SECONDO User Manual

Version 0.9, 08/03/1999

Stefan Dieker, Jose Antonio Cotelo Lema, Miguel Rodriguez Luaces

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, it should also be possible to implement within this frame 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 need to separate the data model independent components and mechanisms in a DBMS (the 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 implemented operations 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:

(a) a data model and a query language,

(b) 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),

(c) 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 so-called 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 (a) above.

The same formalism is used to describe the representation and execution level (b). 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 is 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

Figure 1 shows a coarse architecture overview of the SECONDO extensible database management system. We discuss it level-wise from bottom to top.

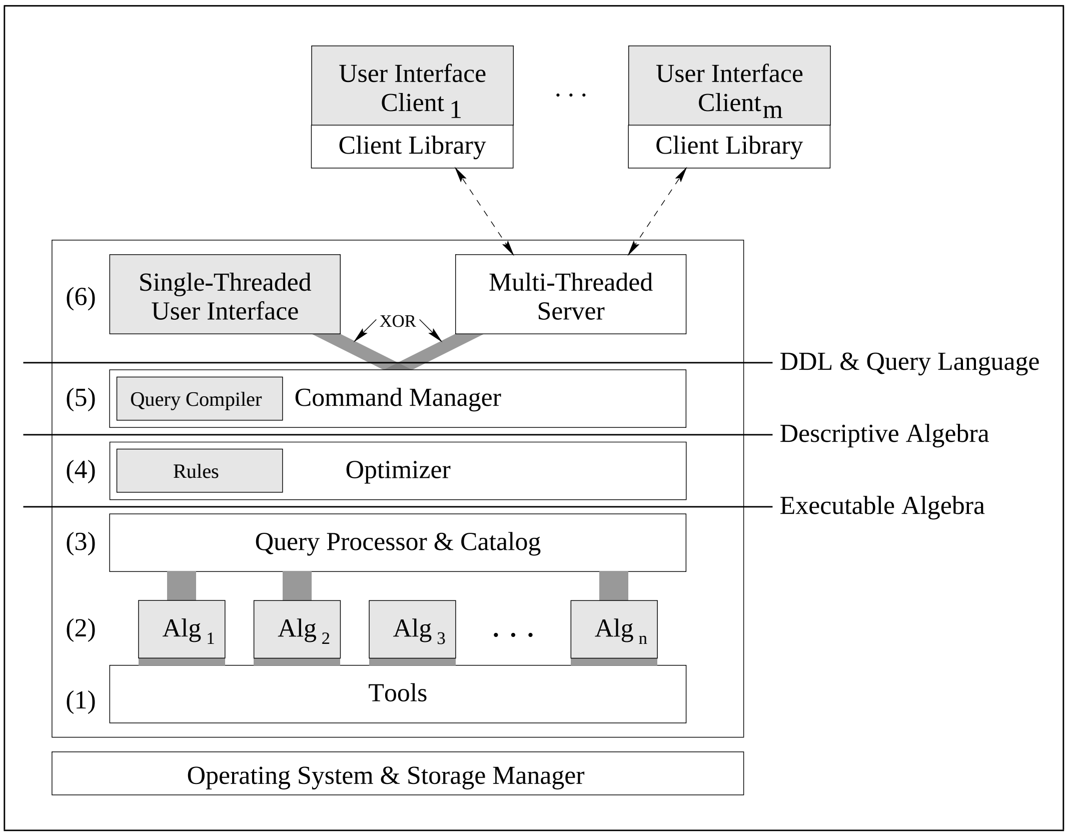


Figure 1: SECONDO architecture

The system is built on top of the Solaris operating system. Since we want to offer a full-fledged database system with features like transaction safety, concurrency control, and recovery, using a storage manager for dealing with persistent data is essential. Actually, we use the storage manager component of the SHORE system.

In level 1 of the SECONDO architecture we find a variety of tools, for instance

- a library of functions for easy handling of nested lists, the generic format to pass values as well as type descriptions.
- a simplified interface to the SHORE functions used most often.
- an efficient and easy-to-use tool for catalog management.
- an efficient implementation for handling of tuples with embedded large objects.

Level 2 is the algebra module level. To some extent, an algebra module of SECONDO is similar to ADTs of Predator or data blades of Informix Universal Server. A SECONDO algebra module defines and implements type constructors and operators using the tools of level 1. While there are some good reasons to use C++ for algebra module implementation, SECONDO allows for implementations in Modula-2 and C, too. 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. In Figure 1, modules 1, 2, and *n* are connected to the kernel, thus *active*, while module 3 is *inactive*. How to select algebra modules for activation is explained in Section 6.

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 active algebra modules are executed at which point of time. Input to the query processor essentially is an evaluation plan, i.e. a term of the executable query algebra defined by active algebra modules.

User queries usually 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. Its the task of the query optimizer depicted in level 4 to transform a descriptive query into an efficient evaluation plan for the query processor. In version 1.0, the query optimizer is not yet included.

The main purpose of level 5 is to provide a procedural interface to the functionality of the lower levels. It is the upper bound of the SECONDO *frame*. The frame is the part of SECONDO that is integrated in every SECONDO installation regardless of a specific algebra modules and user interfaces configuration. Thus, the frame consists of the components of levels 1, 3, 4, and 5.

In level 6 we find the front end of a SECONDO installation, providing the user interface. In general, we have two mutually exclusive alternatives: Either the user interface is a single program linked with the frame and active algebra modules or we do not link a user interface directly but rather make SECONDO to be a server process serving requests of an arbitrary number of client processes, which implement the user interfaces. In the first case, SECONDO is a single-user, single-process system, in the latter case SECONDO is a multi-user capable client-server system.

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 contents of the actual database.

SECONDO acccepts 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 parenthesis, 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, followed 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, provided 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`. On the other hand, SECONDO allows one to define 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 actual collection of active algebra modules. Constants, in general, are list piars of the form `(<type> <value>)`; only for standard data types (`int`, `real`, `bool`, `string`) just giving the value is sufficient. 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. 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 ) 500)))
```

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 500 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

User level syntax is provided to support the formulation of interactive user queries in a more intuitive and less error-prone way. Compared to nested list syntax, writing SECONDO commands in user level syntax essentially has three effects:

- There is no need to enclose commands by parenthesis. The string `list type constructors`, for instance, is valid input.
- Formulation of type expressions is simplified and more straightforward. In user level syntax, the definition of type `cityrel` in Section 2.2 is now given by `rel(tuple([name: string, pop: int]))`.
- SECONDO enables the algebra implementor to define syntactic properties of value expressions using the algebra's operators. Consider a system with an algebra module for standard datatypes and arithmetic operators and another module providing relational data types and operators. If the standard operators are defined to be infix operators, the relational `select` operator is a postfix operator, and the `attr` operator is a function as usual, the `select` value expression in Section 2.2 is now written as

  ```
  query cities select[fun (c: city)(attr(c,pop) > 500)]
  ```

  Exploiting all possibilites for user level syntax definition, the value expression can even be more simplified to

  ```
  cities select[.pop > 500].
  ```

In Section 3, we list all SECONDO commands in detail.

3 Commands

Table 1 contains all commands provided by a SECONDO system, grouped by command *subject* (rows) and the *situation* in which they can be used (columns). Their meaning is described in the following subsections.

	Administration	User Session
Module Configuration	• `list type constructors` • `list operators`	
Databases	• `list databases` • `create database <identifier>` • `delete database <identifier>`	• `open database <identifier>` • `close database`
Databases (File Commands)	• `restore database <identifier> from <filename>` • `extend database <identifier> from <filename>` • `extend and update database <identifier> from <filename>`	• `save database to <filename>`
Types		• `list types` • `type <identifier> = <type expression>` • `delete type <identifier>`
Objects		• `list objects` • `create <identifier> : <type expression>` • `update <identifier> := <value expression>` • `let <identifier> = <value expression>` • `delete <identifier>`
Objects (File Commands)		• `save <identifier> to <filename>` • `add <identifier> from <filename>` • `export <identifier> to <filename>` • `import <identifier> from <filename>`
Queries		• `query <value expression>`
Transactions		• `commit transaction` • `abort transaction`

Table 1: SECONDO Commands

3.1 Interactive commands

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

- `list type constructors`
  Return a nested list of type constructors (and their specifications).

- `list operators`
  Return a nested list of operators (and their specifications).
- `list databases`
  Return a list of names of existing databases.
- `list types`
  Return a nested list of type names defined in the currently open database.
- `list objects`
  Return a nested list of objects existing in the currently open database.

Database commands are used to manage entire databases.

- `create database <identifier>`
  Create a new database.
- `delete database <identifier>`
  Destroy the database `identifier`.
- `open database <identifier>`
  Open the database `identifier` and start a transaction.
- `close database`
  Close the currently open database. Before that, `abort transaction` is executed automatically.

Types and Objects The commands of this category provide creation and manipulation of user defined types and objects.

- `type <identifier> = <type expression>`
  Define a type name `identifier` for the type expression.
- `delete type <identifier>`
  Delete the type name `identifier`.
- `create <identifier> : <type expression>`
  Create an object called `identifier` of the type given by `type expression`. The value is still undefined.
- `update <identifier> := <value expression>`
  Assign the value computed by `value expression` to the object `identifier`.
- `delete <identifier>`
  Destroy the object `identifier`.
- `let <identifier> = <value expression>` (Not yet implemented)
  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 by definition the one of the value expression. This object is temporary so far; it will be automatically destroyed at the end of a session or if `abort transaction` is called.
- `persistent <identifier>` (Not yet implemented)
  Make the object `identifier`, created by the `let` command earlier, persistent, so that it will survive the end of a session. Notice that this command fails if the object is not of a persistent type, as specified by the respective algebra module implementation.

- `query <value expression>`
  Evaluate the `value expression` and return the result value as a nested list.

`Transactions`

- `commit transaction`
  Commit a running transaction; all changes to the database will be effective.
- `abort transaction`
  Abort a running transaction; all changes to the database will be revoked.

3.2 File Input and Output

In the remainder of this subsection, we list all SECONDO commands transporting database contents to files and vice versa. Two different file formats are used by these commands:

SFF, the general SECONDO File Format. An SFF file conceptually consists of a list of (type name, type definition) tuples and a list of (object name, object type, object value) tuples. SFF files can be used to save a complete database as well as just a single object to file, to restore or extend a database, or to (re-)extract specific objects from file.

Value files contain the value of at most one object in nested list format. Neither name nor type of the object are included. This format is most useful for exchanging values between SECONDO and other data processing systems.

In the following we give a short explanation of all file commands.

File Commands for Databases

- `save database to <filename>`
  Write the entire contents of the database `identifier` in nested list format to the SFF file `filename`. If the file exists, it will be overwritten, otherwise it will be created.
- `restore database <identifier> from <filename>`
  Read the contents of the SFF file `filename` into the database `identifier`. Previous contents of the database are lost. This command resets database `identifier` to the state defined by file `filename`.
- `extend database <identifier> from <filename>` (Not yet implemented)
  Read the contents of the SFF file `filename` into the database `identifier`. Previous contents of the database are *not* lost. In case of conflicting type or object names, the definitions in file `filename` are ignored.
- `extend and update database <identifier> from <filename>` (Not yet implemented)
  Works like `extend database`. In case of conflicting type or object names, however, the definitions in file `filename` overwrite those in database `<identifier>`.

File Commands for Objects

- `save <identifier> to <filename>` (Not yet implemented)
  Appends name, type, and value of object `identifier` to SFF file `filename`.
- `add <identifier> from <filename>` (Not yet implemented)
  Precondition: There is an entry for object `identifier` in SFF file `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`.
- `export <identifier> to <filename>` (Not yet implemented)
  Writes the value expression representing the value of object `<identifier>` to value file `filename`. Previous file contents get lost.
- `import <identifier> from <filename>` (Not yet implemented)
  Precondition: Object `identifier` already exists. Updates the value of `identifier` to the value expression in value file `filename`.

4 Requirements

4.1 Mandatory Requirements

SECONDO Version 1.0 is running under Solaris 2.5.1 on SUN SparcStations. Ports to other platforms with modern C++ and Modula-2 compilers should be possible, but are not supported.

The C++ sources in the distribution are compiled with gcc 2.8.1 and g++lib 2.7.2. To compile the various scanners and parsers used in SECONDO, lex and yacc must be installed in your environment, too.

The underlying Storage manager actually is the storage manager component of the SHORE system Version 1.1.1. The SHORE system is not part of the SECONDO distribution, but can be downloaded for free at `http://www.cs.wisc.edu/shore`.

4.2 Optional Requirements

To use SecondoJava, the graphical user interface implemented in Java, SUN's Java interpreter version 1.1.3 must be available. A Java installation is not mandatory to begin to use SECONDO, since there is also SecondoTTY, a simpler interface coded in C++.

Some parts of the system frame are implemented in Modula-2, compiled with the EPC Modula-2 compiler version 2.0.9, sun dialect selected. A Modula-2 compiler is not required for using the distribution or adding new algebra modules coded in C or C++, but only for modifying the system frame and activating Modula-2 algebra modules.

If you'd like to analyze the program sources: the code is written according to the requirements of the PD (program with documentation) system. Though programs and comments are readable with a simple ASCII editor, too, understanding sources formatted by the PD system is much easier. The

PD system can be downloaded at `http://www.informatik/fernuni-hagen.de/import/pi4/PDSystem/index.html`. Notice that LaTeX2_ε must also be installed on your system.

5 Installation

The g'zipped archive file `Secondo10.tar.gz` contains the complete distribution of SECONDO Version 1.0. Install it following the steps listed below:

1. Define the shell variable `SECONDOHOME`, consisting of the path to your favourite SECONDO home directory.
2. Copy `Secondo10.tar.gz` to $SECONDOHOME.
3. Change to the SECONDO home directory: Type `cd $SECONDOHOME`
4. Unpack the SECONDO distribution: `Type gunzip -c Secondo10.tar.gz | tar xvf -`
5. Edit the file `make.inc`. Change the variable `SHROOT` to the absolute path of your local SHORE root directory.
6. Create the SECONDO library `$SECONDOHOME/libsys2.a` and the programs coming with release 1.0: Type `make`
7. Change to the `UserInterfaces` directory: Type `cd UserInterfaces`
8. Create the user interfaces coming with release 1.0: Type `make`

After that, a short test session of SECONDO can be performed as follows.

1. Change to the directory `$SECONDOHOME`/UserInterfaces/SecondoTTY: Type `cd SecondoTTY`
2. Start SecondoTTY: Type `SecondoTTY`
3. In SecondoTTY, create a database: Type `create database geo`
4. Load some data into geo: Type `restore database geo from geo`
5. Commit the changes: Type `commit transaction`
6. Close the new database: Type `close database`

Now the database is ready for usual access.

7. Open the database: Type `open database geo`
8. Execute queries, for example
9. `query cities`
10. `query cities feed filter[.pop > 1000] consume`
11. To finish the session, close the database properly: Type `close database`.
12. Leave SecondoTTY: Type `q` or `quit`.

6 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[1] be included or excluded when compiling and linking the system. The way to specify which modules to include differs for Modula-2, C, and C++ algebra modules.

6.1 Selecting Modula-2 Algebra Modules

Modula-2 algebra module selection takes place in the file `$SECONDOHOME/Algebras/AlgebraManager2.mod`. To activate the module `MAlgebra`, the following steps have to performed:

1. Import the module in the beginning of `AlgebraManager2.mod`:
   `IMPORT MAlgebra;`
2. Register all support functions for `MAlgebra` by extending the body of function `LoadAlgebras`:

```
INC(NumberOfAlgebras);
MAlgebra.InitMAlgebra;
Execute[NumberOfAlgebras]        := MAlgebra.StandFunArray;
TransformType[NumberOfAlgebras]  := MAlgebra.StandTransformArray;
SelectFunction[NumberOfAlgebras] := MAlgebra.StandSelectArray;

TransformModel[NumberOfAlgebras] := MAlgebra.StandModelArray;
InModel[NumberOfAlgebras]        := Algebra.StandInModelArray;
OutModel[NumberOfAlgebras]       := MAlgebra.StandOutModelArray;
ValueToModel[NumberOfAlgebras]   := MAlgebra.StandValueToModelArray;

ExecuteCost[NumberOfAlgebras]    := MAlgebra.StandCostArray;
Constrs[NumberOfAlgebras]        := MAlgebra.StandConstrArray;
Props[NumberOfAlgebras]          := MAlgebra.StandPropsArray;
Ops[NumberOfAlgebras]            := MAlgebra.StandOpsArray;
Specs[NumberOfAlgebras]          := MAlgebra.StandSpecArray;
ConstrNumber[NumberOfAlgebras]   := MAlgebra.ConstrNumber;
OperatorNumber[NumberOfAlgebras] := MAlgebra.OpNumber;
CreateObj[NumberOfAlgebras]      := MAlgebra.StandCreateArray;
DeleteObj[NumberOfAlgebras]      := MAlgebra.StandDeleteArray;
InObj[NumberOfAlgebras]          := MAlgebra.StandInArray;
OutObj[NumberOfAlgebras]         := MAlgebra.StandOutArray;
TypeCheck[NumberOfAlgebras]      := MAlgebra.StandTypeArray;
```

3. Enable unloading the algebra module when a SECONDO session is ended by extending the body of function `UnloadAlgebras`:

```
MAlgebra.UnloadMAlgebra;
DEC(NumberOfAlgebras);
```

4. Add the object file compiled from the algebra module source code to the list of files defining the `ALGEBRAMODULES` variable in `$SECONDOHOME/make.inc`.

1. Of course, module configuration must be *consistent*: Modules which rely on other ones may not be activated without those base modules.

6.2 Selecting C Algebra Modules

Very similar to Modula-2 algebra module selection (see Section 6.1), C algebra module selection takes place in the file `$SECONDOHOME/Algebras/AlgebraManagerC.c`. To activate the module `CAlgebra`, the following steps have to performed:

1. Include the module header file in the beginning of `AlgebraManagerC.c`:
   `#include "CAlgebra.h";`
   a. Register all support functions for `CAlgebra` by extending the body of function `LoadAlgebrasC()`:

```
AlgebraNumber++;
InitCAlgebra();
Execute[AlgebraNumber]        = CAlgebraFunArray;
TransformType[AlgebraNumber]  = CAlgebraTransformArray;
SelectFunction[AlgebraNumber]= CAlgebraSelectArray;
TransformModel[AlgebraNumber]= CAlgebraModelArray;
ExecuteCost[AlgebraNumber]    = CAlgebraCostArray;
Constrs[AlgebraNumber]        = CAlgebraConstrArray;
Props[AlgebraNumber]          = CAlgebraPropsArray;
Ops[AlgebraNumber]            = CAlgebraOpsArray;
Specs[AlgebraNumber]          = CAlgebraSpecArray;
ConstrNumber[AlgebraNumber]   = CAlgebraConstrNumber;
OperatorNumber[AlgebraNumber]= CAlgebraOpNumber;
CreateObj[AlgebraNumber]      = CAlgebraCreateArray;
DeleteObj[AlgebraNumber]      = CAlgebraDeleteArray;
InObj[AlgebraNumber]          = CAlgebraInArray;
OutObj[AlgebraNumber]         = CAlgebraOutArray;
TypeCheck[AlgebraNumber]      = CAlgebraTypeArray;
```

2. Enable unloading the algebra module when a SECONDO session is ended by extending the body of function `UnloadAlgebras`C():

```
UnloadCAlgebra;
DEC(NumberOfAlgebras);
```

3. Add the object file compiled from the algebra module source code to the list of files defining the `ALGEBRAMODULES` variable in `$SECONDOHOME/make.inc`.

6.3 Selecting C++ Algebra Modules

To integrate a C++ algebra module into a SECONDO executable, just add the object file compiled from the algebra module source code to the list of files defining the `ALGEBRAMODULES` variable in `$SECONDOHOME/make.inc`. The rest of the job - registering module support functions with the kernel - is completely done by the C++ run time system.

7 User Interfaces

7.1 Overview

SECONDO Version 1.0 comes with three different user interfaces, SecondoTTY, SecondoTTYCS, and SecondoJava, as presented in Table 2. They can be found in the respective subdirectory of the

		Appearance	
		Shell-based	Window-based
Concurrency	Single-Threaded	**SecondoTTY**	
	Client-Server	**SecondoTTYCS**	**SecondoJava**

Table 2: SECONDO User Interfaces

`$SECONDOHOME/UserInterfaces` directory. SecondoTTY 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 SecondoJava are multi-user client-server interfaces. They exchange messages with the system frame via TCP/IP. Provided that DBServer, the database server process, has been started, multiple user interface clients can access a SECONDO database concurrently.

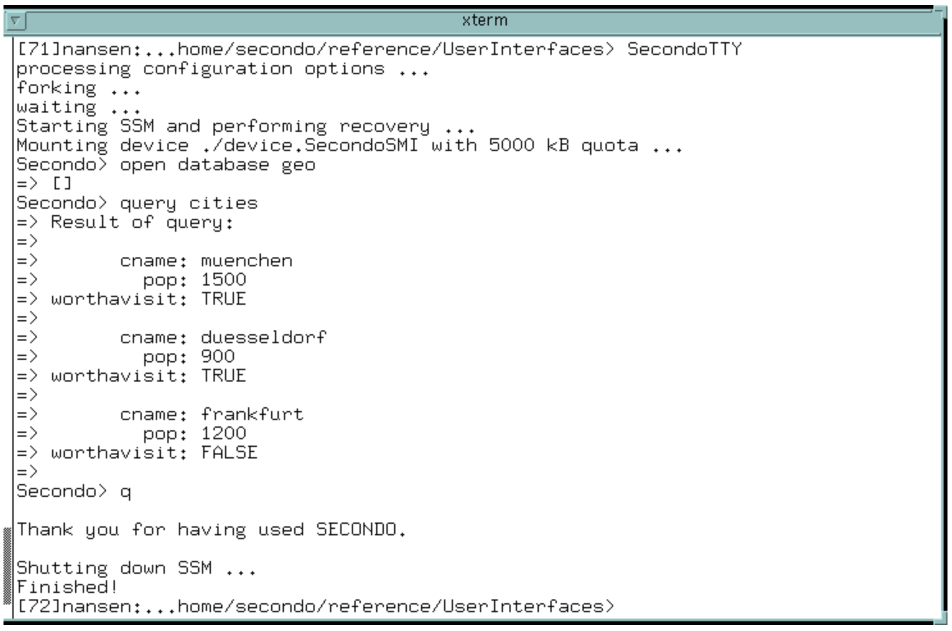
7.2 Single-Threaded User Interface: SecondoTTY

SecondoTTY is a straightforward interface implementation. Both input and output are textual. Since SecondoTTY 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.

Figure 2 shows a screen dump of a short SECONDO session using SecondoTTY:

1. SecondoTTY is started by calling the program `SecondoTTY`.
2. After some messages reporting the state of SHORE, the user prompt `Secondo>` occurs.
3. A database is opened. The answer of the system is preceded by the system prompt `=>`. Here the answer of the system is the nil symbol `[]`, indicating successful execution of a command which does not produce specific result data.
4. The relation `cities` is queried. The contents of `cities` are printed.
5. The user quits the session by typing `q`.
6. Finally, proper termination of SHORE is reported and the shell is ready again.

At the user prompt, all SECONDO commands, as listed in Section 3, can be used. In addition to that, two interface-specific commands can be called:



```
[71]nansen:...home/secondo/reference/UserInterfaces> SecondoTTY
processing configuration options ...
forking ...
waiting ...
Starting SSM and performing recovery ...
Mounting device ./device.SecondoSMI with 5000 kB quota ...
Secondo> open database geo
=> []
Secondo> query cities
=> Result of query:
=>
=>       cname: muenchen
=>         pop: 1500
=> worthavisit: TRUE
=>
=>       cname: duesseldorf
=>         pop: 900
=> worthavisit: TRUE
=>
=>       cname: frankfurt
=>         pop: 1200
=> worthavisit: FALSE
=>
Secondo> q

Thank you for having used SECONDO.

Shutting down SSM ...
Finished!
[72]nansen:...home/secondo/reference/UserInterfaces>
```

Figure 2: SecondoTTY Sample Session

use file or `uf` makes the system prompt for an input file name and switches the interface to batch mode. 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, SecondoTTY returns to interactive mode.

quit or `q` quits the session. A final `abort transaction` is executed automatically. After that, SHORE and SecondoTTY are terminated.

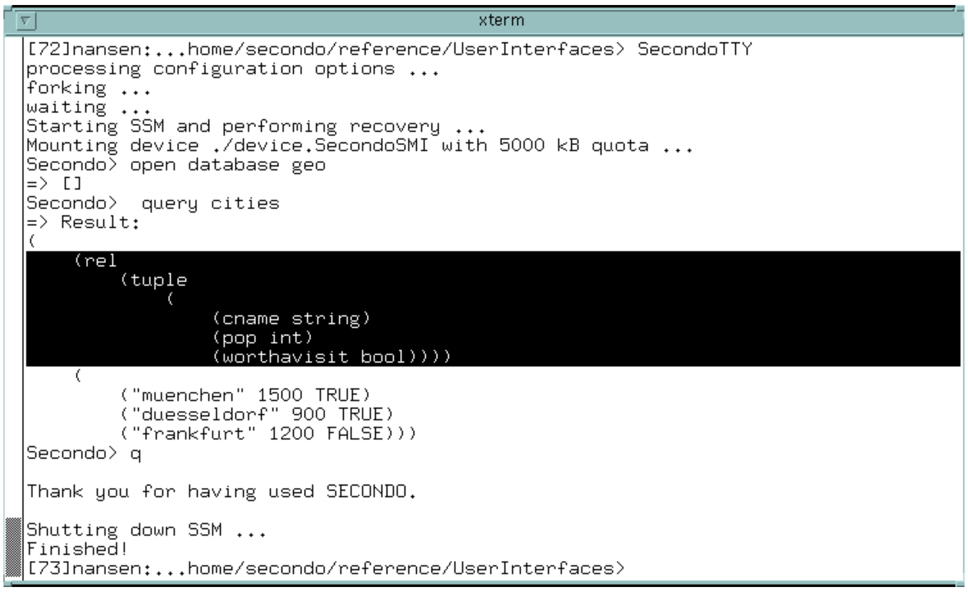
For the algebra modules delivered with release 1.0, SecondoTTY is extended by support functions for pretty printed output of tuples and relations. Notice that an algebra implementor is not obliged to provide these functions. As a consequence, there might be active algebra modules for the types of which no pretty printing can be performed. Figure 3 shows the same SECONDO session that is depicted in Figure 2 with simulated absence of support functions for pretty printing. Now the result is a tuple *(type expression, value)* in nested list format. The region presented in reverse mode is the *type expression* part of the result.

7.3 Multi-User Operation

7.3.1 DBServer

Before the client-server user interfaces can be used, DBServer, the database server process waiting for client requests, must be installed and started. To install DBServer, follow the steps listed below.

1. Determine a free port number (> 1024) where DBServer should listen for incoming requests. A port number can be checked to be unused by typing `netstat -an | grep <port number>`. If no result line is returned the port is free to be used for DBServer.



```
[72]nansen:...home/secondo/reference/UserInterfaces> SecondoTTY
processing configuration options ...
forking ...
waiting ...
Starting SSM and performing recovery ...
Mounting device ./device.SecondoSMI with 5000 kB quota ...
Secondo> open database geo
=> []
Secondo>  query cities
=> Result:
(
    (rel
        (tuple
            (
                (cname string)
                (pop int)
                (worthavisit bool))))
    (
        ("muenchen" 1500 TRUE)
        ("duesseldorf" 900 TRUE)
        ("frankfurt" 1200 FALSE)))
Secondo> q

Thank you for having used SECONDO.

Shutting down SSM ...
Finished!
[73]nansen:...home/secondo/reference/UserInterfaces>
```

Figure 3: SecondoTTY Session without Pretty Printing

2. Edit the file `$SECONDOHOME/ClientServer/dbserver`. Replace the entry `nansen.fernuni-hagen.de` by the full name of your host going to run DBServer.
3. Ask your system administrator to add an entry for a SECONDO service in the /etc/services file of the DBServer host. Such an entry might look like this:

   ```
   secondo        28893/tcp        #secondo server
   ```

   where 28893 is to be replaced by the port number found in step 1.
4. Change to the `$SECONDOHOME/ClientServer` directory.

Now you can start the database server: Type `DBServer`. After some SHORE messages, the DBServer prompt appears in the server, as shown in Figure 4. Now the server is ready to answer client requests. In version 1.0, two commands are supported:

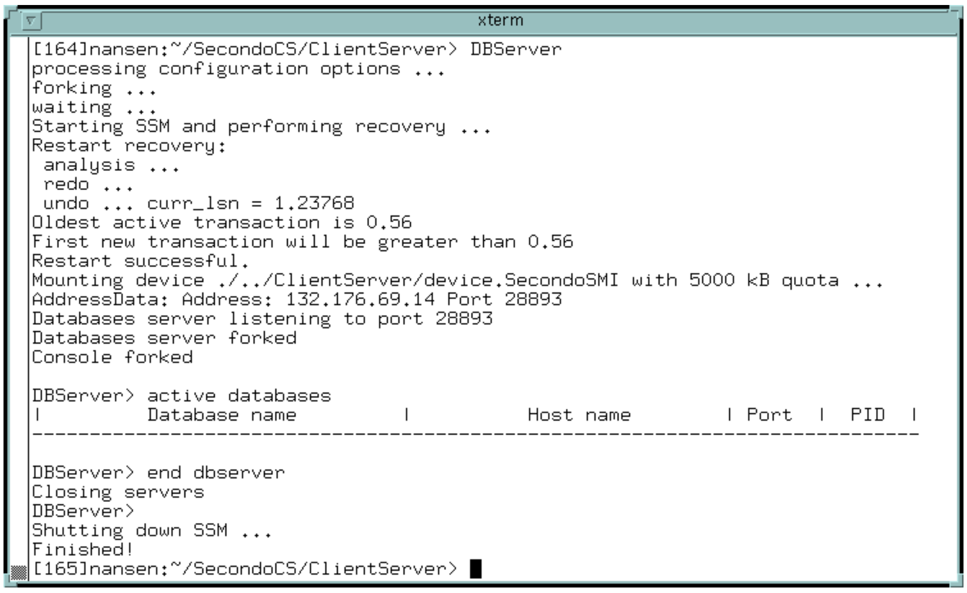
active databases prints information on all databases which are actually open due to client requests

end dbserver closes all open databases and terminates the server process. Since the server does not check for connected clients, this command should be used very carefully!

At the end of a DBServer session, again some SHORE messages are printed to screen.

7.3.2 SecondoTTYCS

SecondoTTYCS is a client version of the single-threaded SecondoTTY described in Section 7.2. 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 SecondoTTY. All commands of Section 3 work in the same way as with the single-threaded user interface.



```
[164]nansen:~/SecondoCS/ClientServer> DBServer
processing configuration options ...
forking ...
waiting ...
Starting SSM and performing recovery ...
Restart recovery:
 analysis ...
 redo ...
 undo ... curr_lsn = 1.23768
Oldest active transaction is 0.56
First new transaction will be greater than 0.56
Restart successful.
Mounting device ./../ClientServer/device.SecondoSMI with 5000 kB quota ...
AddressData: Address: 132.176.69.14 Port 28893
Databases server listening to port 28893
Databases server forked
Console forked

DBServer> active databases
|          Database name         |           Host name        | Port  |  PID  |
-----------------------------------------------------------------------------

DBServer> end dbserver
Closing servers
DBServer>
Shutting down SSM ...
Finished!
[165]nansen:~/SecondoCS/ClientServer>
```

Figure 4: DBServer Sample Session

To start SecondoTTYCS, change to the directory `$SECONDOHOME/UserInterfaces/SecondoTTYCS`, and type `SecondoTTYCS`. Remember that the database server (Section 7.3.1) must be running.

7.3.3 SecondoJava

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

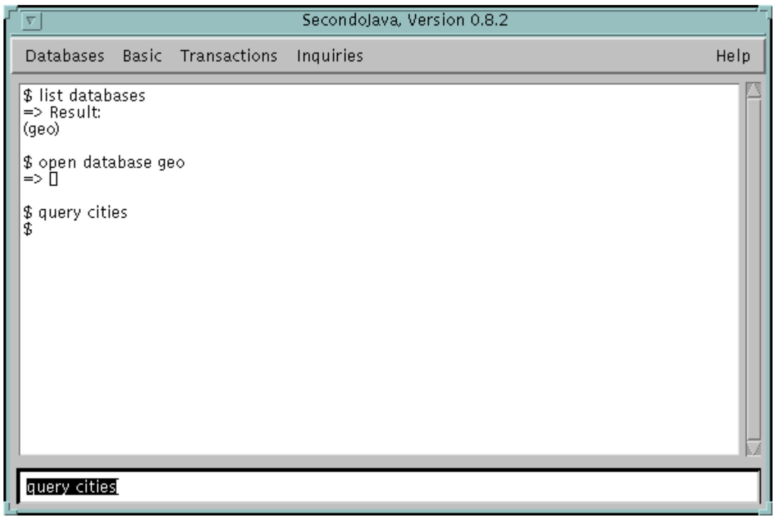
- SecondoJava can be executed in any system in which a Java virtual machine is installed.
- SecondoJava can be started as a java applet by WWW browsers supporting Java 1.1.
- The display functions used for representing the different data types are configurable and extensible. New display functions for new data types can be added by registering them in the configuration file.

Using SecondoJava

Before SecondoJava can be executed, you have to edit the file `SecondoInterface.cfg`. Set the host address and port number entries, `DatabasesServerAddress` and `DatabasesServerPort`, to the correct values (see Section 7.3.1).

After that, `SecondoJava` is ready to be run. The easiest way to start it is calling the `SJstart` script. Remember to start the `DBServer` process before executing the script. If `SJstart` returns an error message reporting that some classes have not been found, you should modify the script by setting the variable `JAVADIR` to the actual path of the directory in which the Java Development Kit is installed.

On calling `SJstart`, three windows will appear on the screen: the graphical results window, the textual results window, and the SecondoJava main window shown in Figure 5. The main window


SecondoJava, Version 0.8.2
Databases Basic Transactions Inquiries Help

```
$ list databases
=> Result:
(geo)

$ open database geo
=> []

$ query cities
$
```

```
query cities
```

Figure 5: SecondoJava Main Window

handles user input and messages from the system; the textual results window shows the results of queries. In version 0.8.2, the graphical results window is not used.

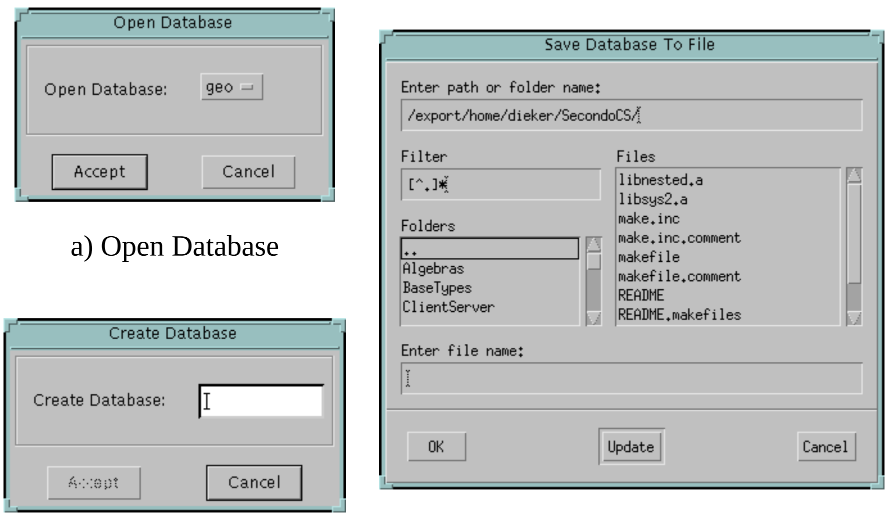
The main window basically consists of three parts:

- the *menu bar* at the top,
- the *system messages area* in the middle,
- the *command area* at the bottom of the window.

Initially, a prompt symbol “$” appears in the system messages area indicating that the system is ready for accepting any user command. Any command that can be executed in `SecondoTTY` or `SecondoTTYCS` can also be executed in `SecondoJava` by typing it in the command area. On execution, the typed command is shown in the system messages area. For all but query commands, the result returned by a command is printed in the system messages area, too. For query commands, however, results are returned in the textual results window. After a command has been executed, a new prompt indicates that the system is ready to accept new commands.

Apart from typing the SECONDO commands directly into the input area, one can also use the menu bar in the main window. This menu bar provides a user friendly interface for non-expert SECONDO users. The menu bar has four main menus: `Databases`, `Basics`, `Transactions` and `Inquiries`. Using the `Databases` menu the user has access to the database commands (e.g., `open`, `close`, `create`, `save`) as well as to the `exit` option for finishing the execution of `SecondoJava`. Using the `Basics` menu the user has access to the commands related with creation, update, etc. of types and objects, as well as execution of queries. The `Transactions` menu provides control over the transaction management commands. Finally, the `Inquiries` menu gives access to the “configuration and catalog inquiries” commands.

For instance, to open the database `geo` one can click on the `Databases` menu item and choose `Open database`.... After that, the dialog box depicted in Figure 6a) allows one to select an existing database to be opened.


Open Database
Open Database: geo
Accept Cancel

Create Database
Create Database:
Accept Cancel

Save Database To File
Enter path or folder name: /export/home/dieker/SecondoCS/
Filter: [^.]*
Folders: .. Algebras BaseTypes ClientServer
Files: libnested.a libsys2.a make.inc make.inc.comment makefile makefile.comment README README.makefiles
Enter file name:
OK Update Cancel

a) Open Database

b) Create Database

c) Save Database

Figure 6: Some SecondoJava Dialog Boxes

In the same way, all other SECONDO commands can either be executed by writing the entire command into the input area or through the menu. Whenever the menu is used and some extra information needs to be provided for the corresponding command, a dialog box is shown where the user can fill in the required data. Whenever appropriate, a selection item with all possible options is provided. For example, Figures 6b) and 6c) show the dialog boxes for the `Delete database`... and `Save database`... menus, respectively.

When executing query commands, their result is shown in the text results window. It is always possible to create a copy of a text result window, so we can keep the result of a query for later use. For this purpose, the user has to click in the text results window and a menu will appear allowing the user either to *clone* the corresponding text result window or to *delete* it. Notice that only clones can be deleted.

Making SecondoJava a WWW Applet

SecondoJava can not only be run as a standalone program, but also as an applet of a Web page (although compatibility with the different browsers has not yet been deeply checked). For this purpose, code similar to the following should be added to the HTML code defining the page:

```
<applet code= "sj.gui.SecondoJavaMain.class"
 archive= "lib/visualrt.zip, lib/Cup_v10i_runtime.jar"
 width=0 height=0>
<hr>
Your browser is not Java-enabled, so it is not
possible to execute SecondoJava on it.
<hr>
</applet>
```

This instructs the browser to load the class `sj.gui.SecondoJavaMain.class` and to search `lib/visualrt.zip` and `lib/Cup_v10i_runtime.jar` for Java classes. In this example the Web page is supposed to be located in the same directory as the code of `SecondoJava` and the configuration file `SecondoJava.cfg`. An example Web page is given in the file `SJ.html`.

Due to the security restrictions of Java v1.1 in browsers, the database server must be running on the same host at which the Web page is located. Moreover, it is not possible to save/restore a database to/from a local file.

Advanced SecondoJava Configuration Options

Apart from configuring the address of the SECONDO server to which the `SecondoJava` program must connect, the `SecondoJava.cfg` file allows us to specify which display functions are used for each data type. An example of a configuration file is given as follows.

```
# SecondoJava v0.8.2

# This file is the configuration file used by SecondoJava for knowing at
# which server address and which port the databases server is listening.
# The DatabasesServerAddress parameter tells SecondoJava in which host
# is the databases server being executed.
DatabasesServerAddress = robinson.fernuni-hagen.de

# The DatabasesServerPort parameter tells SecondoJava at which port in the
# address indicated by DatabasesServerAddress should it try to connect for
# connecting with the databases server.
DatabasesServerPort = 28893

# Now the display classes used for displaying each data type are regis-
tered.
# First, the prefix used for the display classes entries.
TextDisplayClassPrefix = TDCP_

# and second all the entries defining the display class used for each data
# type.
TDCP_int = sj.display.DisplayInt
TDCP_real = sj.display.DisplayReal
TDCP_bool = sj.display.DisplayBoolean
TDCP_string = sj.display.DisplayString
TDCP_rel = sj.display.DisplayRelation
TDCP_tuple = sj.display.DisplayTuples
```

The `DatabasesServerAddress` and `DatabasesServerPort` entries allow the user to configure the address to which the `SecondoJava` program must connect. The `TextDisplayClassPrefix` entry allows one to define which prefix will be used to identify the entries configuring the display functions of a data type. In the example above it is set to `TDCP_`, so any entry in the configuration file beginning with this string will be considered a display function configuration entry. The rest of the name after the `TDCP_` prefix defines the data type to which the entry refers. The string at the right-hand side of the entry defines the class that will be used for displaying the values of the given data type. For example, in the configuration file above `SecondoJava` is configured to use the class `sj.display.DisplayInt` for displaying `int` values.

Any class used for displaying a data type must implement the interface `sj.display.TextDisplayType`, and must be located in the classes path given when executing `SecondoJava` (option `-classpath` when calling the Java interpreter).

If for a given data type no entry is provided, a generic display function will be used.

8 Algebra Modules

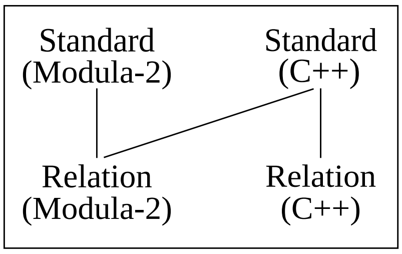
8.1 Overview

Secondo release 1.0 contains four algebra modules:

- A standard algebra implemented in Modula-2. Directory: `$SECONDOHOME/Algebras/Standard-Modula`.
- A standard algebra implemented in C++. Directory: `$SECONDOHOME/Algebras/Standard-C++`.
- A relation algebra implemented in Modula-2. Directory: `$SECONDOHOME/Algebras/Relation-Modula`.
- A relation algebra implemented in C++. Directory: `$SECONDOHOME/Algebras/Relation-C++`.

All modules are operational and can be activated and deactivated according to the guidelines given in Section 6. In release 1.0, both C++ modules are activated by default. Notice that the relation value representation provided by the Modula-2 relational algebra module is not persistent: relations can be created, updated, and queried, but relation objects do not survive the end of a session.

In the figure at the right, edges between modules represent legal module configurations. All other configurations are illegal: We cannot activate both standard algebra modules at the same time because of conflicting type names; the same holds with relational algebra modules. Furthermore, it is not possible to join the C++ relational algebra with the Modula-2 standard algebra due to additional inter-algebra interface requirements which are not met by the Modula-2 standard algebra implementation.



8.2 Standard Algebra (C++)

8.2.1 Standard Type Constructors

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

int is a type for integer values: the domain is that of the `int` type implemented by the C++ compiler, typically -2147483648 to 2147483647.

real is a type for floating point values: the domain is that of the `float` type implemented by the C++ compiler.

bool values are either TRUE or FALSE.

string is a value consisting of a sequence of up to 48 ASCII characters.

Each standard data type contains an additional flag specifying 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.

8.2.2 Standard Operators

Table 3 shows all operators provided by the standard algebra module (C++). For each operator we list its type mapping (several in case of overloaded operators), its SOS syntax specification, and a short description of its semantics.

Operator	Type Mapping	Syntax	Semantics
+	int × int → int	_#_	addition
	int × real → real		
	real × int → real		
	real × real → real		
-	int × int → int	_#_	subtraction
	int × real → real		
	real × int → real		
	real × real → real		
*	int × int → int	_#_	multiplication
	int × real → real		
	real × int → real		
	real × real → real		
/	int × int → real	_#_	division
	int × real → real		
	real × int → real		
	real × real → real		
mod	int × int → int	_#_	modulo
div	int × int → int	_#_	integer division
<	int × int → bool	_#_	less
	int × real → bool		
	real × int → bool		
	real × real → bool		
<=	int × int → bool	_#_	less equal
	int × real → bool		
	real × int → bool		
	real × real → bool		
>	int × int → bool	_#_	greater
	int × real → bool		
	real × int → bool		
	real × real → bool		
>=	int × int → bool	_#_	greater equal
	int × real → bool		
	real × int → bool		
	real × real → bool		
=	int × int → bool	_#_	equal
	int × real → bool		
	real × int → bool		
	real × real → bool		
#	int × int → bool	_#_	not equal
	int × real → bool		
	real × int → bool		
	real × real → bool		

Table 3: Standard operators

Operator	Type Mapping	Syntax	Semantics
starts	string × string → bool	_#_	Returns TRUE if the first argument string begins with the substring passed as second argument, otherwise FALSE.
contains	string × string → bool	_#_	Returns TRUE if the first argument string contains the substring passed as second argument, otherwise FALSE.
not	bool → bool	#(_)	logical not
and	bool × bool → bool	_#_	logical and
or	bool × bool → bool	_#_	logical or

Table 3: Standard operators

8.2.3 Query Examples (Standard Algebra)

Notice that queries can only be passed to system after the user has opened a database as described in Section 5. Remember the query command:

```
query <value expression>
```

where `value expression` is a term over the active algebra(s). For instance, try the queries given in Table 4 to see how to use the standard algebra.

SOS Syntax	Nested List Syntax
`query 5`	`(query 5)`
`query 5.0 + 7`	`(query (+ 5.0 7))`
`query 3 * 4 + 2`	`(query (* 3 (+ 4 2))`
`query (3 * 4) + 2`	`(query (+ (* 3 4) 2))`
`query 6 < 8`	`(query (< 6 8))`
`query ("Secondo" contains "cond") and TRUE`	`(query (and (contains "Secondo" "cond") TRUE))`

Table 4: Query examples (standard algebra)

8.3 Relational Algebra (C++)

8.3.1 Relational 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	<u>*int*</u>, <u>*real*</u>, <u>*string*</u>, <u>*bool*</u> (from standard algebra)
$(\text{IDENT} \times \text{DATA})^+$	→ TUPLE	<u>*tuple*</u>
TUPLE	→ REL	<u>*rel*</u>

So a tuple is a list of one or more (identifier, attribut type) pairs; 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: `name` of type `string` and `pop` of type `int`. A value of this type in nested list representation is a list containing lists of attribute values, e.g.

```
(
 ("New York" 7322000)
 ("Paris" 2175000)
 ("Hagen" 212000)
)
```

8.3.2 Relational Operators

Table 5 shows all operators provided by the relational algebra module (C++). For each operator we list its type mapping (several in case of overloaded operators), its SOS syntax specification, and a short description of its semantics. Notice that type mapping entries are not *second-order signatures* as defined in the SOS formalism, but simplified signatures for fast lookups, sometimes exceeding the formal limits. A formal extension we use is the binary "@" operator for list concatenation.

Operator	Type Mapping	Syntax	Semantics
feed	rel(tup) → stream(tup)	_#	Produces a stream of tuples from a relation
gfeed	rel(tup) → stream(tup)	_#	Like feed, to be used in groupby parameter functions
consume	stream(tup) → rel(tup)	_#	Produces a relation from a stream of tuples.
count	stream(tup) → int	_#	Counts the number of stream elements.
head	stream(tup) × int → stream(tup)	_#[_]	Passes through the number of input stream elements specified as parameter only.
extract	$\text{stream(tuple}([a_1: d_1, ..., a_n: d_n])) \times a_i \rightarrow d_i$		Returns the value of attribute a_i of the first tuple in the input stream.
cancel	stream(tup) × (tup → bool) → stream(tup)	_#[_]	Lets pass the input tuples until the parameter function evaluates to TRUE.
filter	stream(tup) × (tup → bool) → stream(tup)	_#[_]	Lets pass those input tuples for which the parameter function evaluates to TRUE.
project	$\text{stream(tuple}([a_{11}: d_1, ..., a_{1n}: d_n])) \times [a_{21}, ..., a_{2k}] \rightarrow \text{stream(tuple}([a_{21}: d_1, ..., a_{2k}: d_k]))$	_#[_]	relational project operator
extend	$\text{stream(tuple(x))} \times [(a_1, (\text{tuple}(x) \rightarrow d_1)), ..., (a_n, (\text{tuple}(x) \rightarrow d_n))] \rightarrow \text{stream(tuple}(x@[a_1: d_1, ..., b_n: d_n]))$	_#[_]	Extends each input tuple by new attributes as specified in the parameter list.
product	stream(tuple(x)) × stream(tuple(y)) → stream(tuple(x@y))	_ _#	relational product operator
product0	stream(tuple(x)) × stream(tuple(y)) → stream(tuple(x@y))	_ _#	relational product operator
loopjoin	stream(tuple(x)) × stream(tuple(y)) × (tuple(x) × tuple(y) → bool) → stream(tuple(x@y))	_ _#[_]	relational join operator
loopjoin1	stream(tuple(x)) × stream(tuple(y)) × (tuple(x) × tuple(y) → bool) → stream(tuple(x@y))	_ _#[_]	relational join operator
mergejoin	stream(tuple(x)) × stream(tuple(y)) × (tuple(x) × tuple(y) → bool) → stream(tuple(x@y))	_ _#[_]	relational join operator
concat	stream(tuple(x)) × stream(tuple(x)) → stream(tuple(x))	_ _#	union (without duplicate removal).
mergesec	stream(tuple(x)) × stream(tuple(x)) → stream(tuple(x))	_ _#	intersection
mergediff	stream(tuple(x)) × stream(tuple(x)) → stream(tuple(x))	_ _#	difference
groupby	$\text{stream(tuple(x))} \times [(a_1, (x \rightarrow d_1)), ..., (a_n, (\rightarrow d_n))] \rightarrow \text{stream(tuple}(a_1: d_1, ..., a_n: d_n))$	_#[_]	SQL-style groupby
aggr	$\text{stream(tuple}([a_1: d_1, ..., a_n: d_n])) \times a_i \times d_i \times (d_i \times d_i \rightarrow d_i) \rightarrow d_i$		Behaves like a fold operator applied to a list of values of type d_i. Parameter a_i specifies which attribute value is considered.

Table 5: Relational operators

Operator	Type Mapping	Syntax	Semantics
sortby	stream(tuple([a_1: d_1, ..., a_n: d_n])) × [(a_{i1}, {asc, desc}), ..., (a_{im}, {asc, desc})] → stream(tuple([a_1: d_1, ..., a_n: d_n]))	_#[_]	SQL-style sortby
sort	stream(tuple(x)) → stream(tuple(x))	_#	Sorts the input tuple stream.
rdup	stream(tuple(x)) → stream(tuple(x))	_#	Removes duplicates from the *sorted* input stream.
attr	tuple([a_1: d_1, ..., a_n: d_n]) × a_i → d_i	#(_, _)	Extracts an attribute value from a tuple. To be used in parameter functions.
avg	stream(tuple([a_1: d_1, ..., a_i: int, ..., a_n: d_n])) × a_i → real	_#[_]	Returns the arithmetic mean of the attributes a_i of all input tuples.
	stream(tuple([a_1: d_1, ..., a_i: real, ..., a_n: d_n])) × a_i → real		
max	stream(tuple([a_1: d_1, ..., a_i: int, ..., a_n: d_n])) × a_i → int	_#[_]	Returns the maximum value of the attributes a_i of all input tuples.
	stream(tuple([a_1: d_1, ..., a_i: real, ..., a_n: d_n])) × a_i → real		
min	stream(tuple([a_1: d_1, ..., a_i: int, ..., a_n: d_n])) × a_i → int	_#[_]	Returns the minimum value of the attributes a_i of all input tuples.
	stream(tuple([a_1: d_1, ..., a_i: real, ..., a_n: d_n])) × a_i → real		
sum	stream(tuple([a_1: d_1, ..., a_i: int, ..., a_n: d_n])) × a_i → int	_#[_]	Returns the sum of the values of the attributes a_i of all input tuples.
	stream(tuple([a_1: d_1, ..., a_i: real, ..., a_n: d_n])) × a_i → real		
rename	stream(tuple([a_1: d_1, ..., a_n: d_n])) × a_r → stream(tuple([a_r a_1: d_1, ..., a_r a_n: d_n]))		Type operator: Renames all attribute names by preceding them with the prefix passed as parameter.
TUPLE	stream(tuple(x)) × ... → tuple(x)		Type operator: Returns the tuple type of the first argument stream.
TUPLE2	stream(tuple(x)) × stream(tuple(y)) × ... → tuple(y)		Type operator: Returns the tuple type of the second argument stream.

Table 5: Relational operators

8.3.3 Query Examples (Relational Algebra)

In Table 6, we present example queries using standard and relational types and operators. We assume the existence of two relations:

- `cities: rel(tuple([cname: string, pop : int, worthavisit: bool]))`
- `plz: rel(tuple([PLZ: int, Ort: string]))`

Either relation object is defined in the `geo` file coming with release 1.0.

SOS Syntax	Nested List Syntax
`query [const rel(tuple([nr: int])) ((1)(2)(3)(4))]`	`(query((rel(tuple((nr int))))((1)(2)(3)(4))))`
`query cities feed filter[2 > 1] consume`	`(query(consume(filter((feed cit-ies)(fun (c city)(> 2 1))))))`
`query [const rel(tuple([cname: string, pop: int, wav: bool])) (("Ddorf" 200 TRUE))]`	`(query ((rel (tuple ((cname string) (pop int) (wav bool)))) (("Ddorf" 200 TRUE))))`
`query cities feed filter[not(.cname = "muenchen")] consume`	`(query (consume (filter (feed cities) (fun (tuple1 TUPLE) (not (= (attr tuple1 cname) "muenchen"))))))`

Table 6: Query examples (relational algebra)

SOS Syntax	Nested List Syntax
query cities feed filter [not(.cname = "muenchen")] [const rel (tuple([(cname, string), (pop, int), (worthavisit, bool)])) (("Muenchen" 200 TRUE))] feed concat consume	(query (consume (concat (filter (feed cities) (fun (tuple1 TUPLE) (not (= (attr tuple1 cname) "muenchen")))) (feed ((rel (tuple ((cname string) (pop int) (worthavisit bool)))) (("Muenchen" 200 TRUE)))))))
query cities feed filter[not(.cname = "muenchen")] [const rel (tuple([cname: string, pop: int, worthavisit: bool])) (("Muenchen" 200 TRUE))] feed concat consume	
update cities := cities feed filter[not(.cname = "muenchen")] [const rel (tuple([(cname, string), (pop, int), (worthavisit, bool)])) (("Muenchen" 200 TRUE))] feed concat consume	(update cities := (consume (concat (filter (feed cities) (fun (tuple1 TUPLE) (not (= (attr tuple1 cname) "muenchen")))) (feed ((rel (tuple ((cname string) (pop int) (worthavisit bool)))) (("Muenchen" 200 TRUE)))))))
query cities feed sortby[pop asc] consume	(query (consume (sortby (feed cities) ((pop asc)))))
query cities feed sortby[pop asc] groupby[cname, pop; number: group gfeed count] consume	(query (consume (groupby (sortby (feed cities) ((pop asc))) (cname pop) ((number (fun (group1 TUPLE) (count (gfeed group1))))))))
query plz feed filter[(.PLZ > 44000) and (.PLZ < 45000)] groupby[Ort; Anzahl: group gfeed count] consume	(query (consume (groupby (filter (feed plz) (fun (tuple1 TUPLE) (and (> (attr tuple1 PLZ) 44000) (< (attr tuple1 PLZ) 45000)))) (Ort) ((Anzahl (fun (group2 TUPLE) (count (gfeed group2))))))))

Table 6: Query examples (relational algebra)