Secondo User Manual

Version 2, September 26, 2003

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

Praktische Informatik IV, Fernuniversität Hagen

D-58084 Hagen, Germany

Table of Contents

1 Overview

1.1 Motivation

The goal of SECONDO is to offer a "generic" database system frame that can be filled with implementations of a wide range of data models, including for example relational, object-oriented or graph-oriented DB models. Of course, in this frame it should also be possible to implement spatial, temporal or spatio-temporal models. In spite of this variety, SECONDO offers a well-defined system structure and even a fixed set of commands that a SECONDO system executes. The strategy to achieve this goal is the following:

1. We separate the data model independent components and mechanisms in a DBMS (the *system frame*) from the data model dependent parts. Nevertheless, the two sides have to work together closely.
2. The second main idea within SECONDO is to structure the representation system into a collection of algebra modules, each providing specific data structures and operations implemented on them.

1.2 Second-Order Signature

To be able to build appropriate interfaces, we need a formalism to describe to the system frame the following three aspects:

1. a data model and a query language,
2. a collection of data structures in the system capable of representing the elements of the data model, and implemented operations capable of executing the queries (and updates), and
3. rules to express the mapping from the data model level to the representation level (optimization).

Second-order signature is such a formalism. The idea is to use two coupled signatures. The first signature has kinds as sorts, and type constructors as operators. The terms of this signature are called types. This set of types will be the type system, or equivalently, the data model, of a SECONDO DBMS. The second signature uses the types generated by the first signature as sorts, and introduces operations on these types which make up a query algebra. Hence the formalism can describe part 1 above.

The same formalism is used to describe part 2, the representation and execution level. The type system generated by the first signature in this case describes data structures such as representations of atomic data types (e.g. region), specific relation representations, or B-Trees. The algebra defined by the second signature in this case could be a query processing algebra with operations such as merge-join, exact-match search on a B-tree and computing the intersection of two region values.

Finally, a rule system for optimization can also be defined as a term rewriting system for terms of the query algebra and the representation algebra.

1.3 Architecture

Abbildung 1 shows an overview of the SECONDO extensible database management system. We discuss it level-wise from bottom to top.

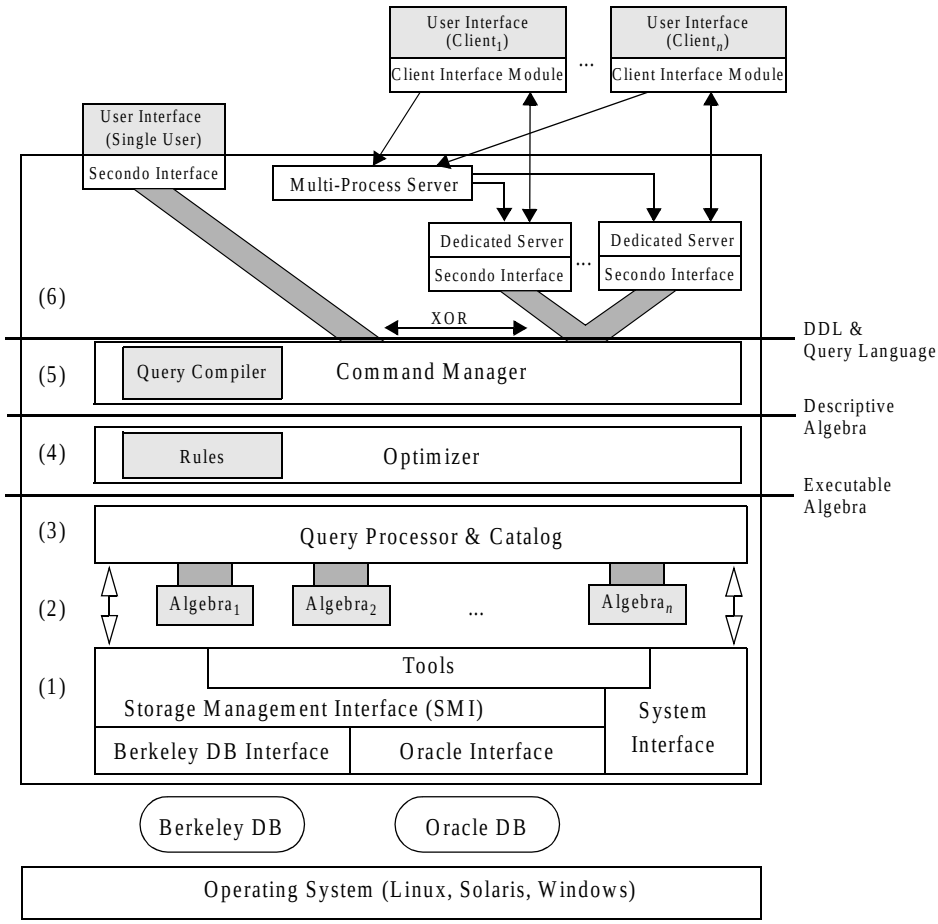

Figure 1: SECONDO architecture

The system is available on three platforms, namely Linux, Solaris and Windows. Since the goal is to offer a full-fledged database system with features like transaction management, concurrency control, and recovery, the use of a storage manager for dealing with persistent data is essential. Currently, the main system used as a storage manager is BerkeleyDB. Additionally, an Oracle DBMS can be used as a storage manager[1]. The Storage Management Interface provides a consistent interface for these two and potential further storage management systems. Additionally on level 1 there is a set of tools, which include libraries for handling nested lists, catalog management etc.

Level 2 is the algebra module level. A SECONDO algebra module defines and implements type constructors and operators using the tools of level 1. Algebra modules must be implemented in the C++ programming language. To be able to use a module's types and operators in user queries, the module must be registered with the kernel, thereby enabling modules in upper levels to call specific support functions provided by the module.

1. However, the Oracle interface has not been maintained for a while and therefore may be not fully functional in the current version.

The kernel, located in level 3, consists of the query processor, the system catalog and the module registration mechanism. During query execution the query processor controls which support functions of algebra modules are executed. Input to the query processor is the query plan, i.e. a term of the executable query algebra defined by algebra modules.

In general, user queries are not defined in terms of an executable algebra, whose operators represent specific algorithms, but rather use a more abstract and user friendly *descriptive algebra*. It is the task of the query optimizer depicted in level 4 to transform a descriptive query into an efficient evaluation plan for the query processor.

The main purpose of level 5 is to provide a procedural interface to the functionality of the lower levels. In principle, it is also possible at this level to implement a specialized query language, e.g. in the style of SQL. In such a case, one needs to implement this language by transforming to SECONDO commands at the level of descriptive algebra, writing a simple *query compiler* using tools like *Lex* and *Yacc*. In any case, the command manager takes commands at any of the available *command levels* (dedicated language, descriptive algebra, or executable algebra) and sends it to the appropriate level below for execution.

On level 6 we find the front end of a SECONDO system, providing the user interface. In general, we have two mutually exclusive alternatives. One choice is to have have a single-user interface, which is a single program linked with the frame and algebra modules. The other way is to have several different clients, which connect to the multi-process server. Then this server starts a new dedicated server with its own SECONDO interface for each client. In this case SECONDO is a multi-user capable client-server system.

This is the architecture in principle. Currently, level 4 has not been implemented in the form shown here, hence, there is no descriptive algebra available and no optimizer integrated into the architecture. However, there is a separate experimental optimizer component implemented in PROLOG on top of the SECONDO system (looking to SECONDO like an application program). It allows one to enter SECONDO commands within a PROLOG environment, and it also implements a small, but essential part of an SQL-like language in a syntax adapted to PROLOG. Query results are returned as PROLOG lists. The PROLOG optimizer is linked together with SECONDO as a single user, stand-alone program like the single user interface of level 6.

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* level (see Section 1.3) 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>(<arg1>, ..., <arg_n>)
```

This is translated into the nested list format

```
(<type constructor> <arg1> ... <arg_n>)
```

Exampe type expressions are

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

The second example uses notations for lists and pairs

```
[elem1, ..., elem_n]
x: y
```

translated by the parser into

```
(elem1 ... 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>`
`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

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.
- `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.

- `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"
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. An identifer is defined by the regular expression `{letter}({letter}|{digit}|_)*`. 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 2 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.

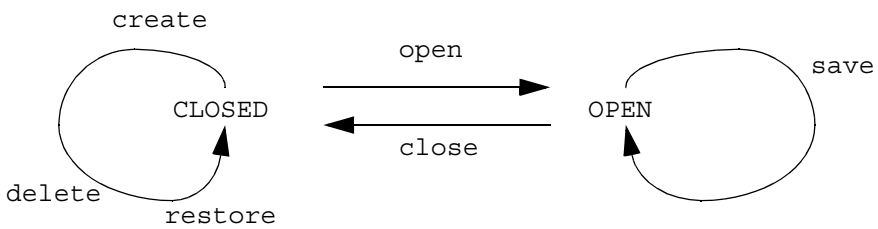

Figure 2: Database Commands and States

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 directory structure of the SECONDO source code contains a makefile with compiling instructions and code dependencies for every subdirectory where C++ code is compiled. Beginning on the top level, `make` will recursively call all other makefiles. The algebra modules are located in the subdirectory `./Algebras` and the file `./Algebras/Mangement/AlgebraList.i` contains a list of all algebras which are loaded at startup. Some example entries are presented below:

```
ALGEBRA_INCLUDE(6, PolygonAlgebra,Executable)
ALGEBRA_INCLUDE(7, DateAlgebra,Executable)
ALGEBRA_INCLUDE(8, BTreeAlgebra,Executable)
ALGEBRA_EXCLUDE(9, RangeAlgebra,Executable)
```

It is important that every algebra has it own unique id. Since this information is needed at compile time, the file has to be maintained manually. If you like to turn off an algebra either change `ALGEBRA_INCLUDE` to `ALGEBRA_EXCLUDE` or just add "`//`" in front of the line to make it a C++ comment. More detailed information is provided within the file. However, as default all algebras are compiled and linked into the SECONDO system. If you like to switch some algebras off edit the file `./makefile.algebras` and run `make alg=auto`, this will create a suitable `AlgebraList.i` file and recompile the applications.

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 three different user interfaces, `SecondoTTYBDB`, `SecondoTTYCS`, and `Javagui`. The shell-based interfaces can be found in the `bin` 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.

`SecondoTTYCS` 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 Interface: 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.
`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 for the types of 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.

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 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.

5.3.3 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 different file formats.
- Query results can be saved in a file.
- New viewers can be added.

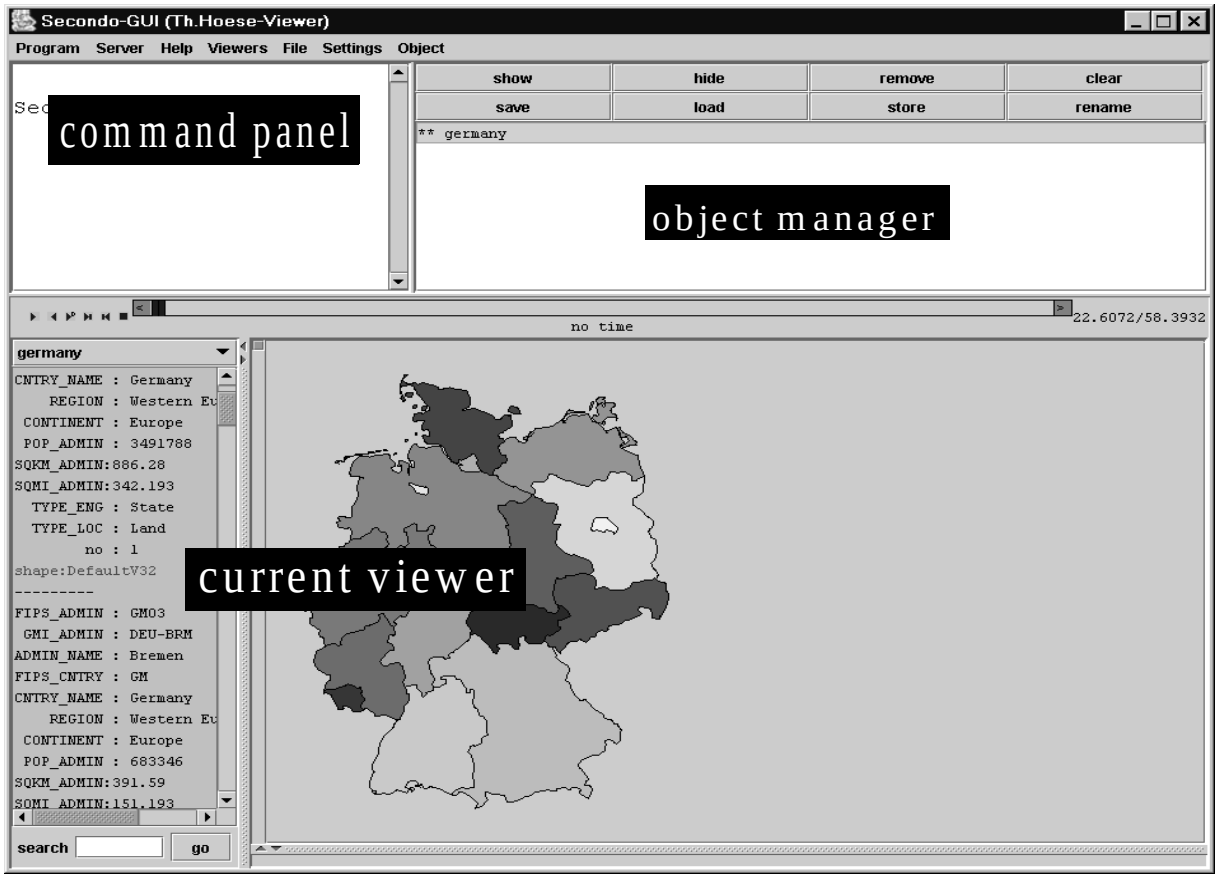

Figure 3: The three Parts of Javagui

The Configuration File

Before `Javagui` can be executed, the user has to make a few settings in the `gui.cfg` file. In a new SECONDO-System the file `gui.cfg.example` should be renamed to `gui.cfg`. 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. If a variable occurs more than one time the value lists are concatenated.

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. Furthermore several directories 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). 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 3).

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
File	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.
	Settings	Shows a dialog to change the address and port used for communication with SECONDO.
	Command	Here a few SECONDO commands can be selected.

Table 3: Menubar of Javagui

Menu	Submenu / Menuitem	Description
Help	Show gui commands	Opens a new window containing all gui commands (see Table).
	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 4).
	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 3: 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

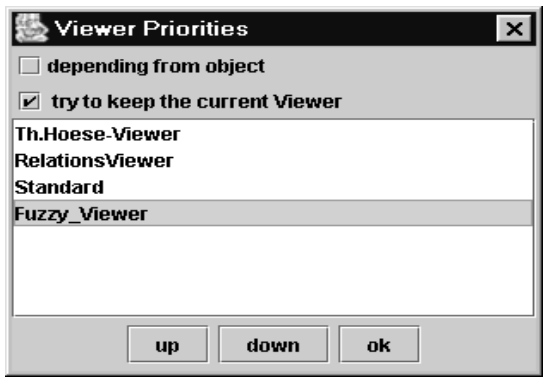

Figure 4: The Priority Dialog

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.
`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 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.

Table 4: Gui Commands

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

Table 4: 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.

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.

Button	Consequence
`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.
`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. Before the first use of this viewer rename the file `GBS.cfg.sample` to `GBS.cfg`.

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.

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

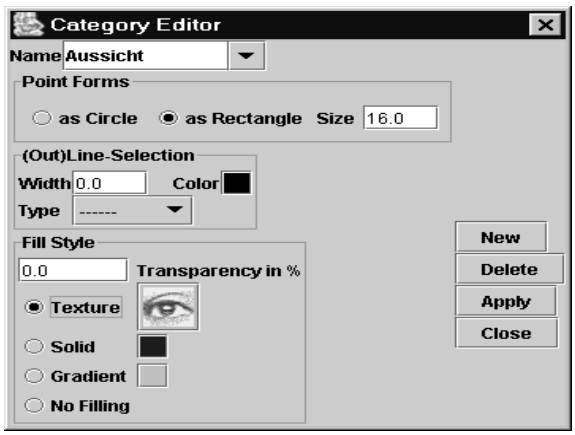

Figure 5: Category Editor

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

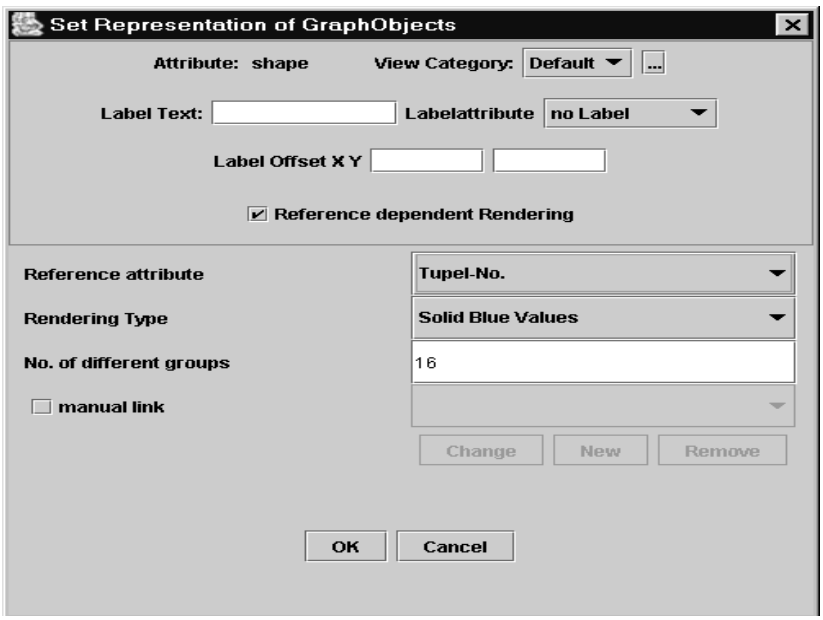

Figure 6: Query Representation

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 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 other buttons are `play backwards`, `play defined`, `go start` and `go end`. You can also use the time scrollbar to select a desired time. If `play defined` is chosen, all intervals in which no object is defined are omitted in the animation.

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 5 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 5 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 5: 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 6. Remember to switch to the executable level (E) before entering the queries. 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.

SOS 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 6: 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)
$(\text{IDENT} \times \text{DATA})^+$	→ 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 7 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 7: 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 8.
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 7: 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 8.
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 7: 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]))
```

Remember to switch to the executable level to be able to make queries.

A first query example (in SOS) 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 SOS 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 8.

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 8: 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

As mentioned in Section 1.3, SECONDO has an experimental[1] optimizer written in PROLOG that works quite well, but is not yet fully integrated with the other components of the system. This has several aspects:

- It is not yet possible to use the optimizer from either `SecondoTTYBDB`, or one of the client user interfaces (`SecondoTTYCS`, `Javagui`). Instead one has to call the `SecondoPL` executable program located in the `Optimizer` directory which is a single-user SECONDO system with a PROLOG command line interface (described below).
- The PROLOG program (the optimizer) is not automatically aware of the contents of databases, i.e., existing objects. It has to be informed through PROLOG clauses explicitly about the contents of a database one wants to work with.
- The same holds for algebras, i.e., type constructors and operators: the optimizer needs to be told about them; it does not get this information via the standard SECONDO extension mechanisms.

Nevertheless, whereas it is uncomfortable, it is not very difficult to inform the optimizer about the contents of a database. Extensions by type constructors and operators are a bit more involved and are described elsewhere. However, extensions by simple predicate operators are easy.

The version of the optimizer that comes with the standard distribution is pretty well informed about the capabilities of the implemented relational algebra and standard algebra. There is also one database, `opt`, in the standard distribution whose contents are known to the optimizer.

8.1 Preparations

To make the start easy, we work with the database `opt`. Hence, enter at any of the user interfaces the commands:

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

The second command will return some error messages:

```
*** Error in Secondo command: 24
Error in type or object definitions in file.

(60 REL
    (btree
        (tuple
            (
                (PLZ int)
                (Ort string)))
        string))=> Kind does not match type expression.

(52 6 plz_Ort)=> Wrong type expression for object.

(60 REL
    (btree
```

1. It is even more experimental than the rest of the system ...

```
        (tuple
            (
                (PLZ int)
                (Ort string)))
        int))=> Kind does not match type expression.

(52 7 plz_PLZ)=> Wrong type expression for object.

=> []
```

Don't worry, this is normal. The reason is that database `opt` contains two B-tree indexes, and we have not yet managed to organize that such an index is created automatically when restoring a database. So just enter a few more commands manually:

```
open database opt
let plz_Ort = plz createbtree[Ort]
let plz_PLZ = plz createbtree[PLZ]
```

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

```
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` that you just created 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 Calling the Optimizer

Now switch to the directory Optimizer and call the optimizer by the command:

```
SecondoPL (or SecondoPL.exe under Windows)
```

After some messages, there appears a PROLOG prompt:

```
1 ?-
```

At this point, we have a PROLOG interpreter running which understands one 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.
```

8.3 Really Calling the Optimizer

What we have done so far was to call the PROLOG interface to SECONDO. The optimizer itself, a PROLOG program, was not yet available. We now load the PROLOG code for the optimizer (from now on we omit the PROLOG prompt):

```
[auxiliary, calloptimizer].
```

This loads the clauses from the files `auxiliary.pl` and `calloptimizer.pl`. The second in turn loads a number of other files. When this is done for the first time, possibly some error messages appear; these can safely be ignored. The reason is that a file generated by the running optimizer is not yet there.

Loading the file `auxiliary` has given us a few more comfortable ways to call SECONDO. First, there is now 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.
```

Assuming the database `opt` is still open 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`, and `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'.
```

8.4 Using the Optimizer

We have already loaded the optimizer with the `calloptimizer` command (goal). The optimizer implements a small part of an SQL-like language by a predicate `sql`, to be written in prefix notation, and some other predicates (with appropriate operator definitions and priorities), e.g. `select`, `from`, `where`, that allow us to write an SQL query directly as a PROLOG term. For example, one can write (still 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 information about database contents also informs PROLOG how these names need to be spelled for SECONDO.

Some interesting messages appear that tell you something about the inner workings of the optimizer. In this case:

```
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 source code documentation, that is, say in the directory `Optimizer`:

```
make
pdview optimizer
pdprint optimizer
```

The SQL kernel implemented by the optimizer 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 "`*`" or "`count(*)`". 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].
```

It is possible to add an order-by clause

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

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].
```

All of these examples should work.

It is planned to also support a group-by clause:

```
select <aggr-list>
from <rel-list>
where <pred-list>
groupby <group-attr-list>
```

One should then be able to say:

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

However, group-by is not yet supported in the current version of the optimizer.

8.5 Describing the Contents of a Database

The optimizer knows about the contents of a database through some predicates in the file `database.pl`. The current state of this file, describing database `opt`, is shown in Appendix A. To make other relations known to the optimizer, one needs to modify this file to

1. describe the relation schema
2. explain the spelling of relation and attribute names (if non-standard)
3. write down the cardinality of the relation
4. optionally create a sample and write down its cardinality

If any indexes are present, one also needs to

5. describe the index

As an example, let us make a relation `Distrikte3` available that can be found in the subdirectory `Data/Objects/Distrikte` as a file `distrikteobjregion`. First copy the file to the Optimizer directory. Then say

```
secondo('restore Distrikte3 from distrikteobjregion').
```

We can easily determine that relation Distrikte3 has the schema:

```
Distrikte3 (Nr: int, DName: string, Flaeche: real, Bev: int,
  Bev_maennlich: int, Gebiet: region, Bev20, real)
```

and that it has 569 tuples. We now go through the steps mentioned above.

8.5.1 Relation Schema

One has to state a fact of the predicate `relation` of the form

```
relation(<relname>, <list of attribute names>).
```

All names are written in lower case. Hence we say:

```
relation(distrikte3, [nr, dname, flaeche, bev, bev_maennlich, gebiet,
bev20]).
```

8.5.2 Spelling of Relation and Attribute Names

The default assumption is that a relation name or attribute name starts with a capital letter followed by lower case symbols. If that is the case, a name must not be mentioned. Otherwise, facts of the from

```
spelling(<relname>, <spelling>).
spelling(<relname>:<attrname>, <spelling>).
```

must be stated. The first argument is given in lower case (the PROLOG notation). The second argument is given in the way this is spelled in SECONDO except that the first letter is given in lower case anyway, even if spelled in upper case in SECONDO. The reason is that if we used an upper case letter, then this would be a variable in PROLOG. About the first letter, the assumption is that it is spelled in upper case in SECONDO except if we explicitly put an `lc` operator around it. Here are some examples:

Variable	spelling in predicate	SECONDO spelling
`staedte`	`(none given, default)`	`Staedte`
`staedte:plz`	`pLZ`	`PLZ`
`plz`	`lc(plz)`	`plz`

Therefore we have to put for our example the fact:

```
spelling(distrikte3:dname, dName).
```

8.5.3 Cardinality of the Relation

This is very simple:

```
card(distrikte3, 569).
```

8.5.4 Create a Sample

The optimizer uses small subsets of existing relations, called *samples*, to determine selectivities of predicates before optimizing. It is not mandatory to create a sample; if there is no sample available when the optimizer needs it, a sample is created on the fly. However, if we provide a sample, optimization works a little bit (very little, in fact) faster.

There is an operator called `sample` in the relational algebra, which can be used to create a sample:[2]

```
let 'Distrikte3_sample = Distrikte3 sample[100, 0.01] consume'.
```

We measure the cardinality of the new relation `Distrikte3_sample`. In our case, it is 100. This is put into the file `database.pl`, too:

```
card(distrikte3_sample, 100).
```

8.5.5 Indexes

Finally, for the sake of the example, let us create an index on district names:

```
let 'distrikte3_DName = Distrikte3 createbtree[DName]'.
```

We notify the optimizer about this index by stating

```
index(distrikte3, dname, btree, distrikte3_DName).
```

That is, we state a fact of the form

```
index(<rel-name>, <attr-name>, <index-type>, <index-name>)
```

The index name is not converted in any way by the optimizer; it must start with a lower case letter. Apart from that, in principle any name could be used. As a convention, we suggest to use always the form <rel-name>_<attr-name> in SECONDO spelling, but starting with a lower case letter.

8.5.6 Summary

Altogether, we have added the following entries into the file `database.pl` to inform the optimizer about the relation `Distrikte3` and an index on it:

```
relation(distrikte3, [nr, dname, flaeche, bev, bev_maennlich, gebiet,
  bev20]).
spelling(distrikte3:dname, dName).
card(distrikte3, 569).
```

2. The precise meaning of the arguments can be looked up (list operators); without more knowledge about the working of the optimizer just use the arguments 100 and 0.01.

```
card(distrikte3_sample, 100).
index(distrikte3, dname, btree, distrikte3_DName).
```

We also created a sample by the command

```
let 'Distrikte3_sample = Distrikte3 sample[100, 0.01] consume'.
```

Creating the sample (and its `card` entry) was optional. To make these changes known to the optimizer, the file database.pl must be reloaded:

```
[database].
```

8.6 Operator Syntax

The optimizer needs to be explicitly told about the syntax of operators used in SECONDO. When we just want to use the optimizer, formulating queries in SQL, we are interested in simple operators on “atomic” data types that can be used in predicates. Examples of such operators are

```
<, >, <=, #, starts, contains, ...
```

All these operators are binary and written in infix notation. The operators mentioned are all known to the optimizer already.

If you want to use another binary infix operator, for example `touches` of the spatial algebra, it is very easy to inform the optimizer about that. Add an entry

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

at the end of the file `opsyntax.pl` whose current contents (in the standard distribution) are shown in Appendix B, and reload the file:

```
[opsyntax].
```

We can then directly write a query using such operators, for example:

```
sql select [d1:dname, d2:dname]
from [distrikte3 as d1, distrikte3 as d2]
where [d1:dname contains "Hagen", d2:gebiet touches d1:gebiet,
  d1:dname # d2:dname].
```

A Database Dependent Information

```
/*
1 Database Dependent Information

[File ~database.pl~]


1.1 Relation Schemas

*/
relation(staedte, [sname, bev, plz, vorwahl, kennzeichen]).
relation(plz, [plz, ort]).
relation(ten, [no]).
relation(thousand, [no]).
relation(orte, [kennzeichen, ort, vorwahl, bevt]).

/*
1.2 Spelling of Relation and Attribute Names

*/
spelling(staedte:plz, pLZ).
spelling(staedte:sname, sName).
spelling(plz, lc(plz)).
spelling(plz:plz, pLZ).
spelling(ten, lc(ten)).
spelling(ten:no, lc(no)).
spelling(thousand, lc(thousand)).
spelling(thousand:no, lc(no)).
spelling(orte:bevt, bevT).

/*
1.3  Cardinalities of Relations

*/
card(staedte, 58).
card(staedte_sample, 58).
card(plz, 41267).
card(plz_sample, 428).
card(ten, 10).
card(ten_sample, 10).
card(thousand, 1000).
card(thousand_sample, 89).
card(orte, 506).
card(orte_sample, 100).

/*
1.4 Indexes

*/
hasIndex(Rel, attr(_:A, _, _), IndexName) :-
  hasIndex(Rel, attr(A, _, _), IndexName).

hasIndex(rel(Rel, _, _), attr(Attr, _, _), Index) :- index(Rel, Attr, _,
Index).

index(plz, ort, btree, plz_Ort).
index(plz, plz, btree, plz_PLZ).
```

B 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).
```