL standard.

Basic Queries

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

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

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

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

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

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

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

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

sel-clause     ->  * | result-list

result         ->  attr | attr-expr as newname

attr           ->  attrname | var:attrname

rel            ->  relname | relname as var

where-clause   ->  where pred-list
                 | empty

pred           ->  attr-boolexpr
```

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

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

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

Finally, *empty* denotes the empty alternative. Hence the where-clause is optional.

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

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

Order

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

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

orderby-clause  ->  orderby orderattr-list
                  | empty

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

For example, we can say:

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

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

Taking Only the First *n* Elements

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

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

For example:

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

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

Grouping and Aggregation

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

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

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

groupattr       ->  attr

aggrop          ->  min | max | sum | avg
```

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

For example, one can say:

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

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

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

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

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

Union and Intersection

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

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

For example:

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

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

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

```
sql mquery
```

The complete grammar can be found in Appendix B.

8.4 Further Ways of Querying

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

```
sql Term
```

Hybrid Queries

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

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

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

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

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

Creating Objects

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

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

For example:

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

This query creates a relation `orte2` with the names of places (“Orte”) that also occur in the postal code relation `plz`. Since duplicate removal is currently not implemented in the SQL language, it is added here using a hybrid query.

Just Optimizing

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

```
optimize(Term)
```

This returns the query plan and the expected cost.

8.5 The Optimizer's Knowledge of Databases

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

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

- *storedRels* - relations and their attributes
- *storedSpells* - spellings of relation and attribute names
- *storedIndexes* - for which attributes do and do not exist indexes
- *storedCards* - cardinalities of relations
- *storedTupleSizes* - average tuple sizes (in bytes) of relations
- *storedSels* - selectivities of selection and join predicates

The optimizer currently does not distinguish between different databases. Hence, if there are different databases having relations with the same name but different properties, the optimizer may get confused. This should be avoided at the current stage of implementation.[2]

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

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

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

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

Reinitializing

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

2. Of course, this is a problem that should be addressed in the optimizer implementation.

Creating and Deleting Relations

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

Creating and Deleting Indexes

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

Informing the Optimizer

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

```
updateRel(Rel)
```

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

```
updateRel(plz).
```

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

```
updateIndex(Rel, Attr)
```

This predicate lets the optimizer only check whether an index exists on `Rel` for `Attr`. Hence this can be used after creating or destroying an index, without losing all the other information collected for relation `Rel`. For example, after deleting the index `plz_Ort` one should inform the optimizer by saying:

```
updateIndex(plz, ort).
```

8.6 Operator Syntax

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

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

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

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

PROLOG Syntax

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

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

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

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

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

SECONDO Syntax

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

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

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

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

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

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

Reload the file after modifying it:

```
[opsyntax].
```

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

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

A Operator Syntax

```
/*
1 Operator Syntax

[File ~opsyntax.pl~]

*/

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

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

/*

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

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

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

  * postfixbrackets, 2 or 3 arguments, of which the last one is put into
the brackets: _ # [ _ ] or _ _ # [ _ ]

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

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

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

For all other forms, a plan_to_atom rule has to be programmed explicitly.

*/

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).
```

```
secondoOp(leftrange, postfixbrackets, 3).
secondoOp(rightrange, postfixbrackets, 3).
secondoOp(remove, postfixbrackets, 2).
secondoOp(project, postfixbrackets, 2).
secondoOp(sortby, postfixbrackets, 2).
secondoOp(loopsel, postfixbrackets, 2).
secondoOp(sum, postfixbrackets, 2).
secondoOp(min, postfixbrackets, 2).
secondoOp(max, postfixbrackets, 2).
secondoOp(avg, postfixbrackets, 2).
secondoOp(tuplesize, postfix, 1).
secondoOp(head, postfixbrackets, 2).
```

B Grammar of the Query Language

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

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

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

sel-clause      ->  * | result-list

result          ->  attr | attr-expr as newname

attr            ->  attrname | var:attrname

rel             ->  relname | relname as var

where-clause    ->  where pred-list
                  | empty

pred            ->  attr-boolexpr

orderby-clause  ->  orderby orderattr-list
                  | empty

orderattr       ->  attrname | attrname asc | attrname desc

first-clause    ->  first int-constant
                  | empty

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

```
groupattr        ->  attr

aggrop           ->  min | max | sum | avg

groupby-clause   ->  groupby attr-list

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

C References

[DG00] Dieker, S., and R.H. Güting, Plug and Play with Query Algebras: SECONDO. A Generic DBMS Development Environment. Proc. Int. Database Engineering and Applications Symposium (IDEAS, Yokohama, Japan), 2000, 380-392.

[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.

[Gü93] Güting, R.H., Second-Order Signature: A Tool for Specifying Data Models, Query Processing, and Optimization. In: Proc. ACM SIGMOD Conference. Washington, USA, 1993, 277-286.