SECONDO

Version 4.2
User Manual [1]

November 12, 2019

Ralf Hartmut Güting, Fabio Valdés, Holger Helmut Hennings, Jan Kristof Nidzwetzki, Florian Heinz, Thomas Behr

FernUniversität in Hagen

Faculty for Mathematics and Computer Science
Database Systems for New Applications
59084 Hagen, Germany

[1]This work was partially supported by a grant Gu 293/8-2 from the Deutsche Forschungsgemeinschaft (DFG), project "Datenbanken für bewegte Objekte" (Databases for Moving Objects)

Contents

1 Introduction

SECONDO is a DBMS prototype built with a focus on extensibility and support of spatial and spatio-temporal data. In fact, the desire to support spatial data types and their operations led to the design of SECONDO's extensible architecture.

Spatial data types such as *point*, *line*, and *region* allow one to represent geometries such as points, curves, or areas in the Euclidean plane, for example, the location of a lighthouse, a road, or a forest. Operations allow one to do calculations with such objects, for example, determine whether the lighthouse is inside a forest, or compute the part of a road within a forest.

Early work [Güt88] describes the integration of spatial data types into a relational data model, the so-called georelational algebra. Spatial data types can be used as attributes within a relation and relational operations such as selection or join can use spatial operations on attribute values. Meanwhile, all major DBMS implementations support spatial data types.

To integrate spatial data types into a DBMS implementation, one needs to implement data structures for the types and methods (algorithms) for the operations. In addition, one needs special types of index structures (such as R-trees) for indexing spatial data and specialized algorithms to perform spatial joins.

It would not be a good idea to build a DBMS with just these extensions. There are many other applications that may be supported by a specialized set of data types and operations, that is, an algebra. For example, one may want to store images, music, text documents, molecules, CAD shapes, all with their respective operations and indexing techniques. Hence one should build a DBMS extensible at least by data types, operations, and index structures.

What we have described so far is a relational or object-relational model extensible by abstract data type packages (algebras). This is to some extent implemented in current open-source or commercial DBMS. On the other hand, there is also a lot of interest in systems supporting data models going beyond the relational model such as XML or graph databases.

1.1 The Secondo Kernel

SECONDO takes a more radical view and structures the entire execution system (called the kernel) as a set of algebras. Hence we not only have algebras for attribute types but also to represent relations with relational operations, indexes with their creation, update and search operations. Moreover, there are algebras providing alternatives to the relational model such as graphs or nested relations.

The SECONDO kernel is generic in the sense that it does not support a specific DBMS data model. Instead, it is able to store data of arbitrary types implemented in any of the available algebras. The kernel is an engine to evaluate arbitrary expressions over the data and operations of the available algebras. Operations are typed so that the engine can check for correctness of expressions.

The SECONDO kernel manages a set of databases. Each database contains a set of named types and named objects. A named object consists of a name, a type, and a value of this

type. The kernel offers seven basic commands to manipulate types and objects:

1. type <*identifier*> = <*type expression*>
2. delete type <*identifier*>
3. create <*identifier*> : <*type expression*>
4. update <*identifier*> := <*value expression*>
5. let <*identifier*> = <*value expression*>
6. delete <*identifier*>
7. query <*value expression*>

Commands 1 - 2 define and delete types. Commands 3 - 6 create and delete objects and assign values to them. Command 7 evaluates an expression and returns the result to the user interface. Type expressions are built from type constructors provided by available algebras and value expressions are built from object names, constants of available types, and operations of available algebras.

Note that a value expression includes what is known in other systems as a query plan, usually represented as an operator tree over query processing operations and generated by the query optimizer. In SECONDO, a user can write query plans directly as expressions. Hence one can write, for example:

```
query 3 * 5
query Employees feed filter[.Age > 35] consume
```

In both cases, the expressions are evaluated and the result is returned to the user interface. The latter is an example of a simple query plan implementing relational selection. Details will be explained in later sections. We call the expressions, viewed as a language for writing queries, the executable language of SECONDO.

The SECONDO kernel has a simple text-oriented user interface (SecondoTTYBDB, SecondoTTYCS) in a shell window. One can type basic commands as shown above, open and close databases, import and export data, manage transactions and so forth.

The kernel is implemented in C++. It relies on several open source software components, in particular BerkeleyDB as a storage manager providing shared files and transaction management. Like the SECONDO components Javagui and optimizer, yet to be discussed, it runs on Linux and MacOS X platforms.

One can use the kernel with its simple user interface either in single user or in multi-user mode. In the first case, all software components are linked together in a single process, in the latter, they run as several processes as shown in Figure 1. Here, a TTY interface first connects to a monitor process which starts a SECONDO kernel instance on its behalf. The SECONDO instances are synchronized via their BerkeleyDB modules.

1.2 The Optimizer

The second major component of SECONDO is the query optimizer. In contrast to the kernel, it is restricted to a relational model extended by algebras for attribute types. This is partly due to the fact that SQL itself is closely tied to the relational model.

The optimizer supports a basic part of SQL such as the select - from - where statement, grouping, sorting, and aggregation, but no subqueries. The where-clause is written as a list of conditions; the meaning is their conjunction. Data definition and update commands are also supported.

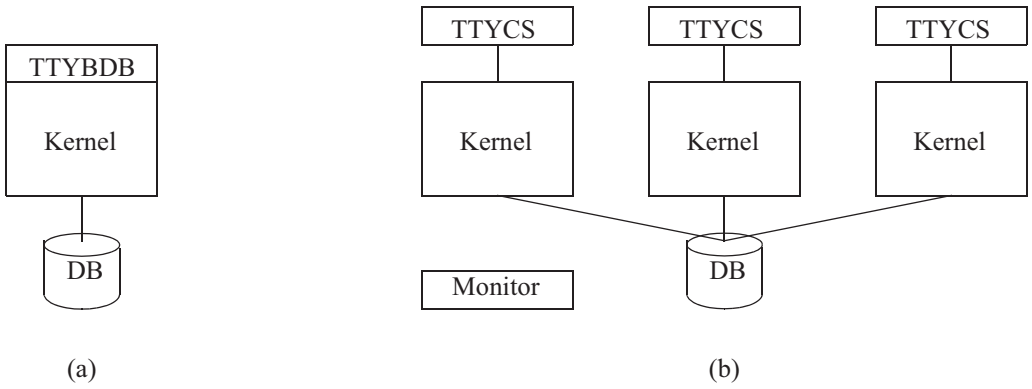

Figure 1: (a) Single User and (b) Multi-User Setup

The optimizer is implemented in Prolog, as this language is particularly suitable to implement rule-based generation and enumeration of query plans. A novel algorithm for cost-based query optimization is used. Selectivities for selections and joins, needed for cost estimation, are determined by sampling, using materialized samples of database relations.

The optimizer creates a query plan in SECONDO's executable language. Any such plan can also be typed by the user directly. Explicit textual plans also help understanding what the optimizer does.

The optimizer is extensible to support new algebras for attribute types. This includes adding optimization rules to create specialized index accesses and join methods and associated cost functions.

The SQL dialect understood by SECONDO is implemented in the optimizer directly in Prolog which means that an SQL query is understood as a Prolog term. This leads to some peculiarities in the notation. Relation and attribute names need to be written in lower case because Prolog interprets words starting with a capital as variables. Instead of a dot in a qualified name such as s.name one needs to use a colon and write s:name. These restrictions apply when the original user interfaces for the optimizer are used (SecondoPL, SecondoPLCS).

However, recently user interfaces are available (SecondoPLTTY, SecondoPLTTYCS) which do some preprocessing of a query so that relation and attribute names can be written in the same way as in the kernel and also the dot notation may be used. Moreover, these interfaces also allow one direct access to the kernel so that commands can be written as in the kernel interfaces. Figure 2 shows some configurations involving the optimizer.

1.3 The Javagui

The third major component of SECONDO is the graphical user interface written in Java, therefore termed the Javagui. Its basic structure is shown in Figure 3.

In the command window, one can enter commands for the kernel or SQL commands for the optimizer, similar to SecondoPLTTY. Results from queries are displayed in the viewer window. The object window contains a list of queries or object names currently displayed (or at least loaded into the user interface).

The Javagui is extensible by viewers that can be specialized to display particular data types. Therefore the bottom part changes according to the selected viewer. Viewers exist, for

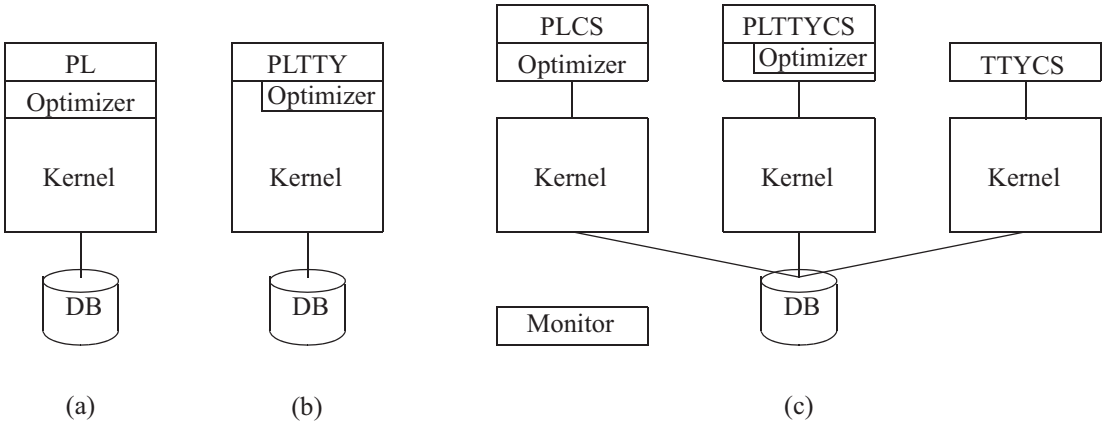

Figure 2: Configurations involving the optimizer. (a) and (b) Single User Mode. (c) Multi-User Mode

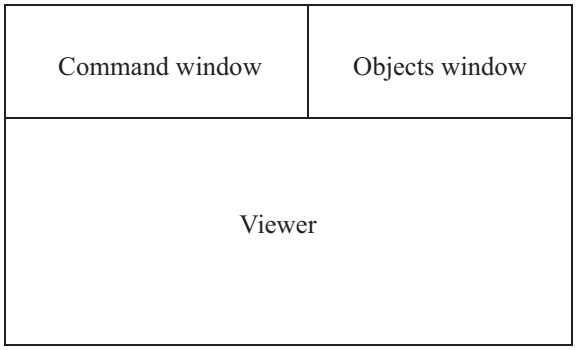

Figure 3: Structure of the Javagui

example, to represent relations in a tabular form, to show relations with spatial objects displayed in a graphics window with a map background, to show moving objects (spatio-temporal data) and animate their movement, to display large text documents (pdf) or images, to play audio data, to update relations and format their data, or even to display chess games.

The Javagui cannot be linked together with the kernel or optimizer but only used in client-server (multi-user) mode. This is shown in Figure 4. The Javagui can either communicate with a kernel only as shown left. Of course, in this case only kernel commands are supported. Or it may also use an optimizer server which works as a server for the Javagui and as a client to the kernel. In this case, SQL commands are sent to the optimizer which computes a query plan and sends it back to the Javagui. The optimizer server talks itself to the kernel to get information about the database, compute selectivities, etc. The Javagui then sends the plan to the kernel as if it had been typed directly by the user. The optimizer server can in fact be added or removed while the Javagui is running.

In contrast to kernel and optimizer, the Javagui can also be used in a Windows environment. Hence it is possible to have SECONDO servers run on Linux or MacOS X and to access them from a Javagui under Windows.

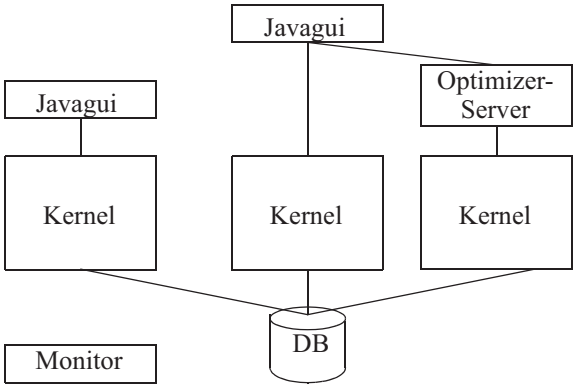

Figure 4: Using the Javagui with or without Query Optimizer

1.4 The Two Language Levels

SECONDO is to our knowledge unique in providing the executable language to formulate queries, in addition to SQL as offered by DBMS in general. The question is which level to use when. Both levels have advantages and disadvantages:

- The executable level is more complex to use and requires detailed knowledge of the available query processing operations. The user also needs to select the best operations for a query and to arrange them in a good order. This is normally the job of the optimizer and may be impossible for a complex query.
- On the other hand, simple queries can be written easily in executable language. The user has full control of the steps of manipulating data. The full power of the kernel system is available, even the most recent additions of type or index structures may be used. It is possible to use other structures than relations and indexes on disk. For example, main memory relations and indexes, graphs and other data structures in memory, column-oriented relation representations, or nested relations are all available at the executable level in the current SECONDO, but not (yet) in SQL.
- The SQL level is much easier to use. It provides cost-based optimization and may therefore construct better plans than even an expert SECONDO user. It is able to construct quite sophisticated plans for certain queries or commands.
- On the other hand, the SQL level is restricted to the relational model and the shape of queries supported by SQL. Advanced additions to the kernel require extra work to be integrated into the optimizer if that is possible in the SQL framework. The development of the optimizer therefore lags behind the development of the kernel.

The executable language can be characterized as being halfway in complexity and expressive power between SQL and a programming language such as C++. In fact, one can write complex "programs" in SECONDO executable language. These can be stored as scripts in files and be executed by SECONDO. Examples of such scripts are the BerlinMOD benchmark [DBG09] (e.g. `BerlinMOD_CreateObjects.SEC` and other scripts) or the construction of a road network from OpenStreetMap data (`OrderedRelationGraphFromFullOSMImport.SEC`) in the directory `secondo/bin`.

1.5 This Manual

This manual explains in some detail the use of SECONDO. It is structured as follows. Section 2 describes installation on various platforms. Section 3 explains how to start SECONDO for the various configurations discussed above. Section 4 describes in more detail the concepts for querying in executable language as well as in SQL. Section 5 shows for both language levels how typical tasks can be expressed such as creating relations, updating and querying them, with or without indexes. Data import and export is also covered. Section 6 is devoted to the Javagui and its most important viewers. Finally, Section 7 addresses configuration issues.

2 Installation

2.1 Trying Secondo

For trying SECONDO, it is possible to use a precompiled SECONDO within a virtual machine. As a first step, the Workstation player (formerly VMware player) must be installed. For downloading this software and further instructions see `www.vmware.com`.

After installing the Workstation Player, visit the SECONDO website `dna.fernuni-hagen.de/secondo`. On the left side, there is the item *VM Appliance* in the section *Other Downloads*. Click on this link and download the latest version of the VM appliance. After downloading the file, unpack it using your favored unpacking tool.

Now, start the Workstation player and select *Open a Virtual Machine*. Navigate to the folder where you have unpacked the SECONDO appliance. Select the `vmx` file and click on the *Open* button, then click on *Power On*. In the following dialog select *I Copied it*. After starting the system, a password is required, enter `secondo` here. Open a terminal by clicking the appropriate symbol on the left side. From here, you can start SECONDO according to the instructions from Section 3.

2.2 Full Secondo Installation

SECONDO is available as source code and must be built from these sources before it can be used. To enable the compilation, several tools and libraries are required. Furthermore, some system variables must be set. In the following the set of required tools, libraries, and variables is called SECONDO Development Kit, SDK for short.

For several operating systems, we provide bash scripts executing the bulk of the work of installing the SDK. Note that these scripts exploit the standard package manager of the system implying that root access to the system is required. If no root access is available, ask the system administrator to run the script or install the required tools and libraries locally (see Section 2.2.3).

2.2.1 SDK Installation on openSUSE, Fedora, and Ubuntu

The procedure is very similar for all of these distributions. Visit the SECONDO website `dna.fernuni-hagen.de/secondo` and click on *Installation Instructions* on the left side. Download the appropriate installation script. Then open a terminal and run the script by entering `bash <name of the script>`. The root password will be requested. After some time, all required tools are installed and a file `.secondorc` has been created in the home directory.

On SUSE systems, a manual installation of the BerkeleyDB library is required, since the distribution does not provide a version that is built with compiling options required by SECONDO. For installing it, just follow the instructions printed at the end of the installation script.

In the file `.secondorc`, several environment variables are set. To activate these settings, open the file `.bashrc` located in the home directory with an editor of your choice and add the line

```
source $HOME/.secondorc $HOME/secondo
```

at the end of this file. After saving this change, close the current terminal and open a new one. You can check the success by entering

```
echo $SECONDO_BUILD_DIR
```

resulting in the output `/home/<user>/secondo`.

Download the newest SECONDO version from the website (select *Sources* on the left side). Unpack the version into your home directory.

2.2.2 SDK Installation on macOS

On macOS platforms, a lot of the required tools are part of the *xcode command line tools*. For installing these tools, open a terminal and enter

```
gcc --version
```

If *xcode* is already installed, the compiler's version is displayed. Otherwise, a window pops up informing that xcode is necessary for the gcc-command. Just click on the *install* button and wait for some time.

For compiling the graphical user interface of SECONDO, the Java Development Kit is required. Open a terminal and enter

```
javac -version
```

If the JDK is installed, its version number is displayed. Otherwise, a new window appears. By clicking on the *install* button, a web browser is opened showing Oracle's Java download page. Accept the license, download the JDK for macOS, and install it following the given instructions.

The remaining required tools are collected in a file available on the SECONDO website. On the website `dna.fernuni-hagen.de/secondo`, select *Installation Instructions* on the left side. After that scroll down until *Installation of the Secondo SDK on OS X platforms*. Download the package for your macOS version. If it is not unpacked automatically, unpack this file. Now open a terminal, navigate to the folder containing the unpacked files, and run the script starting with `Install_On`. The remainder of the file name depends on the used macOS version. After the script finishes, open the file `.profile` located in the home directory in an editor of your choice and append the following lines:

```
export SECONDO_SDK=$HOME/secondo-sdk
export SECONDO_PLATFORM=mac_osx
export SECONDO_BUILD_DIR=$HOME/secondo
source $SECONDO_SDK/secondorc
```

Save the changes, close the terminal and open a new one.

2.2.3 SDK Installation on other Systems

Even if no SDK installation script is provided for your system, it is possible to get SECONDO to run. The next table shows all required tools and libraries. Install these tools either using the software management tool of your system or build them from scratch following the instructions coming with these tools. If the tools are installed outside the normal system paths, insert the paths to the header files into the system variable `CPLUS_INCLUDE_PATH` or `CPATH` and the paths to the libraries into the variable `LIBRARY_PATH`. If shared libraries are involved, add the paths to them to the variable `LD_LIBRARY_PATH`, too.

Tool/Library	**Edition**	**Version**
flex	dev	$\geq$ 2.5.33
bison	dev	$\geq$ 2.1
gcc/g++		$\geq$ 4.7
berkeley db	dev,c++	$\geq$ 4.3.29
libjpeg	dev	6.2
JDK	dev	$\geq$ 7
readline	dev	$\geq$ 5.2
recode	dev	$\geq$ 3.6
ncurses	dev	$\geq$ 6.0
swi-prolog	dev	$\geq$ 7.2.2
jpl	dev	$\geq$ 1.0
bash		
make		
required for certain algebras only		
gsl	dev	
xml2	dev	
boost	dev	

Beside the installation of tools, SECONDO requires the presence of some variables listed in the following table:

Variable	**Meaning**	**Example Values**
SECONDO_BUILD_DIR	home of SECONDO	/home/user/secondo
SECONDO_PLATFORM	operating system	linux, linux64, max_osx
BERKELEY_DB_DIR	home of the berkeley db	/home/user/BDB
BERKELEY_DB_LIB	name of the berkeley db lib	db_cxx
BDB_INCLUDE_DIR	include dir of berkeley db	/home/user/BDB/include
BDB_LIB_DIR	home of the berkeley db lib	/home/user/BDB/lib64
J2SDK_ROOT	home of the jdk	/usr/lib64/jvm/java-1.8.0-openjdk-1.8.0/
JAVAVER	java version	1.8
SWI_HOME_DIR	home of swi prolog	/usr/lib64/swipl-7.2.2
PL_LIB_DIR	home of swi prolog lib	/usr/lib64/swipl-7.2.2/lib/x86_64-linux/
PL_LIB	name of prolog lib	swipl
PL_INCLUDE_DIR	prolog's include dir	/usr/lib64/swipl-7.2.2/include
PL_VERSION	prolog's version	72020
JPL_DLL	path to JPL library	/usr/lib64/swipl-7.2.2/lib/x86_64-linux//libjpl.so
JPL_JAR	path to JPL java file	/usr/lib64/swipl-7.2.2/lib/jpl.jar
readline	should readline be used	true, false
SECONDO_CONFIG	path to SECONDO's config file	/home/user/secondo/bin/SecondoConfig.ini
PD_HEADER	path to PD header file	/home/user/secondo/Tools/pd/pd_header_listing

2.2.4 Building Secondo

After installing all tools required by SECONDO, it is an easy task to build the system. Just open a terminal and enter:

```
cd secondo
make
```

This will take a while, thus it is a good idea to take a pot of coffee.

Now, SECONDO is ready to be started.

3 Starting Secondo

Several ways exist to run the DBMS SECONDO for different purposes. When debugging, importing larger datasets or for smaller, isolated tasks, the single user interfaces are sufficient. For regular use, the client-server interfaces, which communicate over TCP/IP network connections, are recommended. The graphical user interface also depends on a server connection.

3.1 Prerequisites

The scripts and program binaries for starting SECONDO are located in the `bin/` directory below the SECONDO home directory or in the `Optimizer/` directory for the programs integrating the SECONDO optimizer. SECONDO looks for its configuration in the file specified in the variable `$SECONDO_CONFIG`; the default location is `$(SECONDO_BUILD_DIR)/bin/SecondoConfig.ini`. The example configuration file packaged with SECONDO is a good starting point and can be tailored to specific needs (e.g. bind IP, port).

For debugging purposes, most scripts can also be called with the options `--valgrind` for checking common errors, `--valgrindlc` for full memory leak checking, or `--profile` for profiling function calls. The tool **valgrind**, which is available with many linux distributions, has to be installed first.

3.2 Single User Interfaces

The single user interface runs the database management system and the user interface in a single process. It is crucial, that only one process at a time accesses the same database directory, which is specified in the configuration file `SecondoConfig.ini` and points to `$(HOME)/secondo-databases` by default. No other process, single user or multi user, may be running simultaneously, otherwise data corruption or crashes might occur.

3.2.1 SecondoTTYBDB

The script `SecondoTTYBDB` is a wrapper around `SecondoBDB`. After startup, a prompt waits for user input. Database queries are sent straight to the SECONDO kernel, but some operations are processed directly by the user interface. For example, use `@cmds` to read and execute database commands from a file named `cmds`. A comprehensive list of those commands is shown with the `HELP` command. All other commands are terminated either with a semicolon or with two newlines (for multi-line commands).

In the following example, the interface is started and the dump of the example database `berlintest`, which is located in the `bin/` directory, is restored:

```
db@server:/home/db$ cd secondo/bin
db@server:/home/db/secondo/bin$ SecondoTTYBDB
[ ... some informational output ... ]

Secondo TTY ready for operation.
Type 'HELP' to get a list of available commands.
Secondo => restore database berlintest from berlintest;
```

```
command
'restore database berlintest from berlintest'
started at: Mon Sep 10 12:41:24 2018

Reading file /home/db/secondo/bin/berlintest ...
Restoring types ...
Restoring objects ...
BGrenzenLine ... processed and succeeded.
Faehren ... processed and succeeded.
[ ... some more informational output ... ]
Total runtime ...   Times (elapsed / cpu): 10.847sec / 10.28sec = 1.05515
=> []
```

The database *berlintest* has now been restored and is already open (ready for queries).

```
Secondo => query mrain atinstant instant("2003-11-20-06:06");
command
'query mrain atinstant instant("2003-11-20-06:06")'
started at: Mon Sep 10 12:45:19 2018

noMemoryOperators = 0
perOperator = 0
Total runtime ...   Times (elapsed / cpu): 0.001737sec / 0sec = inf

Generic display function used!
Type : iregion
Value:
("2003-11-20-06:06"
    (
        (
            (
                (-1203.7703583062 5341.8599348534)
                (-1299.9429967427 5966.9820846906)
                (1200.5456026059 7024.8811074919)
                (2835.4804560261 6111.2410423453)
                (5528.3143322476 5582.2915309446)
                (5720.6596091205 4380.1335504886)
                (4566.5879478827 3370.3208469055)
                (2883.5667752443 3177.9755700326)
                (1056.2866449511 3177.9755700326)
                (1777.5814332248 4332.0472312704)
                (1248.6319218241 5438.0325732899)
                (-290.1302931596 5197.6009771987)))))
Secondo => close database;
command
'close database'
started at: Mon Sep 10 12:46:09 2018

Total runtime ...   Times (elapsed / cpu): 0.002482sec / 0.01sec = 0.2482

=> []
Secondo => Q
*** Thank you for using SECONDO!
```

```
db@server:/home/db/secondo/bin$
```

Besides database queries, which are sent to the SECONDO kernel, several commands are directly processed by the interface:

`HELP`	Displays a help message
`@FILE`	Read and execute commands from FILE
`@@FILE`	Read and execute commands from FILE, stop on errors
`@%FILE`	Read and execute commands from FILE, ignore pd-style[2]
`@&FILE`	Read and execute commands from FILE, ignore pd-style comments and stop on errors
`DEBUG 1`	debug mode (show annotated query and operator tree)
`DEBUG 2`	trace (show recursive calls)
`DEBUG 4`	trace nodes (construction of nodes of the operator tree, and execution of the query processor's Eval() method)
`DEBUG 8`	localInfo (prints a warning if an operator did not destroy its localinfo before the operator tree is deconstructed)
`DEBUG 16`	debug progress (after sending a REQUESTPROGRESS message to an operator, the ranges in the ProgressInfo are checked for whether they are reasonable. If not so, the according operator and ProgressInfo are reported)
`DEBUG 32`	trace progress (prints the result of each REQUESTPROGRESS message)
`DEBUG 64`	show type mappings
Hint:	The debug numbers can be added to activate multiple features

3.2.2 SecondoPL

The interface `SecondoPL`, which resides in the directory `Optimizer/` runs the SWI Prolog interpreter with SECONDO bindings. The SECONDO query optimizer is integrated seamlessly.

```
db@server:/home/db$ cd secondo/Optimizer
db@server:/home/db/secondo/Optimizer$ ./SecondoPL
[ ... some informational output ... ]

Type 'helpMe.' to get an overview on user level predicates.

?-
```

For input, the normal SWI prolog syntax applies; usually a predicate name followed by an optional list of arguments in brackets and terminated by a period. The following command opens the database *berlintest*, which was imported previously in Section 3.2.1.

```
?- open database berlintest.
```

The predicate (command) `showOptions` lists all optimizer-specific options. These can be modified with `setOption(X)` or `delOption(X)` and saved with `saveOptions`.

[2]PD (program with document) comments follow a certain format, which allows one to create nicely formatted PDF documentation directly from source files with *pdview* in the `Tools/pd/` directory.

```
?- showOptions.
Optimizer options (and sub-options):
 [x]    standard:          Adopt options for standard optimization process.
 [ ]    useCounters:       Insert counters into the computed plan.
 [ ]    noHashjoin:        Disables hashjoin.
 [ ]    noSymmjoin:        Disables symmjoin.
[ ... more options ... ]
?- setOption(useCounters).
Switched ON option: 'useCounters' - Insert counters into the computed plan.
true.

?- saveOptions.
true.

?-
```

The predicate `helpMe` lists all other SECONDO database commands available. To take advantage of the optimizer, SQL queries have to be formulated. The following example lists all cities with a population greater than 100000 people. The query is then optimized and sent to the SECONDO kernel in native format. The whole query must be formulated in lowercase characters.

```
?- sql select * from staedte where bev > 100000
Computing best Plan ...
Destination node 1 reached at iteration 1
Height of search tree for boundary is 0

The best plan is:

Staedte  feed filter[(.Bev > 100000)] {1}  consume

Estimated Cost: 0.38494000000000006

command
'query Staedte  feed filter[(.Bev > 100000)] {1}  consume'
started at: Tue Sep 11 11:54:42 2018
[ ... query result ... ]

true.
?-
```

Many other predicates are just wrappers around standard SECONDO commands (create, query, update, open to name just a few).

3.2.3 SecondoPLTTY

This interface is a combination of the `SecondoPL` and the `SecondoTTYBDB` interface. The same preprocessing semantics of `SecondoTTYBDB` apply: Commands are terminated with semicolon or two newlines and files can be loaded and executed with @FILE (see also Section 3.2.1). The terminating period for Prolog queries may be omitted here. Both SQL and executable style queries can be formulated here (lowercase and the leading `sql` are not mandatory any more):

```
SecondoPLTTY => select * from Staedte where Bev > 100000;

      SName : Aachen
        Bev : 239000
        PLZ : 5100
    Vorwahl : 0241
Kennzeichen : AC

...

SecondoPLTTY => query Staedte feed filter[.Bev > 100000] consume;

      SName : Aachen
        Bev : 239000
        PLZ : 5100
    Vorwahl : 0241
Kennzeichen : AC

...
```

3.2.4 NT-Versions

The single user interfaces also exist in NT (No Transactions) variants `SecondoTTYNT`, `SecondoPLNT` and `SecondoPLTTYNT`. The main difference is that the runtime flag `SMI:NoTransactions` is set, which disables the transaction submodule (see also Section 7.2). This means, performance is improved, but the database is prone to unrecoverable inconsistencies when crashes occur, since no transaction logs are written; hence, this mode should only be used for big imports, which would otherwise fail ("out of locks/lockers") or take too long to conclude, but not for regular production use.

3.3 Client Server Architecture

The Client Server model for SECONDO enables multiple users or applications to access databases concurrently by splitting into a server part and a user interface (client), which uses a TCP/IP connection to the server and hence does not need to be local anymore.

3.3.1 SecondoMonitor

The SECONDO listener, which in turn starts a SECONDO server on an incoming client connection, is started by the program `SecondoMonitor` in the `bin/` directory. Currently, it has to be started in the `bin/` directory itself, since it depends on two other programs being located in the current working directory.

After startup, a prompt waits for command input. Currently, the following commands are defined:

`HELP`	Displays the command help
`STARTUP`	Starts the Secondo Listener
`SHUTDOWN`	Stops a running Secondo Listener
`SHOW LOG`	Shows new logfile entries
`SHOW USERS`	Displays a list with connected users
`SHOW DATABASES`	Shows the currently opened databases
`SHOW LOCKS`	Shows a list of active database locks

The SECONDO listener is started with the `STARTUP` command. Alternatively, it is started automatically with the `SecondoMonitor` if the `-s` flag is specified. The listen port of the server can be configured with the option `SecondoPort` in the file `SecondoConfig.ini` (Default: 1234).

```
db@server:/home/db$
db@server:/home/db$ cd secondo/bin
db@server:/home/db/secondo/bin$ ./SecondoMonitor -s

*** Secondo Monitor ***

[ ... some informational output ... ]

Startup in progress ... Starting Process:
Program: SecondoListener
Args: "/home/db/secondo/bin/SecondoConfig.ini" 1234 /home/db/secondo-databases
completed.
SEC_MON> QUIT
SEC_MON> yes

Shutdown in progress ... completed.
Secondo Listener terminated with return code 0.

Terminating Secondo Monitor ...

Terminating Secondo Registrar ... completed.
Secondo Registrar terminated with return code 0.

Terminating Checkpoint Service ... completed.
Checkpoint service terminated with return code 0.
SecondoMonitor terminated.

db@server:/home/db/secondo/bin$
```

Several command line options exist:

`--help`	show the command line options
`-s` or `-startup`	Automatically start up the listener
`-V` or `-version`	Display version information and exit
`-c <configfile>`	Specifies the configuration file (overrides the default location)
`-d <directory>`	Use this database directory (overrides the configuration file)
`-p <port>`	Sets the listen port (overrides the configuration file)

3.3.2 SecondoTTYCS

The program `SecondoTTYCS` is the console client program, which connects to a running SECONDO server instance. The host and port of the server can be specified on the command line with the flags `-h` and `-p`; otherwise the defaults from the configuration file are used (`SecondoHost` and `SecondoPort`).

After successfully connecting to the server, this client behaves pretty much like the `SecondoTTYBDB` client described in Section 3.2.1. Some informational output from SecondoTTYBDB such as the running time of queries is omitted.[3]

```
db@server:/home/db$ SecondoTTYCS -h 127.0.0.1 -p 1234

*** Secondo TTY ***

[ ... some informational output ... ]

Connecting with Secondo server '127.0.0.1' on port 1234 ...
You are connected with a Secondo server.

Secondo TTY ready for operation.
Type 'HELP' to get a list of available commands.
Secondo =>
```

These command line options are available:

`--help`	show the command line options
`-c <configfile>`	Specifies the configuration file (overrides the default location)
`-h <host>`	Sets the server host (overrides the configuration file)
`-p <port>`	Sets the server port (overrides the configuration file)
`-u <user>`	Connect with the specified user name
`-s <secret>`	Authenticate using a given secret (password)

3.3.3 SecondoPLCS / SecondoPLTTYCS

The programs `SecondoPLCS` and `SecondoPLTTYCS` both behave analogously to `SecondoTTYCS`; they connect to a server instance and behave otherwise just like `SecondoPL` resp. `SecondoPLTTY` as already documented in Sections 3.2.2 and 3.2.3.

3.3.4 OptimizerServer

If the optimizer should be used together with the Javagui interface, the optimizer server has to be started. The startup script `StartOptServer` is located in the `Optimizer/` directory and also has to be executed from this directory. After starting, it connects to the SECONDO host and port specified in the configuration file `SecondoConfig.ini` and opens the port 1235 for incoming connections (an alternative port number can be specified at the command line).

[3] Nevertheless, the running time of queries can be determined in this interface or in the Javagui by a `query SEC2COMMANDS`, which lists previous queries with their running times.

```
db@server:/home/db$ cd secondo/Optimizer
db@server:/home/db/secondo/Optimizer$ ./StartOptServer 1235

java -Djava.library.path=. -cp ../Jpl/lib/classes:. OptimizerServer 1235

[ ... some informational output ... ]

waiting for requests

opt-server > quit
db@server:/home/db/secondo/Optimizer$
```

Valid commands are:

`quit`	Stops the server and exits
`clients`	Shows the number of connected clients
`trace-on`	Print debug messages about commands and optimizations
`trace-off`	Disable debug messages

3.3.5 Javagui

Aside from the command line interfaces there is a convenient graphical user interface, the Javagui, which is located in the directory `Javagui/`. The Javagui connects through a TCP/IP connection to a running SECONDO server (see Section 3.3.1). If the optimizer server is running, the Javagui automatically connects and makes use of it. The host and port of the server can be specified in the main configuration file `Javagui/gui.cfg`. The most interesting parameters are described in Section 6.3.

The Javagui itself is started with the script `sgui` and has to be executed in `Javagui/` as current working directory.

```
db@server:/home/db$ cd secondo/Javagui
db@server:/home/db/secondo/Javagui$ ./sgui
Info: start Javagui without any argument
Info: load configuration data from: /home/db/secondo/Javagui/gui.cfg
Info: set ServerName to 127.0.0.1
Info: set port to 1234
[ ... some more informational output ... ]
```

4 Querying Secondo

Query processing is the main task of SECONDO. As in other database management systems (DBMS), queries are written in a query language. In contrast to other systems, SECONDO offers two query languages: (1) an *executable language* and (2) an *SQL-like language*.

Executable language: The executable language is a low-level language. In this language, the data flow and the interaction between operators are described in detail. The description of the data flow is called the *query plan*. Specifying query plans directly is a unique feature of SECONDO. To the best of our knowledge, no other system allows the direct specification of query plans. In other DBMS query plans are generated automatically from another input language (e.g., an SQL query). The advantage of the executable language is that it is possible to specify exactly how a query should be processed and which operators should be used.

SQL-like language: The SQL-like language is a declarative language; only the result of the query is specified. The query optimizer of SECONDO generates the needed query plan (described in the executable language) which calculates the desired result. The SQL-like language has some advantages compared to the executable language. The language is easier to use for people that are familiar with SQL, the queries are shorter and the queries are optimized (the query optimizer generates a cost-optimized query plan).

See also the discussion of the two language levels in the introduction.

4.1 Executable Language

The executable language is used to specify query plans by describing an operator tree. Operator trees describe the way how the data are processed. No optimization of the query plan is performed, the operations are executed exactly in the specified way.

4.1.1 Stream Processing

A relation consists of tuples. When a relation is processed, the processing is done tuple by tuple. For example, the *filter* operator of the **RelationAlgebra** uses a stream of tuples as input and produces a stream of tuples as output. The *filter* operator lets only the tuples pass from the input stream to the output stream which match a certain criterion.

The *filter* operator cannot operate directly on a relation, the relation has to be converted into a stream of tuples first. This can be done by the *feed* operator.

In SECONDO, the result of a query has to be of a known data type and cannot be a stream. Several operators do exist to collect a stream and convert it into a certain data type. The operator *consume* takes the stream and creates a new relation (see Figure 5). The operator *count* instead takes the stream and produces an integer with the number of elements in the stream as the result.

4.1.2 Operator Tree

The main task of the executable language is to describe operator trees. These operator trees are executed by the query processor of SECONDO. In this section an example is presented

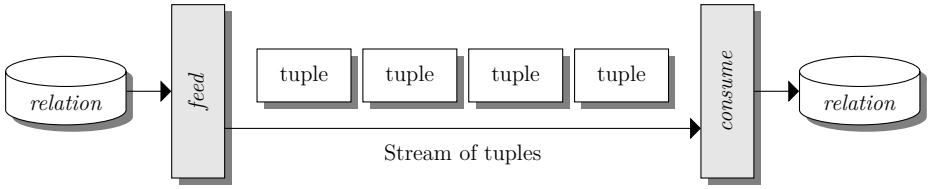

Figure 5: Converting a relation into a stream of tuples with the operator *feed* and converting the stream back into a relation with the operator *consume*.

to show how an operator tree can be described.

In this example an equi-join of the two tables *Orte* and *plz* should be computed. The table *Orte* contains towns with their names and additional information like the population. The table *plz* contains zip codes and town names. The join is executed to combine the data of both tables. To make the query a bit more complex, only towns with a zip code greater than 80000 should be included in the result. In the executable language the join can be described with the expression from Figure 6.

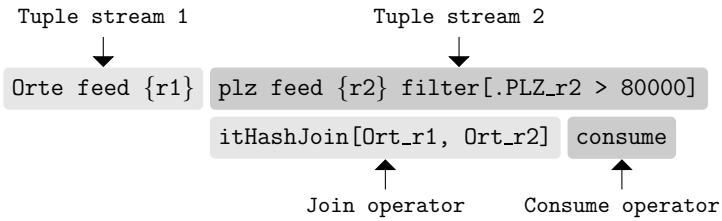

Figure 6: A join of two tables described in the executable language.

The operator tree in Figure 7 shows the operator tree for this calculation. Both relations are read by the *feed* operator. Each operator instance creates one tuple stream for its argument relation. The stream of the *plz* relation is filtered, the stream of the *Orte* relation is used directly. The *join* operator performs the join of both tuple streams and emits a stream of joined tuples. The *consume* operator creates a new relation from the stream.

4.1.3 Direct Execution of the Query Plan

For many tasks, SECONDO provides a wide range of operators. These operators implement different algorithms to achieve the same goal. Besides, different query plans can calculate the same result. Some of the plans are faster and some of the plans are slower. It is the responsibility of the user to specify an efficient query plan which calculates the result in a short time.

For example, an equi-join can be executed in SECONDO by the operators *sortmergejoin*, *mergejoin*, or *itHashJoin* (among others). The operator *sortmergejoin* takes up two tuple streams, sorts them and calculates the join result. The operator *mergejoin* instead assumes, that the input streams are already sorted. This operator avoids the expensive sorting step, but the operator produces the correct result only if both input streams are sorted. The operator *itHashJoin* creates a hash table from the first stream and matches the tuples of the second stream against the hash table to find join candidates. The efficiency of the different operators depends on factors such as the size of the input, the available memory for the operator and additional characteristics of the data (e.g., whether both input streams are sorted).

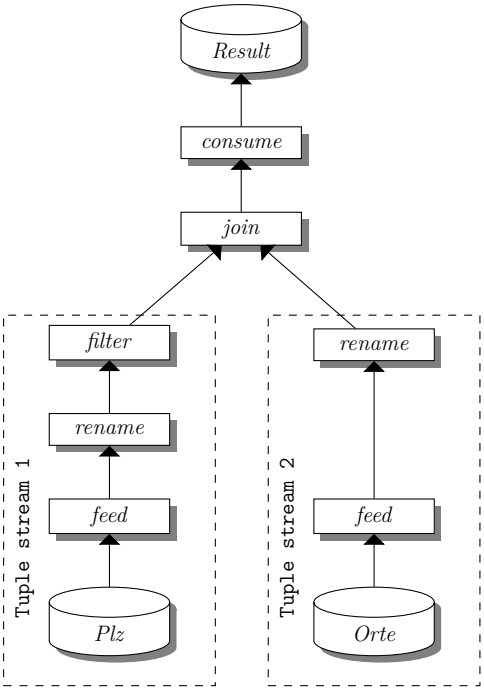

Figure 7: The operator tree for the join. The figure shows a simplified version. Some nodes (e.g., the function of the filter operator) have been omitted.

4.1.4 Constants

A constant value can be needed to formulate queries. For example, a *filter* operator can compare the values in an input stream with an integer. This operation was already shown in the operator tree in Figure 7. In the operator tree, the *filter* operator lets only pass the tuples which have a zip code greater than 80000. In SECONDO, constants have a type and a value and they can be defined with the following syntax:

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

For example, the following expressions define constants:

```
[const int value 5]
[const string value "secondo"]
[const bool value TRUE]
[const rectangle value (12.0 16.0 2.5 50.0)]
```

Please note, simple types such as integers, strings or booleans can be used directly as constants. The first three examples can be written as `5`, `"secondo"`, and `TRUE`.

4.1.5 Type Expressions

SECONDO supports a wide range of data types. As shown in the last section, a type expression has to be specified when a constant is defined. Besides the already shown simple types (e.g., integer or boolean) more complex types such as relations can be specified. The general syntax for creating a type is:

```
<type constructor>
```

or

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

Type expressions in SECONDO can be used to describe relations. The following term defines a relation of tuples. Each tuple has two attributes (`Name` and `Pop`):

```
rel(tuple([Name: string, Pop: int]))
```

A constant relation with three tuples can be defined as follows:

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

4.1.6 Value Expressions

SECONDO is a database management system which allows one to calculate very complex value expressions. Among others, the value expression can be an arithmetic expression or a complex join over several relations.

The algebra modules of SECONDO contain operators which can be used to perform complex calculations. Operators can have different numbers of parameters. Therefore, no general syntax for a value expression can be given. The syntax of the most important operators is discussed in Section 5.

An example for a more complex value expression is the following:

```
StaedteTest feed filter [.SName contains "burg"] project [SName, Kennzeichen]
  consume
```

The expression reads all tuples from the relation *StaedteTest* and removes all tuples whose name does not contain `"burg"`. Afterwards, the tuples are reduced to the attributes `SName` and `Kennzeichen` and a new relation is created as the result of the expression.

Renaming When multiple tables are queried, the attributes of the tables can be renamed to avoid name conflicts. In the following expression, the tables *Orte* and *plz* are used. Both tables contain an attribute with the name `Ort`. To avoid name conflicts, the syntax {`<suffix>`} can be used. It is a shortcut for the *rename* operator of the **RelationAlgebra**. It renames all attributes of a tuple to `<attr>_<suffix>`. In the expression, the attribute `Ort` of the relation *Orte* is renamed to `Ort_r1` and the attribute of the relation *plz* is renamed to `Ort_r2`.

```
query Orte feed {r1} plz feed {r2} itHashJoin[Ort_r1, Ort_r2] consume
```

4.1.7 Parameter Functions

A useful feature of SECONDO are anonymous parameter functions. For example, these functions are used when a stream of tuples is processed and a filter operator has to decide whether or not the tuple passes the filter. The filter operator calls the parameter function with each tuple and the function can return true or false, depending on whether or not the tuple fulfills the filter condition.

```
filter[fun (tuple1: TUPLE) attr(tuple1, No) > 5]
```

In the above expression, an anonymous parameter function is passed to the filter operator. The function takes one argument with the type TUPLE[4]. `fun` is the keyword for the function, `tuple1` is the name which is assigned to the current tuple of the stream. By using this name, the tuple can be accessed. On each call, the function extracts the attribute with the name `No` from the tuple and calls the `>` operator. True is returned when the `No` attribute of the tuple is larger than 5. In this case, the tuple passes the filter operator. Otherwise, the tuple is not forwarded to the result stream of the operator.

Short syntax The function in the example above is complex to write. SECONDO allows a shorter syntax for defining such functions. In the abbreviated form, the function can be shortened to:

```
filter[attr(., No) > 5]
```

In this short notation, no type and name for the parameter are specified. The notation is replaced by the notation above by the SECONDO parser. The name of the input variable is shortly referred to with the `.` symbol. In functions which accept two arguments (e.g., a function that joins two tuple streams), the second input is referred to with the `..` symbol.

Attribute access Accessing attributes is a very common functionality. The expression `attr(., attrname)` can be shortly written as `.attrname`. The same is true for the second parameter. The attributes of this parameter can be accessed by calling `..attrname`. By using this syntax, the anonymous parameter function from the example in this section can now be written as:

```
filter[.No > 5]
```

4.1.8 Operator Memory

A query can consist of several operators and each of these operators can require some memory for its work. SECONDO controls how much memory each operator may consume. This ensures that queries are executed without using more memory than available. Some operators can complete their work faster if they have more memory available.

An example is given in the following: the operator *itHashJoin* performs a hash based join. The operator builds a hash map from the first input stream and matches the tuples from the second input stream against this hash map. The hash map is stored in memory and has a limited size.

If the first tuple stream can not be inserted completely into the hash map, it is partitioned into chunks. The chunks have a size so that they can be loaded completely into the hash map. By using multiple chunks, the operator performs several iterations to perform the join. The second stream needs to be read for each chunk and compared against the hash maps. When the operator is allowed to use more memory, the hash map can become larger and the number of needed iterations is reduced.

The maximum memory usage per operator is controlled by the parameter `GlobalMemory` in the configuration file (see Section 7.2). This parameter describes how much main memory

[4]The TUPLE operator extracts automatically the type of the tuple from the input stream. Without this keyword, the complete function has to be written as: `filter[fun (tuple1: tuple([No: int])) attr(tuple1, No) > 5]`.

(in MB) all operators in a query can use together. The query processor splits this amount of memory equally among operators registered as benefiting from large memory. Other operators get a minimum amount of 16 MB.

However, SECONDO allows one to overwrite this parameter dynamically in the query. The keyword {`memory memory_in_MB`} after an operator allows one to determine the maximum amount of memory in MB the operator can consume. In this case, only the remaining memory is split equally among the remaining memory-using operators.

In the following query, the operator *itHashJoin* can consume up to 512 MB of memory:

```
Orte feed {r1} plz feed {r2} itHashJoin[Ort_r1, Ort_r2] {memory 512} consume
```

4.1.9 Commands

Basic commands In this section, the most basic commands are described. They can be used to create objects and to execute queries.

- `query <value expression>` - Evaluates the given value expression and displays the result object.
- `let <identifier> = <value expression>` - This command does almost the same as the `query` command. In contrast, the result of the `<value expression>` is not displayed on the screen. Instead, the result is stored in an object with the name `<identifier>`. The command only runs successfully if the object does not exist yet in the database; otherwise, an error message is displayed.
- `derive <identifier> = <value expression>` - This works basically in the same way as the `let` command. The difference is the handling of the created objects during creating and restoring a database dump. The `derive` command should be used for objects that have no external representation, e.g., indexes.
- `update <identifier> := <value expression>` - Assigns the result of the value expression to an existing object in the database.
- `delete <identifier>` - Deletes the object with the name `<identifier>` from the currently opened database.
- `type <identifier> = <type expression>` - Creates a named type in the database.
- `delete type <identifier>` - Deletes a named type from the database.
- `create <identifier> : <type expression>` - Creates an object of the given type with undefined value.
- `kill <identifier>` - Removes the object with the name `<identifier>` from the opened database catalog without removing its data structures. Generally, the `delete` command should be used to remove database objects. The `kill` command should only be used if the `delete` command crashes the database due to corrupted persistent data structures for this object.

The first four commands are the most important ones. The next three are rarely used, but available for special purposes. The last one is not needed in ordinary circumstances but useful to remove a corrupt object from a database e.g. in program development (debugging).

An `<identifier>` is defined by the following regular expression with a maximum length of 48 characters: `[a-z,A-Z]([a-z,A-Z]|[0-9]|_)*`. For example, `lineitem`, `employee`, `cities_pop` are valid identifiers, whereas `_x_` or `10times` are not.

Combined commands Besides the basic commands described in the last section, SECONDO offers the possibility to combine commands to more complex ones.

- `if <value expr> then <command$_1$> [ else <command$_2$> ] endif` - Runs the command `<command$_1$>` only if the condition `<value expr>` evaluates to `true`. If the result is `false`, nothing happens except the optional `else` part specifies `<command$_2$>` that is executed. If `<value expr>` does not represent a defined `bool` value, the command fails. Otherwise, the result of the executed command is forwarded as the result of the conditional command.
- `while <value expression> do <command> endwhile` - Executes `<command>` while the condition `<value expression>` evaluates to `true`.
- { `<command> [ | <command> ]`* } - Executes a sequence of commands enclosed in braces and separated by the pipe symbol (|). All commands are executed independently of the fail of a single command. The success of the whole command sequence corresponds to the success of the last command.
- {{ `<command> [ | <command> ]`* }} - Works very similarly to the previously described command sequence, except that the execution stops after the first failed command.

Databases The commands in this section allow the management of databases. With these commands databases can be created, opened, and deleted.

- `create database <databasename>` - Creates a new database with the name `<databasename>`.
- `open database <databasename>` - Opens the database with the name `<databasename>`.
- `close database` - Closes the currently open database.
- `delete database <databasename>` - Deletes the database with the name `<databasename>`. All databases need to be closed before a database can be deleted.

Transactions Transactions allow the user to control whether the result of one or more operations should be persisted or not. The commands for controlling transactions are discussed in this section.

- `begin transaction` - Starts a new transaction; all commands until the next commit or abort command are managed as one common unit of work.
- `commit transaction` - Commits a running transaction; all changes are persisted in the database.
- `abort transaction` - The current transaction is reverted and all changes are discarded.

Import and Export In this section the commands are discussed that are needed for importing or exporting data from and into SECONDO. The commands are useful for creating backups or to load an existing database. Further possibilities are presented in Section 5.

- `save database to <file>` - Write the currently opened database in the nested list format into the file `<file>`. If the file exists, it will be overwritten, otherwise it will be created.
- `restore database <identifier> from <file>` - Imports the contents of the file `<file>` into the database `<identifier>`. If the database already exists, it will be overwritten. If the database is not yet present, it will be created.
- `save <identifier> to <file>` - Writes the object `<identifier>` into the file `<file>`. If the file already exists, it will be overwritten.
- `restore <identifier> from <file>` - Creates a new object with the name `<identifier>`. If another object with the same name already exists, the command fails. Type and value of the object are read from file `<file>`.

Inquiries The commands in this section allow one to show information about the available databases, the existing types and objects and the operators and algebras that are known by SECONDO.

- `list databases` - Displays a list of names for all known databases.
- `list type constructors` - Displays all names of type constructors together with their specification and an example on the screen.
- `list operators` - Nearly the same as the command above, but information about the known operators is shown.
- `list algebras` - Displays a list containing all names of active algebra modules.
- `list algebra <identifier>` - Displays type constructors and operators of the specified algebra.
- `list types` - Displays a list of named types present in the currently opened database.
- `list objects` - Displays a list of objects present in the currently opened database.

Named types are hardly used in practice. The commands `list type constructors` and `list operators` have been useful in the early days of SECONDO but are nowadays not practical any more, because the number of available types and operators is overwhelming. At the time of writing, the number of operators in the author's system is 3419. To get information about types and operators, it is still practical to list one particular algebra or to use the queries described next.

4.1.10 Online Help

SECONDO provides some virtual system tables; these tables are automatically present in every database. The tables provide information about the known operators and statistical information about the performed queries. The name of all these special tables starts with *SEC2*. The most interesting tables for a user are *SEC2OPERATORINFO* and *SEC2TYPEINFO*.

SEC2OPERATORINFO The table *SEC2OPERATORINFO* contains information about all known operators. To see all known operators, their signature and an example, the following command can be used:

```
query SEC2OPERATORINFO
```

The content of the tables can be queried and filtered like a regular table. To show only the operators which work with a stream, the following command can be used:

```
query SEC2OPERATORINFO feed filter[.Signature contains "stream"] consume
```

To show more information about the operator *feed*, the following command can be used:

```
query SEC2OPERATORINFO feed filter[.Name = "feed"] consume
```

To simplify this task, a parameter function can be created. In the following example, a function with the name `showop` is defined. The function expects a string as parameter. The operator for which the information is shown is determined by this parameter.

```
let showop = fun(arg: string) SEC2OPERATORINFO feed
  filter[toLower(.Name) = toLower(arg)] consume
```

Now, the function can be used. In the following example, the function is used to show information about the *feed* operator:

```
query showop("feed")
```

SEC2TYPEINFO Information about the types that are known by SECONDO are contained in the table *SEC2TYPEINFO*. To show all known types, the following command can be used:

```
query SEC2TYPEINFO
```

The following command shows more information about the type `interval`:

```
query SEC2TYPEINFO feed filter [.Type = "interval"] consume
```

4.2 SQL-like Language

As a further query language, SECONDO provides an SQL-like language. This language is based on SQL and implements a subset of the SQL standard. The SQL-like language is processed by the optimizer which is written in Prolog. The optimizer generates a query plan in the executable language and the generated queries are passed to the SECONDO kernel for execution. During the generation of the query, the optimizer tries to find the cost-optimal query (i.e., the query that calculates the result in the fastest way).

In the following examples, we assume that a user interface such as SecondoPLTTY or the Javagui is used. These permit the so-called relaxed notation which is closer to the original SQL. In contrast, if the user interfaces SecondoPL or SecondoPLCS are used, some restrictions apply, because these are direct Prolog interfaces. The differences between the two notations are explained at the beginning of Section 5. These latter interfaces are mainly used for optimizer development.

4.2.1 General Information

The SQL-like language has the following syntax:

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

For example, the following query can be used to select all tuples from the table *Staedte* with the condition `Bev > 500000`[5].

```
select * from Staedte where Bev > 500000
```

After the query is executed, it takes some time and the following query plan in the executable language is calculated:

```
Computing best Plan ...
Destination node 1 reached at iteration 1
Height of search tree for boundary is 0

Optimized plan is:
query Staedte feed filter[(.Bev > 500000)]
   {0.20668965517241378, 0.6724137931034483} consume

Estimated Costs are:
0.41194000000000003
```

The output contains also the query plan in the executable language. The term {`0.206689 65517241378, 0.6724137931034483`} is an annotation about the *cost of the predicate* and the estimated *selectivity*. These values are used by the SECONDO kernel for progress estimation.

When multiple tables are queried, the attributes of the tables can be renamed to avoid name conflicts. In the following query, the tables *Orte* and *plz* are used. Both tables contain an attribute with the name `Ort`. To avoid name conflicts, the keyword `as <name>` is used. It renames all the attributes of the table to `<name>.<attr>`. In the following query, the attribute `Ort` of the relation *Orte* is renamed to `o.Ort` and the attribute `Ort` of the relation *plz* is renamed to `p.Ort`.

```
select *
from [Orte as o, plz as p]
where [o.Ort = p.Ort, o.Ort contains "dorf", (p.PLZ mod 13) = 0]
```

4.2.2 Syntax of the Language

Table 1 contains the syntax of the SQL-like language as a context free grammar. The table contains only the basic elements of the language. Advanced topics such as aggregation or ordering will be discussed later in this section. The update of relations is shown in Section 5.

We use the following conventions: words written in `typewriter font` are grammar symbols (non-terminals), words in **bold face** are terminal symbols. The symbols "`->`" and "`|`" are meta-symbols denoting derivation in the grammar and separation of alternatives. Other characters like "*" or ":" are also terminals. "id" is any valid SECONDO identifier. ϵ denotes the empty word.

The SQL-like language allows one to determine the ordering of the result. Besides, the number of tuples in the result can be restricted. It is possible to restrict to the first or last tuples in the result set or to take a random sample (using **some**). The additional grammar of the language is shown in Table 2.

For example, the following query can be formulated with our SQL-like language:

[5]It is assumed that the database opt is opened already. Otherwise, the database has to be opened first by executing `open database opt`

query	->	**select** distinct-clause sel-clause **from** rel-clause where-clause
distinct-clause	->	**all** | **distinct** | ϵ
sel-clause	->	* | result | [result-list] | **count**(distinct-clause *)
result	->	attr | attr-expr **as** newname
result-list	->	result | result, result-list
attr	->	attrname | var:attrname
attr-list	->	attr | attr, attr-list
attrname	->	**id**
newname	->	**id**
rel	->	relname | relname **as** var
rel-clause	->	[rel-list]
rel-list	->	rel | rel, rel-list
relname	->	**id**
var	->	**id**
where-clause	->	**where** [pred-list] | **where** pred | ϵ
pred	->	attr-boolexpr
pred-list	->	pred | pred, pred-list

Table 1: The main grammar of the SQL-like language.

query	->	**select** distinct-clause sel-clause **from** rel-clause where-clause orderby-clause limit-clause
orderby-clause	->	**orderby** [orderattr-list] | **orderby** orderattr | ϵ
orderattr	->	attrname | attrname **asc** | attrname **desc**
orderattr-list	->	orderattr | orderattr, orderattr-list
limit-clause	->	**first int-constant** | **last int-constant** | **some int-constant** | ϵ

Table 2: The ordering and limiting part of the SQL-like language.

```
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]
first 10
```

As in SQL, aggregations or grouping can also be applied in our SQL-like language. The corresponding grammar is shown in Table 3.

query	->	**select** aggr-clause **from** rel-clause where-clause groupby-clause orderby-clause first-clause
aggr-clause	->	aggr2 | [aggr2, aggr-list]
aggr2	->	**count**(distinct-clause *) **as** newname | **aggrop**(ext-attr-expr) **as** newname | arbitrary-aggr **as** newname
aggr	->	groupattr | groupattr **as** newname | aggr2
aggr-list	->	aggr | aggr, aggr-list
aggrop	->	**min** | **max** | **sum** | **avg** | **extract** | **count**
aggr-fun	->	(*) | (+) | **union_new** | **intersection_new** | any name fun of a binary SECONDO-operator or function object with syntax fun: T x T -> T which should be associative and commutative. Infix-operators must be enclosed in round parentheses.
arbitrary-aggr	->	**aggregate**(ext-attr-expr, aggrfun, datatype, datatype-constant)
datatype	->	any name of a SECONDO datatype, e.g., int | real | bool
groupby-clause	->	**groupby** groupattrs | **groupby** groupattrs **having** preds
groupattrs	->	groupattr | [groupattr-list]
groupattr-list	->	groupattr | groupattr, groupattr-list
groupattr	->	attr
preds	->	pred | [pred-list]

Table 3: The grouping and aggregation part of the SQL-like language.

For example, a query with grouping and aggregation looks like:

```
select [Ort, min(PLZ) as Minplz, max(PLZ) as Maxplz, count(*) as Cntplz]
from plz where PLZ > 40000
groupby Ort
having Cntplz > 100
orderby Cntplz desc
```

The SQL-like language of SECONDO also supports **union**, **intersection**, and **minus** operations. They can be used as binary operations or be applied to lists of simple queries. Of course, all relations resulting from simple queries must have the same schema. The grammar for using such operations is shown in Table 4.

query	->	query **union** query | query **intersection** query | query **minus** query
mquery	->	**union**[query-list] | **intersection** [query-list]
query-list	->	query | query, query-list

Table 4: The set operation part of the SQL-like language.

This part of the language allows queries such as the following:

```
select * from plz
minus
select [PLZ, Ort] from [Orte as o, plz] where o.Ort = Ort

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

4.2.3 Updating the Optimizer's Knowledge

When a database is opened for the first time by an interface involving the optimizer (e.g. `SecondoPL`, `SecondoPLTTY`, `Javagui` with optimizer server), the optimizer creates a catalog on relations and indexes among others. It contains the existing indices, samples, and selectivities of predicates. When a database is reopened, the optimizer compares the database catalog with its own info and makes its info consistent when necessary.

When database objects are created or deleted through an interface involving the optimizer, the optimizer info of this SECONDO instance is automatically kept up to date.

However, in a client-server configuration it may be necessary to explicitly update the optimizer of one client when another client creates or deletes database objects.

Also, if the stored data values have been significantly changed so that samples are not accurate any more, the optimizer must be informed to adapt to these changes. This is the purpose of the following predicates.

Updating one relation The following command causes the optimizer to delete all information it has about the relation *Rel*, including selectivities of predicates. An existing sample of the data is also destroyed. A query afterwards involving this relation collects all information from scratch. Existing or non-existing indexes are also discovered.

```
updateRel(Rel)
```

Updating the knowledge for database With the following command, the optimizer's knowledge about a complete database is updated. In particular, it will check whether any relations and/or indexes have been added or removed and update its knowledge base accordingly. Hence this can be used after creating or destroying an index by a third party, without losing all the other information.

```
updateCatalog
```

Complete knowledge update The following command will retract all metadata on objects of a database. When the database is opened for the next time, the optimizer will need to recollect metadata. Before the command can be executed, the database needs to be closed first.

```
updateDB(DB)
```

4.2.4 Optimizer Options

The optimizer is configurable, features can be enabled or disabled at runtime. When the optimizer starts up, a list of all settings is displayed. This list can be re-displayed by using the command:

```
showOptions
```

To activate a certain feature, the following command can be used:

```
setOption(OptionName)
```

To disable a certain feature, the following command can be used:

```
delOption(OptionName)
```

When the `autosave` feature is active, the option settings will be saved and restored on the next start of the optimizer.

To display the online help of the optimizer, the following command can be called:

```
helpMe
```

5 Examples

In this section, we will show some examples how to use SECONDO. Even though SECONDO supports different data models, we will limit this section to the well-known relational data model. As described in Section 4, SECONDO provides two levels of querying, the executable language and the SQL-like language. In this section, we will describe both variants.

In the case that both variants should be tested, a user interface supporting the executable language and the SQL-like language must be started. The attentive reader knows that `SecondoPLTTY` and the Javagui are possible options. If only one of the variants is in the focus, one of the appropriate interfaces must be started (see Section 3).

As mentioned earlier, the SQL notation used differs slightly depending on the user interface. In SecondoPL or SecondoPLCS which are direct interfaces to a Prolog interpreter, some restrictions apply:

- Relation and attribute names must be written in lower case.
- A period cannot be used within a qualification (such as `a.name`); instead a colon has to be used (`a:name`).
- An SQL statement or query needs to be prefixed with `sql` and terminated with a period.
- Only some kernel commands such as `open database <db>`, `delete <object>` are available directly; others have to be embedded into a `secondo('...')` command.

We call this notation the *restricted* form. These user interfaces are more useful for optimizer development.

In contrast, in the Javagui or in `SecondoPLTTY` interfaces, a preprocessor transforms queries into the form shown above. Hence the following is allowed:

- Relation and attribute names can be written in their normal notation as in executable language.
- A period may be used within a qualification; hence writing `a.Name` is fine.
- An SQL statement or a query needs no prefix and can be terminated as in executable language.
- All general commands are available.

We call this the *relaxed* form or notation. In the sequel we will show the relaxed form and provide a few additional examples of queries in the restricted form.

5.1 Preparations

Before objects can be created, deleted, or manipulated, a database must be opened. For the next examples, we use a database called `exampleDB`. The following commands create and open this database.

```
create database exampleDB;
open database exampleDB;
```

In restricted form (`SecondoPL`) these commands are written as

```
create database exampleDB.
open database exampleDB.
```

5.2 Creating an Empty Relation

5.2.1 Executable Language

In the executable language, there is a `const` construct allowing one to describe a value of an object of an arbitrary supported type. For creating an empty relation, we have to enter the type (schema) of this relation as well as its value. The value of an empty relation is just an empty list written as a pair of brackets.

```
let myfirstrel = [ const rel(tuple([Name : string, Age : int])) value () ];
```

Here, a new relation named *myfirstrel* having two columns `Name` and `Age` with the given types is created. Up to now, this relation is empty, i.e., it contains no tuple. Note that all attribute names have to start with an upper case.

The presence of this relation can be checked using the command:

```
list objects;
```

5.2.2 SQL-like Language

A relation with the same schema can be created in the SQL-like language using the command:

```
create table myfirstrel columns [ Name : string, Age : int ];
```

Restricted form:

```
sql create table myfirstrel columns [ name : string, age : int ].
```

The success of the creation of this table can be checked using the command:

```
showDatabaseSchema
```

5.3 Inserting Tuples into a Relation

5.3.1 Executable Language

Inserting Single Tuples

SECONDO's `update` command just replaces the whole object stored in the catalog by another value. Thus, inserting tuples into a relation is realized as a side effect of a query. For the manipulation of relations, operators of the **UpdateRelationAlgebra** are used. We insert two new tuples into the relation *myfirstrel*:

```
query myfirstrel inserttuple["Anna", 27] count;
query myfirstrel inserttuple["Hans", 42] count;
```

The *inserttuple* operator gets as input the relation to be updated and a list of attribute values that have to fit the relation's schema. Note that SECONDO does not support default values. The operator produces a stream of tuples consisting of the freshly inserted tuple extended by a tuple id. This is useful for updating indexes on this relation as described in Section 5.10. Here, we just count the inserted tuples. The result will always be 1.

Inserting A Stream of Tuples

A stream of tuples can be inserted into a relation provided the tuples in the stream and in the relation have the same schema. This is done using the *insert* operator.

For example, suppose we have a relation *mysecondrel* with the same schema (type) as *myfirstrel*. Then we can insert all tuples from *myfirstrel* having `Age < 40` into *mysecondrel* as follows:

```
query myfirstel feed filter[.Age < 40] mysecondrel insert count;
```

The *insert* operator gets a stream of tuples and a relation of the same tuple type. It returns a stream of inserted tuples extended by tuple id as motivated above.

We can check the success of the insertions by printing the relations:

```
query myfirstrel;
query mysecondrel;
```

5.3.2 SQL-like Language

Inserting Single Tuples

Using the SQL-like language, the tuples can be inserted by:

```
insert into myfirstrel values ["Anna", 27];
insert into myfirstrel values ["Hans", 42];
```

Inserting A Set of Tuples

```
insert into mysecondrel
  select * from myfirstrel where Age < 40;
```

We can check the success by displaying the relations:

```
select * from myfirstrel;
select * from mysecondrel;
```

5.4 Removing Tuples from a Relation

5.4.1 Executable Language

If tuples in a relation become obsolete, these tuples should be removed from this relation. The **UpdateRelationAlgebra** provides some operators for this purpose. In the example, we will use the *deletedirect* operator. This operator uses a tuple stream and removes all tuples from the relation that have the same tuple ids as the incoming tuples. The result is a stream of the removed tuples extended by the tuple id. Because up to now there is no index created over this relation, we just count the removed tuples.

```
query myfirstrel feed filter[.Name = "Hans"] myfirstrel deletedirect count;
```

5.4.2 SQL-like Language

In the SQL-like language tuples can be deleted from a relation by the `delete` command.

```
delete from myfirstrel where [Name = "Hans"];
```

5.5 Changing Tuples in a Relation

5.5.1 Executable Language

For the manipulation of tuples, the **UpdateRelationAlgebra** provides some operators. In the example, the *updatedirect* operator is used.

```
query myfirstrel feed filter[.Name = "Anna"]
  myfirstrel updatedirect[Age : .Age + 1]
  count;
```

This command works as described in the following. In the first part, the tuples for the update are selected. This stream together with the relation to update itself are the first two arguments of the *updatedirect* operator. The third argument is a list of functions that describe how to update a tuple from the stream. Before the colon, the name of the attribute is given that should be changed. After the colon, the description of the computation of the new value follows. For accessing the present value in the tuple, the dot notation is used, e.g., here `.Age` corresponds to the present value of the `Age` attribute.

5.5.2 SQL-like Language

In the SQL-like language, an update command is used:

```
update myfirstrel set [Age = Age + 1] where [Name = "Anna"];
```

Restricted form for `SecondoPL`:

```
sql update myfirstrel set [age = age + 1] where [name = "Anna"].
```

5.6 Importing Data from Files

In some cases, data are already present in files in a certain format. For several of these formats SECONDO provides import operators that can be used to store the contents of the file into a relation.

This functionality is only available in the executable language. In a pure optimizer environment (`SecondoPL`, `SecondoPLCS`), it is possible to execute such commands by embedding them into the `secondo` predicate:

```
secondo('<command in executable language>').
```

If single quotes are required in such an executable command, they must be masked with a backslash.

5.6.1 Comma Separated Values

A frequently used exchange format for the representation of tables in files are comma separated values (csv). Here, each tuple is provided within a single line where the attributes are separated by some character (mostly a comma). The operator used here is *csvimport*. This operator needs at least four arguments. The first argument is a relation providing the schema of the table stored in the file. The content of this relation is untouched. The second argument is the name of the file containing the csv data. Some files provide additional information in the first lines. Such lines should be ignored during the import. The third argument of *csvimport* corresponds to the number of such header lines. If no such lines are present, just use 0 here. On the other hand, some lines in the file may be commented out. This is done using a special character at the begin of a line, given as the next argument to the operator. If no comments are present in the file, an empty

string is used here. The operator has further optional arguments. For example, it is possible to change the separator character to another than comma. Please read the online documentation of this operator for more information. The operator produces a stream of tuples that can be collected by the *consume* operator.

The `bin` directory of SECONDO contains a file named `Trk110731.csv` that is used here to demonstrate the import of csv files. It contains a recorded track of a vehicle. See the first lines in the file for further information. The file can be imported by the following command:

```
let Trk110731  = [const rel(tuple([
  Lat : real,
  Long : real,
  UTC : string,
  Alt : int,
  Dist : real,
  Speed : int,
  Date : string,
  Name : int,
  Sat : int]))
value ()]
csvimport['Trk110731.csv', 7, "#"] consume;
```

5.6.2 DBase Files

DBase is one of the first database management systems for home computers. It uses simple files for storing data. This file format is used frequently as an interchange format. SECONDO supports importing DBase files in version 3 by using the *dbimport2* operator. The only argument of this operator is the file name. It returns a stream of the tuples stored in this file. SECONDO's `bin` directory contains a file `plz.db3`. This file can be imported by:

```
let plz = dbimport2('plz.db3') consume;
```

5.6.3 Shape Files

SECONDO supports spatial data types like point, multipoint, line, and region within the **SpatialAlgebra**. A common interchange format for this kind of data are ESRI Shapefiles. A table in this format consists of three parts. One part is a file with extension `.shp` that contains the geometries. A second file with extension `.shx` provides an index over this file. The last part is a DBase-III file providing additional attributes of the tuples. The geometries are connected to the DBase tuples by their position in the file, e.g, the third geometry in the `shp` file is connected to the third tuple in the `db3` file.

SECONDO provides the operator *importshp2* for importing the geometries of a shape file. This operator provides a stream of attributes that can be converted into a tuple stream using the *namedtransformstream* operator. The DBase file can be imported using the *dbimport2* operator known from the last section. The two tuple streams can be combined by the operator *obojoin* joining two tuple streams one by one. The index file is ignored by SECONDO. Within the directory `bin/testData` some sample files are located. Here, we will import street data from Berlin.

```
let streets = dbimport2('testData/streets.db3')
  shpimport2('testData/streets.shp') namedtransformstream[Geometry]
  obojoin
  consume;
```

5.6.4 Other File Formats

Besides the aforementioned formats, SECONDO supports also the import of other formats. Examples include OpenStreetMap data, nmea[6] data, gpx[7] files, ais[8] data, and some others. For further information see the online help of the appropriate operators. Note that some of the import operators are only available if some additional algebras have been included (see Section 7.1). For example the import of OpenStreetMap data depends on the activation of the **OSMAlgebra**.

5.7 Finding Data

5.7.1 Executable Language

Once the data are stored in tables, SECONDO provides mechanisms to find data with some specified properties. The simplest mechanism is a full table scan followed by a selection. Assume we have imported the `street` relation using the command above. If we want to find those streets having a name starting with an *A*, we can use the following command:

```
query streets feed filter[.Name starts "A"] project[Name] consume;
```

The *feed* operator puts the tuples of the relation into a tuple stream. The *filter* operator removes such tuples not fulfilling the given condition from this stream. Here, the condition is fulfilled if the value of the attribute `Name` of the current tuple starts with an *A*. The *project* operator restricts each tuple of the incoming stream to its `Name` attribute. The operator *consume* collects the tuples into the result relation.

5.7.2 SQL-like Language

In the SQL-like language the same search can be done using:

```
select Name from streets where [Name starts "A"]
```

5.8 Creating Indexes

A full table scan is a simple mechanism for finding data that works for each search condition. However, scanning all tuples of a relation is very expensive. SECONDO provides a set of index structures including B-trees, R-trees, M-trees, and hash tables to accelerate the search of data with given properties. Further index structures are available but beyond the scope of this introductory manual.

Note that for creating large indexes it is preferable to use a `SecondoTTYNT` interface, i.e., to switch off transactions. Otherwise logging all changes to the index structure leads to considerable overhead and slow execution.

5.8.1 Executable Language

It is possible to create several indexes over a single relation. Here, we build a B-tree and a hash index over the `PLZ` attribute of the `plz` relation and an M-tree over the `Ort` attribute of this relation. For indexing geo data, the R-tree is a commonly used index structure. We create such an index

[6] National Marine Electronics Association format
[7] GPS Exchange format
[8] Automatic Identification System

over the geometries stored in the streets relation. For creating an R-tree, also a bulkload variant is possible. This is faster for bigger relations. Since index structures have no external representation, for the creation of indexes, the *derive* command should be used to be able to save and restore the whole database.

Note that the naming convention used in these examples, namely `<relation>_<attribute>_<indextype>` needs to be followed to let the index be recognized by the optimizer. Indexes created in SQL automatically follow this convention.

```
derive plz_PLZ_btree = plz createbtree[PLZ];
derive plz_PLZ_hash = plz createhash[PLZ];
derive plz_Ort_mtree = plz createmtree[Ort];
derive streets_Geometry_rtree = streets creatertree[Geometry];
```

Alternatively we can create an R-tree index by a bulkload technique:

```
derive streets_Geometry_rtree =
  streets feed addid sortby[Geometry] bulkloadrtree[Geometry];
```

The bulkload technique first sorts a stream of tuples by the spatial attribute to be indexed. What does sorting by a spatial attribute mean? First of all, spatial values are reduced to a point, the center point of the bounding box. These points are then sorted into *z-order*. This is a space-filling curve that maps a 2d or higher-dimensional space into a 1d linear order, preserving proximity [Ore86]. Geometries from this z-ordered stream of tuples are then packed sequentially into pages of an R-tree, building the R-tree bottom-up. Because only complete pages are written, this is much faster than a sequence of random insertions into an R-tree.

There is one issue that needs to be observed: Sorting into z-order is based on the x- and y-coordinates of points. From these coordinates, *only the integer part is used*. This is no problem for the *streets* relation used above, as it has coordinates in meters. However, when geographic coordinates are used, the integer part just describes degrees of latitude or longitude and such coordinates are by far not precise enough for indexing.

The solution is to just scale up geographic coordinates by a large factor so that the integer part contains all useful information. For a rectangle value, this can be done by the *scalerect* operator which multiplies all coordinates by a given factor in x and y.

Suppose we have a relation *Buildings* in geographic coordinates. We build an R-tree index by bulkload as follows:

```
let Buildings_GeoData_rtree = Buildings feed addid
  extend[Box: scalerect(bbox(.GeoData), 1000000.0, 1000000.0)]
  sortby[Box] remove[Box] bulkloadrtree[GeoData]
```

Since the additional attribute `Box` is needed only for the sorting step, it can be removed directly after this step. It also needs to be removed to let the R-tree have the correct tuple type.

5.8.2 SQL-like Language

In the SQL-like language, these indexes can be created by the commands:

```
create index on plz columns PLZ;
create index on plz columns PLZ indextype hash;
create index on plz columns Ort indextype mtree;
create index on streets columns Geometry indextype rtree;
```

Restricted form for `SecondoPL`:

```
sql create index on plz columns plz.
sql create index on plz columns plz indextype hash.
sql create index on plz columns ort indextype mtree.
sql create index on streets columns geometry indextype rtree.
```

Unfortunately, the bulkload techniques are not yet available at the SQL level.

5.9 Using Indexes

If an index over an attribute of a relation is present, it can be used to accelerate selections on this relation.

5.9.1 Executable Language

Different index structures support different kinds of selections. For example, a hash structure supports only a selection by equality while a B-tree index additionally supports range queries.

The next query returns the city names having the zip code 58085.

```
query plz_PLZ_hash plz exactmatch[58085] project[Ort] consume;
```

Here, the *exactmatch* operator gets three arguments, the index, the indexed relation, and the value to search for. It returns the tuples having exactly the given value within a stream. The operator *project* reduces the tuples in this stream to the attribute `Ort`. Finally, the *consume* operator collects this tuple stream into a relation.

The same query also works with a B-tree:

```
query plz_PLZ_btree plz exactmatch[58085] project[Ort] consume;
```

For finding the city names in a range of zip codes, the hash index is not usable. The B-tree index supports this kind of query:

```
query plz_PLZ_btree plz range[58000, 59000] project[Ort] sort rdup consume;
```

The *range* operator uses the index, the relation, and an interval of attributes as input and returns such tuples where the indexed value is enclosed in the given interval. Because some cities have more than one zip code, the stream will have duplicates in the city names. The operator *rdup* removes such duplicates. Since this operator requires a sorted stream, we sort the stream using the operator *sort*.

The M-tree supports also range queries. In contrast to the previous query, the range is not defined as an interval but by a value and a maximum distance to this value. Since the M-tree is built over an attribute of type string, the well-known edit distance is used here. The next query returns the names of the cities having a maximum edit distance of 1 to Hagen. As before, we remove duplicates in the result.

```
query plz_Ort_mtree plz rangesearch["Hagen", 1.0]
  project[Ort] sort rdup consume;
```

Besides range queries, an M-tree supports also nearest neighbor queries.

```
query plz_Ort_mtree plz nnsearch["Hagen", 5] consume;
```

The result consists of 5 tuples, each having the value Hagen in its `Ort` attribute.

Geometric range queries are supported by the R-tree created in the last section. The range is described as a rectangle. In contrast to the previously used indexes, the index does not return

exact results, but such tuples whose bounding box[9] of its geometric attribute intersects the given rectangle. Hence the created tuple stream contains candidates of the result and must be filtered to get only those tuples having an intersection in the real geometry. The following query counts how many streets intersect some rectangle.

```
let box1 = [const rect value (4751.26 15618.2 6537.32 16482.1)];
query streets_Geometry_rtree streets windowintersects[box1]
  filter[.Geometry intersects1 box1 rect2region] count;
```

5.9.2 SQL-like Language

In the SQL-like language the optimizer decides on the usage of indexes. Of course, if no index exists, a full table scan is the only option. In the presence of one or more indexes, the cheapest plan is used.

Again, we want to find the city name having 58085 as its zip code.

```
select Ort from plz where  PLZ = 58085
```

If looking at the output, one can see the usage of the B-tree for this query.

The zip code range query above can be formulated as:

```
select distinct Ort from plz where between(PLZ, 58000, 59000)
```

By reading the output we will see that the B-tree index is not used here. Instead, a full table scan is preferred.

Spatial range queries can also be performed:

```
secondo('let reg1 = [const rect value (4751.26 15618.2 6537.32 16482.1)]
  rect2region');
select count(*) from streets where Geometry intersects reg1
```

As displayed, the R-tree has been applied.

5.10 Updating Relations with Indexes

If the content of a relation changes, also the indexes related to it must be updated. When using the optimizer, updates are forwarded automatically to all existing indexes. This means updates are entered in the same way independently whether indexes exist or not. Hence, only the executable language is described in this section.

When inserting a new tuple into a relation using the *inserttuple* operator, the result of this operator is a tuple stream containing the inserted tuple extended by a tuple id. This tuple stream can be used to inform indexes about this insertion.

The following command inserts a fictious city together with its zip code into the plz relation and updates the indexes created before:

```
query plz inserttuple[1111, "Gotham City"]
  plz_PLZ_btree insertbtree[PLZ]
  plz_PLZ_hash inserthash[PLZ] count;
```

Unfortunately, the M-tree does not support any updates, thus this index is omitted in the update. After the above update, the new tuple can be found using the B-tree index and the hash index.

[9] The minimum axis-parallel rectangle that encloses the whole geometry

```
query plz_PLZ_btree plz exactmatch[1111] consume;
query plz_PLZ_hash plz exactmatch[1111] consume;
```

When removing a tuple, the indexes must be changed, too:

```
query plz feed filter[.Ort contains "Gotham"] plz deletedirect
  plz_PLZ_btree deletebtree[PLZ]
  plz_PLZ_hash deletehash[PLZ]
  count;
```

Very similarly, updates on existing tuples can be forwarded to indexes using the operators *updatebtree* and *updatehash*. This task is left as an exercise to the reader.

5.11 Sorting

5.11.1 Executable Language

If result tuples are expected in a special order or if operators require a sorted tuple stream, unsorted streams must be sorted. In SECONDO there are two sorting possibilities, sorting by the whole tuple and sorting by a selection of attributes. To sort a tuple stream, the operator *sort*[10] is used. This operator sorts firstly by the first attribute in the tuples; if the first attribute is equal the second attribute is used as the next sorting criterion and so on.

The next command returns the zip codes and city names sorted by the whole tuple:

```
query plz feed sort consume;
```

For different priorities of the attributes or if a descending order is required, the operator *sortby* or the operator *sortbyh* can be used. The following command sorts the *plz* relation firstly by the city name and secondly descending by the zip code:

```
query plz feed sortbyh[Ort, PLZ desc] consume;
```

5.11.2 SQL-like Language

In SQL sorting of the result can be realized by using the `orderby` clause. Internal sorting, for example when using operators requiring a special order is automatically embedded into the plan by the optimizer.

The following command returns the *plz* relation sorted by `plz` and `ort`:

```
select * from plz orderby [PLZ, Ort]
```

The ordering can also be inverted for an attribute:

```
select * from plz orderby [Ort, PLZ desc]
```

5.12 Aggregations

5.12.1 Executable Language

For some evaluations, aggregations on relations are needed. SECONDO provides some standard aggregation functions like *min*, *max*, *sum*, and *avg*. While *min* and *max* can be applied to any

[10] or an alternative implementation *sorth* that does not support progress estimation but is a bit faster

attribute, the operators *sum* and *avg* are only applicable to numeric values (int, real). Additionally to these standard aggregations, SECONDO has also general aggregation operators, namely *aggregate* and *aggregateB*. Here, the aggregation function is freely choosable.

The following query returns the city name having the lexicographically largest value:

```
query plz feed max[Ort];
```

The following query sums up all zip codes of Hagen:

```
query plz feed filter[.Ort = "Hagen"] sum[PLZ];
```

Using the following command, the average age of our firstly created relation can be computed:

```
query myfirstrel feed avg[Age];
```

With the general version of the aggregation, we can for example build the union of all water areas containing `"see"` in their names:

```
query WFlaechen feed filter[.Name contains "see"]
  aggregateB[GeoData; fun(r1:region, r2:region) r1 union1 r2
  ; [const region value ()] ];
```

The operator *aggregateB* gets a tuple stream, the name of the attribute that should be aggregated, an aggregation function, and a value that is returned in case of an empty stream.

5.12.2 SQL-like Language

In the SQL-like language aggregations are possible only in combination with grouping described in the next section.

5.13 Grouping

Aggregations on whole relations are unusual. Mostly aggregations are computed for some groups (tuples having the same value in certain attributes) of a relation.

5.13.1 Executable Language

In the executable language, the *groupby* operator can be used for grouping. The arguments of this operator are a tuple stream ordered by the grouping attributes, the names of the grouping attributes, and a set of functions that are applied to each group. The result of each function must be an attribute data type. The result of *groupby* is a tuple stream consisting of the grouping attributes and the computed function values.

The next command determines the number of zip codes for each city name in the plz relation and sorts the result by this number.

```
query plz feed sortby[Ort] groupby[Ort; Cnt : group count] sortby[Cnt]
  consume;
```

5.13.2 SQL-like Language

Of course, grouping is also available in the SQL-like language.

```
select [Ort, count(*) as Cnt] from plz groupby Ort orderby Cnt
```

5.14 Combining Several Relations (Joins)

In this section, we discuss join operations. Note that set operations for combining relations with the same schema are also available as described in Section 4.2.

For the following examples, we will use the database `berlintest` containing more relations than the example database created before. If the `berlintest` database is already present, use the commands:

```
close database;
open database berlintest;
```

If the database `berlintest` does not exist, use the following commands for creating it:

```
close database;
restore database berlintest from berlintest;
```

In the Javagui or in `SecondoPLTTY`, the path to the `berlintest` database needs to be specified:

```
restore database berlintest from '../bin/berlintest';
```

5.14.1 Executable Language

For combining relations, SECONDO provides a set of join operators. The most general join operation in SECONDO is the operator *symmjoin*. Here, the join condition has no limitations. Note that the run time of this operator is proportional to the product of the stream cardinalities.

The following command retrieves city names that are properly contained in another city name. To reduce the complexity of this query, we restrict the city names to those starting with an *A*.

```
query plz feed filter[.Ort starts "A"]
  plz feed filter[.Ort starts "A"] {a}
  symmjoin[ (tolower(.Ort) contains tolower(..Ort_a))
    and (not( .Ort = ..Ort_a)) ]
  consume;
```

Note that join operators can be used only on tuple streams, whose attribute names are disjoint. For renaming the attributes, SECONDO provides a *rename* operator that is written as `{suffix}` behind the tuple stream. Each attribute name will be extended by an underline followed by the given suffix.

For less complex join conditions, indexes can be exploited to accelerate the computation of the result. This can be done by using the *loopjoin* operator. This operator gets a stream of tuples as its first argument. The second argument to this operator is a function that maps a single tuple from the stream into a stream of other tuples. It connects a tuple from the first input stream with all tuples created by the function. In the following example, the relation *Orte* (containing city information) is used. This table has the following schema:

```
Orte(Kennzeichen : string, Ort : string, Vorwahl : string, BevT : int)
```

The first attribute describes the licence plate code for the city, the second attribute is the name of the city followed by its STD code, and the population in units of thousand. The following command extends this relation by the zip code coming from the plz relation:

```
query Orte feed
  loopjoin[plz_Ort_btree plz exactmatch[.Ort] project[PLZ]]
  consume;
```

Index structures are only available on relations that are stored in the database. For derived streams, SECONDO provides further join mechanisms depending on the join conditions. For equality conditions, the operator *hashjoin* or the alternative implementation *itHashJoin* can be used. The next example computes the same result (disregarding the order) as before without using an existing index:

```
query Orte feed plz feed {a} itHashJoin[Ort, Ort_a]
  remove[Ort_a] renameAttr[PLZ : PLZ_a]
  consume;
```

The operator *itHashJoin* combines all tuples from both streams that have the same value for the specified attributes. In the result stream both attributes exist, so we remove one of them using the *remove* operator. After that, we rename the attribute `PLZ_a` back to `PLZ`.

Since SECONDO has support for spatial data types, it also provides a spatial join operator. For this operation, several variants are implemented, too. The following example uses the *itSpatialJoin* operator. It gets two tuple streams each containing a spatial attribute (point, points, line, region). It combines those tuples where the bounding boxes of the spatial attributes intersect. Note that this join produces candidate pairs. A check for a real intersection of the attributes must be done after the join. The next command finds all pairs of streets in Berlin that intersect each other:

```
query strassen feed
  strassen feed {a}
  itSpatialJoin[GeoData, GeoData_a]
  filter[.Name < .Name_a]
  filter[.GeoData intersects .GeoData_a]
  project[Name, Name_a]
  consume;
```

5.14.2 SQL-like Language

SECONDO's SQL dialect supports joins in the `where` clause, but no explicit joins in the `from` clause. The join algorithm is chosen automatically by the optimizer.

Again, we search for city names contained in other city names restricted to names beginning with an *A*.

```
select *
from [plz as p1, plz as p2]
where [p1.Ort starts "A", p2.Ort starts "A",
  tolower(p1.Ort) contains tolower(p2.Ort), p1.Ort # p2.Ort]
```

The next example extends the tuples from the relation *Orte* by the zip code from the relation *plz*:

```
select [o.Ort as Ort, o.Kennzeichen as Kennzeichen, o.Vorwahl as Vorwahl,
  o.BevT as BevT, p.PLZ as PLZ ]
from [Orte as o, plz as p]
where o.Ort = p.Ort
```

Of course, the optimizer also supports spatial joins:

```
select [s1.Name, s2.Name]
from [strassen as s1, strassen as s2]
where [s1.Geodata intersects s2.Geodata, s1.Name < s2.Name]
```

The last query written in restricted form is:

```
sql
select [s1:name, s2:name]
from [strassen as s1, strassen as s2]
where [s1:geodata intersects s2:geodata, s1:name < s2:name].
```

5.15 Exporting Data

To use SECONDO's computation results in other programs, there are some operators able to export tuple streams into files. As for the import, several formats are supported. Here, the most important ones are explained. Such specialized operators are only available in the executable language.

For exporting a table for spreadsheet programs like LibreOffice Calc, the operator *csvexport* can be used. The export is limited to a small set of types. For showing the set of supported types, the following command can be used:

```
query SEC2TYPEINFO feed
  loopjoin[.Type kinds filter[. = "CSVEXPORTABLE"]
    namedtransformstream[Kind]]
  project[Type]
  consume;
```

The operator *csvexport* requires at least three arguments. The first argument is the tuple stream to be exported. The second argument has the type string or text and identifies the name of the file the tuples should be exported into. The third argument of type boolean states whether an existing file should be extended (TRUE) or overwritten (FALSE). An optional fourth argument can be used to write the attribute names of the relation in the first line of the output. After that argument another argument can be given changing the separator that is a comma by default.

The following command exports the *plz* relation into a file called `plz.csv` with a `#` as separator character:

```
query plz feed csvexport['plz.cvs', FALSE, TRUE, "#"] count;
```

For exporting spatial attributes, the shape file format can be used. This is done by *shpexport*. This operator gets a tuple stream, a file name, and a name of a spatial attribute in the stream. Optionally, the name of the index file can be a further argument. Without this argument no index file is created. This operator will export one spatial attribute only and not the standard attributes contained in the stream. The result of this operator are the unchanged tuples from the input stream. For exporting the standard attributes, the *db3export* operator can be used. It exports exactly such attributes that can be represented within a DBase-III file. Other attributes are ignored. With this knowledge, we can export our *streets* relation into a set of files.

```
query streets feed
  shpexport['streets.shp', Geometry, 'streets.shx']
  db3export['streets.dbf']
  count;
```

Of course, the operator *db3export* can also be used outside a spatial context.

5.16 Writing Scripts

Often, the same processing has to be performed on different data. It would be arduous to enter the same set of commands again and again for each data set. Hence SECONDO provides the possibility to write commands into a text file and after that, this file can be executed by SECONDO. Such a file just contains the commands as they are entered directly into the user interface.

Within a TTY environment, the commands for executing such a script are:

- `@<filename>` runs a script ignoring errors
- `@@<filename>` runs a script and stops after the first failed command
- `@%<filename>` runs a script ignoring some special comments

- `@&<filename>` runs a script ignoring special comments and stops at the first occurrence of an error

In SECONDO's graphical interface, scripts can be executed via the menu entry `Program` → `Execute file`. Also here it can be selected whether the script runs until the end or stops at the first error.

In the `bin/Scripts` directory, some example scripts can be found.

6 The Javagui

6.1 Preface

Aside from the command line interfaces in SECONDO there is a convenient graphical, window-oriented user interface implemented in Java. Its main features include:

- The Javagui can be executed in any system in which a Java virtual machine (Ver. 1.5.0 or higher) is installed. Hence in contrast to other SECONDO components it can run in a Windows environment, connecting to a server running under Linux or MacOS.
- It provides a large set of viewers to display a lot of different types (e.g., spatial data types).
- Data of different formats can be imported.
- Query results can be saved into a file.
- New viewers can be added.
- Javagui supports the SECONDO optimizer.

In the section "Javagui in General" we describe the basic functionality of Javagui, i.e., starting the gui, appearance, handling and so on. The "Java configurations" section holds some information about configuration possibilities of the gui. Finally the "Viewer" section sketches some important viewers.

6.2 Javagui in General

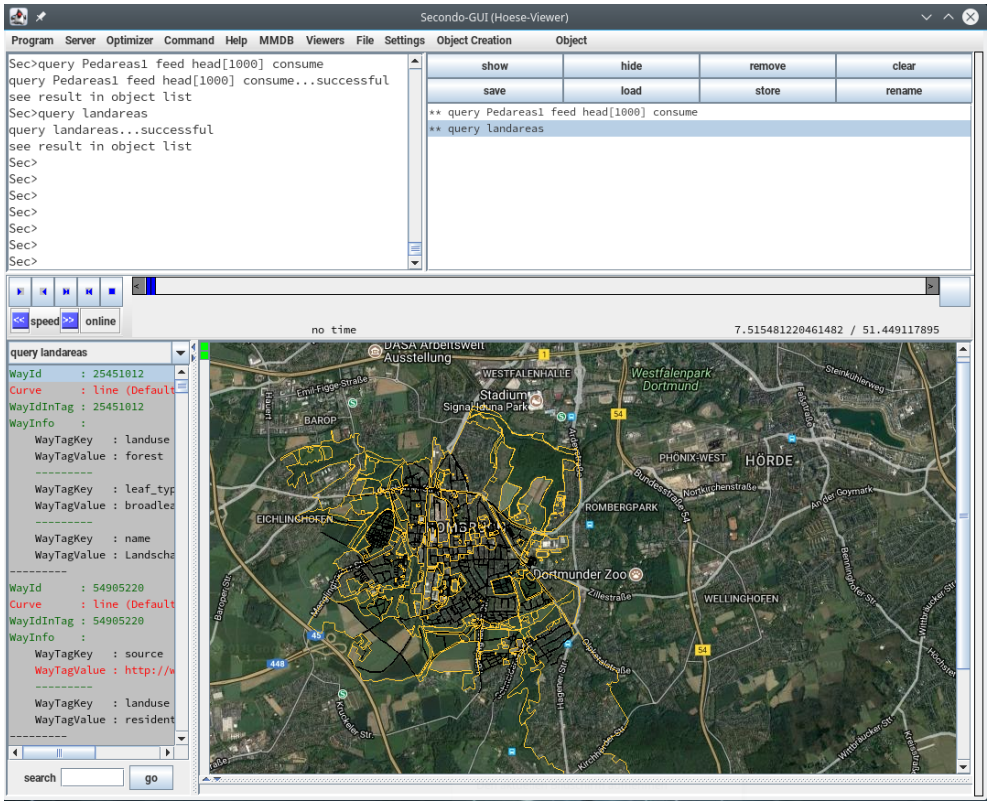

Figure 8: Javagui-Layout (Hoese Viewer)

Introduction The easiest way to start Javagui is to call the `sgui` script. Remember to start the SECONDO Listener process before executing the script (see Section 3.3.1). For optimizer functionality, ensure that the Optimizer Server is also running (see Section 3.3.4). After some license information, a window will appear on the screen (Figure 8). This window has three main parts:

the `Object Manager` (top right window), the `Command Panel` (top left window) and the current viewer part bottom. When a long running query is executed, a progress bar will appear on the right side of the gui window. On top of the gui, you find the menu bar. The first two parts will be briefly introduced in the following sections. For a detailed description of the respective viewer parts, please refer to suitable chapters concerning the different viewers (Section 6.4). You can load a viewer via the `Menu Bar` (`Viewers→AddViewer`), by selecting the appropriate viewer class file or you can insert the name of the viewer in the gui configuration file (Section 6.3). The standard configuration file `gui.cfg` is located in the `Javagui` directory of your SECONDO installation and is loaded when you start the Javagui with the `sgui` script. You can specify other configuration files and start the gui with their presettings without touching the `gui.cfg` file. We come back to that in Section 6.3.

The Command Panel Using the `Command Panel`, the user can execute commands and queries. After the prompt `Sec>`, commands terminated by return can be entered. The command is stored in the history. A history entry can be selected by the cursor-up and cursor-down keys. Similar to the TTY based interface, you can use `shift+tab` keys to extend the current input to known words. All SECONDO commands are available and you can use SQL syntax if the optimizer is activated. You can paste commands by using `CTRL-C` and `CTRL-V`. Each non-empty query result requested in the Command Panel is sent to the Object Manager and shown in a viewer according to the viewer priority settings (Section 6.4.1). If no other viewer is found, which is capable to display the requested object, the StandardViewer is used.

If the optimizer is enabled, queries and updates in SQL syntax are possible. All queries beginning with `select` or `sql` are send to the Optimizer Server to get a query plan. Embedded SQL queries are also possible. If the command starts with `insert into`, `delete from`, or `update <identifier> set`, also the optimizer is used to get an executable plan for that statement. The received plan is sent to the SECONDO Server for execution.

Additionally, some commands exist to control the behavior of the `Javagui`, as shown in Table 5.

Command	Description
`gui exit`	Closes the connections to SECONDO and the Optimizer and quits Javagui.
`gui clearAll`	Removes all objects from Javagui and clears the history.
`gui addViewer <viewer name>`	Adds a new viewer at runtime. 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 listed in the Object Manager in the current viewer whose types are supported by this viewer.

Table 5: Gui Commands ⋮..

Command	Description
`gui hideObject <ObjectName>`	Removes the object with the given name from the current viewer.
`gui hideAll`	Removes all objects from the current viewer.
`gui removeObject <ObjectName>`	Removes the object with the given name from the Object Manager and from all viewers.
`gui clearObjectList`	Removes all objects from the Object Manager and all viewers.
`gui saveObject <ObjectName>`	Opens a file dialog to save the object with the given object name.
`gui loadObject`	Opens a file dialog to load an object.
`gui setObjectDirectory <directory>`	Sets the object directory. This directory is initially shown when a load or save command is executed.
`gui loadObjectFrom <Filename>`	Loads the object with the specified filename. The file must be located in the object directory.
`gui storeObject <ObjectName>`	Stores an object into the currently open database. The object name must not 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`	Hides the Command Panel and the Object Manager. To show the hidden components use the Viewers entry in the `Menu Bar` (`Viewers`→`Show all`).
`gui executeFile [-i] <filename>`	Batch processing of the file. If -i is set, file processing continues even if an error occurs.
`gui status`	Displays information about the connection to SECONDO as well as the name of currently open database.
`gui set`	Can be used for changing the values of some Javagui settings. The complete list of the variables can be obtained by the help menu entry. The effect of the variables is described in the configuration file of Javagui.

Table 5: Gui Commands

The Object Manager This panel manages all objects resulting from queries or file input operations. The manager provides a set of buttons described in Table 6.

Button	Description
`show`	Shows the selected object in the viewer depending on priority settings.
`hide`	Removes the selected object from the current viewer.
`remove`	Removes the selected object from all viewers and from the Object Manager.
`clear`	Removes all objects from all viewers and also from the Object Manager.

Table 6: Features of the Object Manager

⋮..

Button	Description
`save`	Opens a file dialog to save the selected object to a file. If the selected object is a valid SECONDO object and the chosen filename has the suffix `obj`, then the object is saved in a format (see Figure 21 for an example) that can be restored using the `restore` command (Section 4.1.9) in the `Command Panel`. Otherwise only the value description and the type is stored. In both cases one can use the `Load` button for importing. Note that the object is not restored automatically in the current database. For this you first have to rename the object name `File: filename` to a valid object name and then use `store` as a second step.
`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 for shape and dbf files exist. The generated object name in the `Object Manager` has the format `File: filename`.
`store`	Stores the selected object into the currently open database. The object name must be a valid identifier.
`rename`	Replaces the Object Manager by a dialog to rename the selected object.

Table 6: Features of the Object Manager

The Menu Bar The Javagui `Menu Bar` consists of two parts: one depending on the current viewer and another one which is independent from it. The description in Table 7 includes only viewer-independent parts.

Menu	Submenu/Menu Item	Description
`Program`	`New`	Clears the history and removes all objects from Javagui. The state of SECONDO (opened databases etc.) is not changed.
	`Fontsize`	Here, the fontsize of the Command Panel and Object Manager can be changed.
	`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. It can be chosen how errors are handled. Note, there exist two different script styles which are described and can be selected in the configuration file.
	`History`	In this menu the current history can be manipulated.
	`Favoured Queries`	Here you can manage frequently used queries for easy access.
	`Snapshot`	Stores a Picture of the Javagui window into a file as png image. The key combination `<alt C>` can also be used to create a snapshot.
	`Snapshot as eps`	Stores a picture of the Javagui window into a file as eps image. The key combination `<alt shift C>` can also be used to create an eps snapshot.
	`Screen snapshot`	Works similar to the `Snapshot` menu entry but creates a snapshot of the whole screen instead of only the Javagui window.
	`Exit`	Closes the connection to SECONDO and quits Javagui.
`Server`	`Connect`	Connects Javagui to SECONDO.

Table 7: The Menu Bar ⋮..

Menu	Submenu/Menu Item	Description
	`Settings`	Shows a dialog to change the address and port used for communication with SECONDO. Permanent changes of these values can be performed with the configuration file.
	`User settings`	If authorization is enabled (off by default), the username and the password can be entered here.
`Optimizer`	`Enable`	Connects Javagui to the Optimizer Server.
	`Disable`	Closes the connection to the Optimizer Server.
	`Command`	In this menu, the update functions of the optimizer for relations and indexes can be called.
	`Settings`	Opens a dialog to change the settings of host name and port number of the Optimizer Server.
`Command`		This menu contains all available SECONDO commands. Menu entries beginning with a ~ require additional information. If such an entry is selected, a template of the command is printed out to the Command Panel. Other commands are processed directly without further user input.
`Help`	`Show gui commands`	Opens a new window containing all gui commands (see Table 5).
	`Show secondo commands`	Shows a list of all known input commands supported by Javagui.
	`Show support input formats`	Shows a list of all known SECONDO input formats.
`Viewers`	`<name list>`	All known (loaded) viewers are listed here. By choosing a new viewer the current viewer is replaced by the selected one.
	`Set priorities`	Opens a dialog to define priorities for the loaded viewers (see Figure 11).
	`Add Viewer`	Opens a file input dialog for adding a new viewer at runtime.
	`Show only viewer`	Hides 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.
	`Show in own window`	The current viewer is shown in an own window.

Table 7: The Menu Bar

The Menu Bar - MMDB In the last paragraph we omitted a feature of the Javagui, the Main Memory Database application (MMDB). It allows one to process objects loaded into the Javagui by further queries. Note that these objects reside in memory allocated to the Javagui, not in kernel memory. Hence the somewhat expensive transfer of these objects from the kernel to the GUI does not need to be repeated for MMDB queries.

To realize this functionality, data types and operations had to be reimplemented in Java. Therefore only a small part of the types and operations available in the SECONDO kernel is available here.

You can select the MMDB module in the Menu Bar. Two styles of querying are supported: (i) You can make simple requests via gui (see Figure 10) or (ii) you can write textual queries in the Command Panel just as for the kernel using the prefix `mmdb`.

The MMDB module supports a special MM-Object Representation. Objects in main memory are represented in the Object Manager by an additional symbol:

"[+]" The object exists exclusively as a main memory relation.
"[++]" The object also has a usual SECONDO object (nested list) representation.

For an object to be displayed in the Javagui, it must have a nested list representation. So a conversion from MMDB object to a nested list representation must be performed. Non-MMDB objects shown in the `Object Manager` already have such representation. An example is presented later. First we give an overview of the `Menu Bar` items in Table 8.

Menu	Description
`Load Object from Query`	Queries formulated in the Command Panel are sent to the SECONDO kernel. If the command is executed successfully and the received result is an object whose (attribute) types are supported, a main memory object is generated and the result is displayed in the Object Manager.
`Load Object from Explorer`	When an object is already listed in the Object Manager, a main memory object is generated for the object if the object is selected.
`Load Objects from Database`	This feature allows all objects in the open database to be loaded in the MMDB with only one command.
`Convert selected Object to NL`	Converts an (MM) object selected in the Object Manager into nested list representation, if no such representation already exists.
`Convert all Objects to NL`	All selected (MM) objects are converted into nested list representation. After this operation has finished, a window listing the failures (objects that could not be converted) is displayed.
`Autoconvert query results to NL format`	The option can be used to decide whether Command Panel query results are directly converted to the nested list format and displayed or only saved as in MMDB representation.
`Export MM-Object(s)`	You can select MM objects from the object list of the Object Manager and export them into a file.
`Import MM-Object(s)`	You can import MM objects stored in MM files (`*.secmm`).
`Supported Operators`	Shows a list of supported operators.
`Programmer's Guide(PDF)`	Opens the Programmers' Guide for creating new MMDB operators.
`Generate Index`	It is possible to create indices on certain attributes. These will be used automatically during query executing in order to accelerate processing time. In the index creation dialog, first select a relation, afterwards an attribute to be used for indexing and finally the type of index you would like to create. Only indexable attributes are displayed. The current implementation does not allow one to create several indices on one attribute.

Table 8: The MMDB module ⋮..

Menu	Description
`Execute Query`	Six top level query operations are available: SELECTION: Select a tuple subset depending on a condition. PROJECTION: Select an attribute subset from the relation's tuples. EXTENSION: Add new attributes to the relation's tuples via an operation. UNION: Merge the tuples of two relations. Attribute sets must be identical. JOIN: Merge attributes of two different relations depending on a condition. AGGREGATION: Calculate aggregations of a certain attribute for all tuples of a relation.
`Manage Memory`	Since all relations are stored in main memory it might occur that the JVM runs out of memory which means that the application crashes. To prevent these OutOfMemoryErrors, memory is permanently monitored. If there is an impending overflow you will be given the chance to remove objects that are not needed anymore to free memory. This can be done any time by selecting this menu item. Besides the dialog runs an asynchronous thread which performs garbage collection once at startup and at specified time intervals (default = 60 sec).
`Supported Types`	Shows a list of supported data types.
`Help`	Shows the MMDB help text

Table 8: The MMDB module

MMDB - Example For the example we use the database *opt*. Open the database with the command

```
open database opt
```

or restore it if it is not present by executing

```
restore database opt from '../bin/opt'
```

in the Javagui Command Panel. We switch to `MMDB`→`Load Objects from database` and import all the objects into the MMDB. After executing and confirming the import, we can see that 7 objects have been imported in main memory (see Figure 9). They are named $R1$-$R7$ in this example. Notice as well that not all of the objects in the opt database have been imported due to some incompatibilities.

The "[++]" on the right side of the object names in the `Object Manager` indicates that the objects already have a nested list representation, so you can visualize a relation by double clicking on the object or selecting the show button in the `Object Manager`. Next we choose `MMDB`→`Execute query` and the window that can be seen in Figure 10 pops up. Choose `SELECTION` and select the object $R1$:`query Orte[++]`, the `OPERATOR GREATER`, the `ATTRIBUTE BevT(int)`, the int value `FROM TEXT` and type in `1000` in the input field. Choose `CONVERT RESULT RELATION AUTOMATICALLY TO NESTED LIST` if you like. Otherwise you have to convert the result later for visualization purposes. Press `EXECUTE`. Another MM object appears in the `Object List` of the `Object Manager` ($R8$:`SELECTION ON query Orte; [++]`). The result of this query comprises 4 tuples with 4 attributes including cities with more than 1,000,000 inhabitants.

Via gui you can choose only one operation at each step. For example a projection on the selected tuples above cannot be done in one step. So if you want to know only the city names that have more than 1,000,000 inhabitants, you have to execute the projection on the object $R8$ as a second step manually.

Fortunately you can use the Command Panel with the `mmdb` query prefix to execute more complex queries utilizing more than one operation:

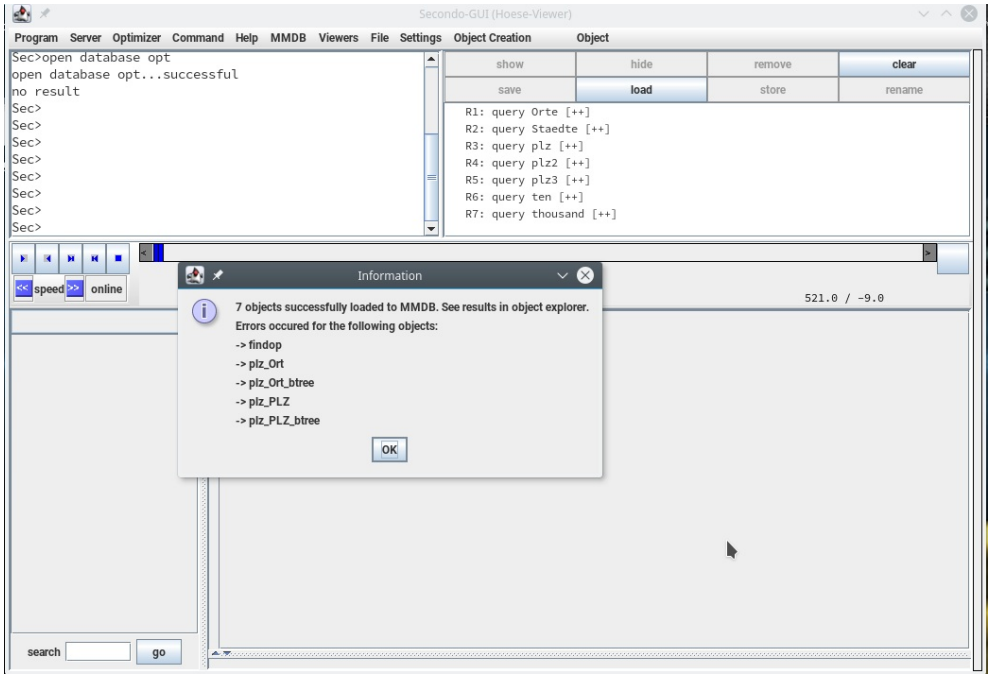

Figure 9: MMDB - Import from database

```
mmdb query R1 feed filter [.BevT > 1000] project[Ort] consume
```

It is important to notice at this point that we do not use `Orte` here, because in main memory the same relation is bound to the name $R1$.

6.3 Javagui Configurations

As already mentioned above, Javagui can be started by executing the script `sgui`. In this process, the specifications that are defined in the `gui.cfg` configuration file are considered. For other purposes you can create other configuration files and execute them using the `-config` option. For example, with the command `sgui -config myconfig.cfg` Javagui is started with the configurations defined in the file `myconfig.cfg`. So the `gui.cfg` does not have to be changed, if you want to start Javagui with other parameters.

In Table 9 we list the most common configuration options. Feel free to look at the default configuration file for more configurable parameters.

Parameter	Description/Settings
SERVERNAME	Hostname or IP of the SECONDOServer. Default is local host: 127.0.0.1
SERVERPORT	TCP port of the SECONDO Server. The default value is 1234.
KNOWN_VIEWERS	A list of automatically loaded viewer classes (must exist in Javagui/viewer/). With the standard configuration `StandardViewer`, `RelViewer`, `FormattedViewer`, `HoeseViewer`, `UpdateViewer` and `UpdateViewer2` are included with ascending priority order.
SECONDO_HOME_DIR	The SECONDO directory can be specified. The default is the directory above Javagui if no other value is set.
START_CONNECTION	If set to `true`, automatically connects to the SECONDO Server on startup. The default value is `true`.

Table 9: Javagui configuration file options (excerpt)

Parameter	Description/Settings
OBJECT_DIRECTORY	You can set the directory for loading/saving objects. The default is Secondo_Home_Dir/Data/GuiData/objects.
HISTORY_DIRECTORY	Set directory for loading/saving histories. The default is Secondo_Home_Dir/Data/GuiData/histories.
STARTSCRIPT <file> $[-i]$	You can execute a file at start, for example, a saved history. The parameter $-i$ ignores errors.
COMMAND_FONTSIZE	The font size in the Command Panel can be set. Default value is 14.
LIST_FONTSIZE	Set font size for the Object List in the Object Manager. Default value is 12.
OPTIMIZER_HOST	Specify the host-name or the IP adress of the OptimizerServer. The default is localhost.
OPTIMIZER_PORT	The port number of the Optimizer Server can be set. The default port number is 1235.
ENABLE_OPTIMIZER	Enable optimizer at start (true (default) or false).
SHOW_COMMAND	If this variable is set to true, each command is printed out before it is sent to the SECONDO server.
ENCODING	Specify the encoding. Standard value is ISO-8859-1. For example, you can use UTF8 instead.
KEEP_CURRENT_VIEWER	If this is set to true (default), the current viewer is kept if it is possible to display the current object with it, even if another viewer exists that can represent this object in a better way.
OBJECT_DEPENDING_VIEWER_SELECTION	Set it to true if you want to use the object-depending selection of the viewer. Default value is true.
EXTENSIONS	You can specify a set of useful extensions (use shift + tab to extend a word). You can add words in the default list as given in the config file.

Table 9: Javagui configuration file options (excerpt)

6.4 Viewers

6.4.1 Introduction

As we mentioned in previous sections, there are three ways to load a viewer in Javagui. You can do it via a gui command in the Command Panel, via the Menu Bar or you can insert the viewer's name into the gui configuration file. Keep in mind that only the third approach will add the specified viewer permanently to Javagui. There are different viewers which can display the same data type(s). To select one of these viewers, priorities are used.

The initial order of priorities after the Javagui start is determined by the sequence of viewers in the configuration file. The last viewer of that list has the highest default priority.

In the priority dialog of the Menu Bar (Viewers → Set priorities) you can change the priorities of the loaded viewers depending on your personal preferences (Figure 11). The viewer at the top has the highest priority. In order to change the position of a viewer, select it and use the up or down button. If depending from object is selected, Javagui asks the viewers about their display capabilities for a specific object and uses this information to select the right viewer. In the case of equal display capabilities, the priorities of the viewers are making the difference and the viewer with the highest priority is chosen. If the box try to keep the current viewer is selected, the current viewer is only replaced by another one if it cannot display the object. It is important to repeat that the gui.cfg configuration file parameters KEEP_CURRENT_VIEWER and

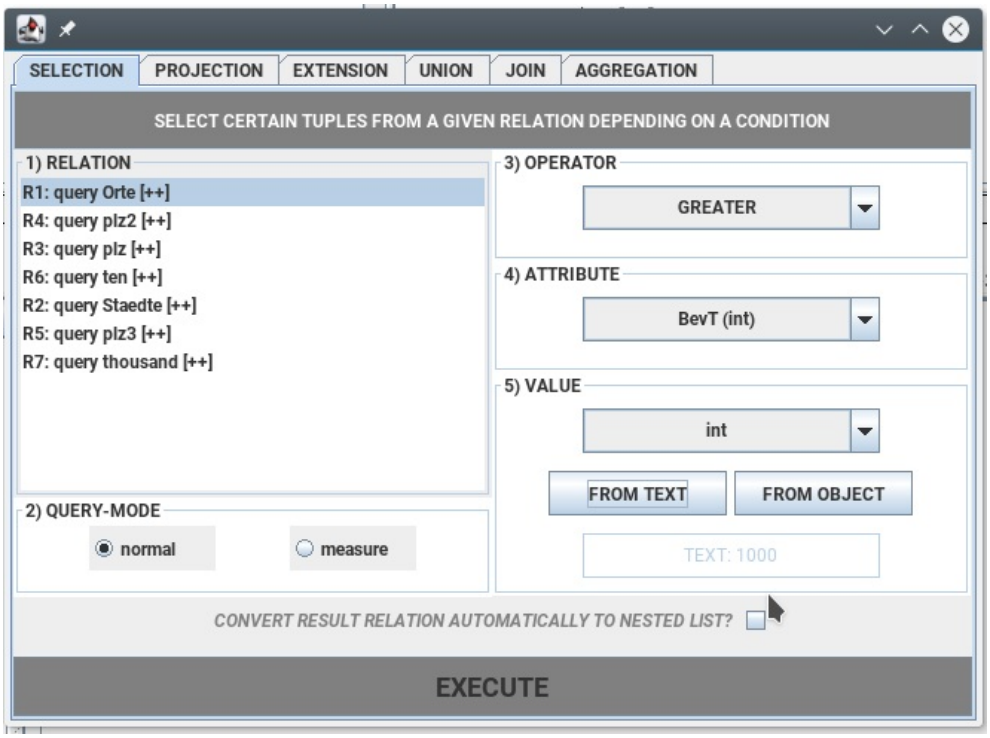

Figure 10: MMDB - queries via gui

`OBJECT_DEPENDING_VIEWER_SELECTION` are both set to `true` by default (Section 6.3). Furthermore, `depending from object` and `try to keep the current viewer` initially are set or not set depending on these configuration file parameter values. Consequently by default they are both enabled at Javagui start.

Keep in mind that changes via the `Set priorities` menu are discarded and the default state (depends on the configuration file entries) is restored after a Javagui restart. In this section we present the most important viewers and their basic functionalities. You can check out other viewers in the viewers directory of SECONDO (`/secondo/Javagui/viewer`).

6.4.2 HoeseViewer

The HoeseViewer (Figure 12) is very powerful as it is able to display a lot of different SECONDO object types. The viewer consists of several different parts to display textual, graphical, and temporal data.

On the left, you find a text panel and on the right the graphical panel is located. Above the text panel you can see a combo box, some buttons and to the right of these there is a time line. If an object in the textual part is selected, then the corresponding graphical representation is also selected (if it exists) and vice versa.

In the `Menu Bar` there are four more menu entries: `File`, `Settings`, `Object Creation` and `Object`.

The Textual Representation of an Object Using the combo box at the top of the text panel you can choose another object (query result) to display. A string in the text representation of the selected object can be searched 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 the list is reached, the search continues at the beginning.

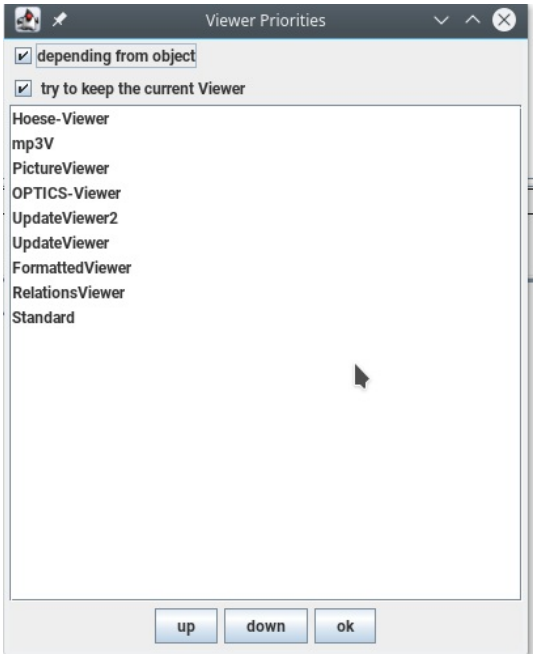

Figure 11: Viewer priorities

The Graphical Representation of Objects The graphic panel contains geometric and/or spatial objects. Press the right mouse button and drag the mouse holding the right mouse button for zooming in. Stepwise zoom in and zoom out is available in the `Settings` menu (`Zoom +` / `Zoom -`) or by pressing `Alt +` / `Alt -`. To get an overview of all objects click on Zoom out in the Settings menu or press `Alt z`.

Each query result is displayed in a single layer. Using layers, the order in which the objects are displayed can be changed. 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 using the Object menu. Here, the user can also change the display settings for a single selected object.

The menu `Settings`→`Projections` offers the possibility to enable one of a set of projections. This is helpful for displaying data containing geographical coordinates (longitude, latitude). The usual view of such data is obtained using the Mercator or the Gauss-Krueger projection.

The `Object` menu allows to manipulate the appearance of an object: You can choose `Hide`, `Show`, `Change category` or `Label attributes`. Furthermore the objects layer position can be changed.

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

Categories A category contains information about how an object is to be displayed. Such information contains color or texture of the interior, color and thickness of the borderline, or size and shape of a point. Categories can be loaded and saved via the `File` menu. To edit an existing category, the `category editor` (see Figure 13) available via the `Settings` menu has to be invoked. There are several possibilities to assign a category to an object or to attributes of a relation. The method can be chosen in the `Settings` menu. If `Category: manual` is chosen, a selection window pops up for each object (or for each graphical attribute of a relation). `Category: auto` creates a new random category for each graphical object. If `Category: name` is used, two cases are distinguished. First, if the name of the object (attribute) is equal to the name of a category, this category is

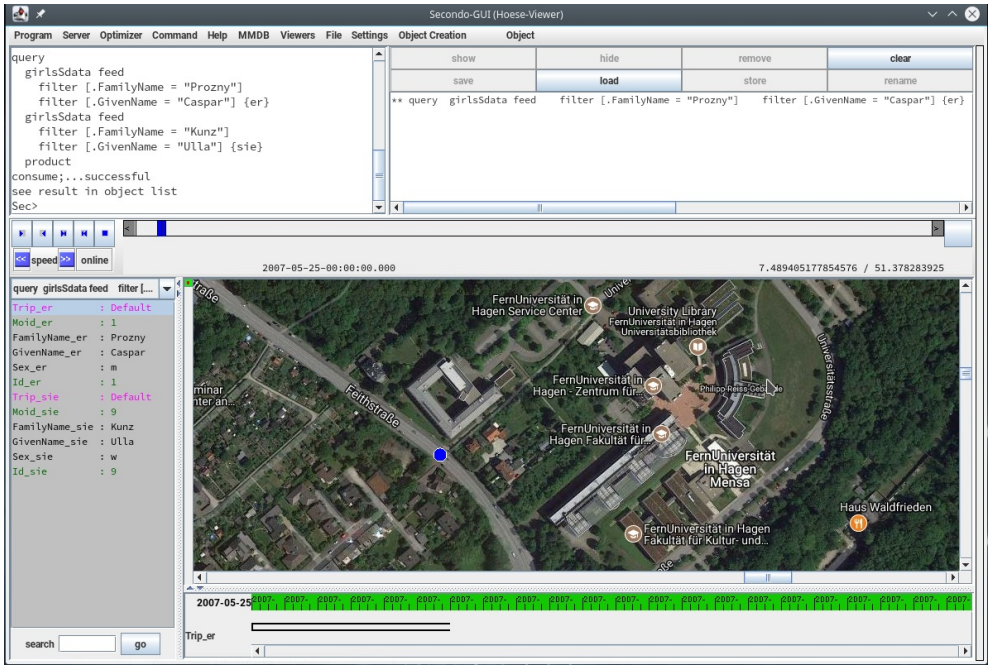

Figure 12: HoeseViewer in action

chosen automatically. Otherwise, the user is asked for a category.

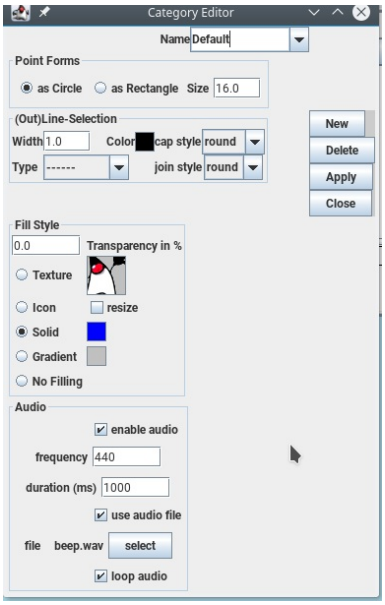

Figure 13: Category Editor

Query Representation In this window the user can alter settings for displaying a query result with graphical content (see Figure 14), i.e., a single graphical object or a relation with one or more graphical attributes. At the top the user can choose an existing category for all graphical objects of this query with the same (attribute) name. The button labeled with "..." invokes the `category editor` to create or change categories. A graphical object may have a label. whose content can be entered as `Label Text`. If the object is part of a relation, the value of another or even the same attribute can be used as label. This feature is available in the `Labelattribute` combo box. In this case, the user can also customize graphical settings for objects contained in the relation. If `Single Tuple` is selected, an own category can be chosen for each single tuple in the relation.

Another possibility is to choose the category depending on an attribute in the relation. Thus, the point size, the line width or the color can be chosen to be dependent on the value of another attribute. The possible values for these features are distributed in a linear way over the values of the selected attribute. For a non-linear distribution or for attribute values which do not support this function, a manual link between value and category can be created.

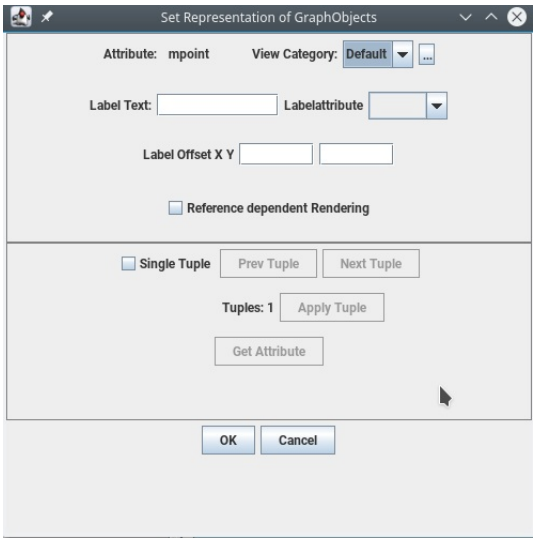

Figure 14: Query Representation

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

Displaying Special Objects Some objects can be displayed in a separate window. These objects are marked by a special color in the textual representation. By double clicking on the object, an additional window is opened displaying the selected object.

Managing Backgrounds The background of the graphic window can be changed by the user. The color can be chosen by invoking the `Settings`→`Background`→ `Color: Choose` menu. This color is used if no background image is given and for all areas not covered by the background image. A background image can be used to show the context of other objects (... → `Image: Import`). For positioning the image, its bounding box must be defined together with the image. For simplifying the positioning, so-called `tfw` files can be used. Such files are also used in geographic information systems. Another possibility to set the background is to capture the current display as background (... → `Image: Capture All`, ... → `Image: Capture Visible`). This may be useful if many non-moving objects are displayed and additional moving objects are animated. After capturing the static objects as background, they can be removed from the display to reduce the computation effort during the animation.

When objects having geographic coordinates (longitude/latitude) are displayed, it is also possible to use maps from *OpenStreetMap* or *Google* as a background. This requires an internet connection. Although this kind of background works with many projections, it is recommended to choose the `Mercator` (or even better the `OSM-Mercator`) projection from the `Settings`→`Projections` menu to avoid distortions of the maps. After choosing `Settings`→`Backgrounds`→`TiledMap(OSM,`

`GoogleMaps`) a new frame appears. Here, one of several predefined map backgrounds can be selected. Because the Google servers limit access to theirs map tiles, we recommend to use one of the `OpenStreetMap` backgrounds (Figure 15).

You can edit the map server properties (e.g. to use a different map style) and some display settings (see Figure 15).

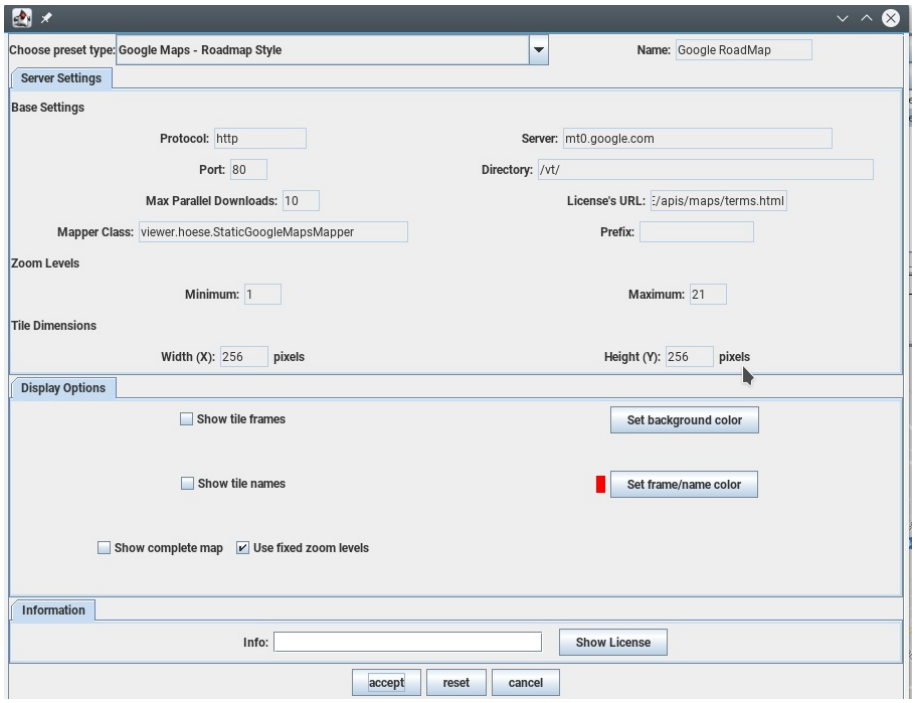

Figure 15: Setting properties for the TileMap background

Create Objects The HoeseViewer offers the possibility to create simple graphical objects. An object type can be chosen in the `Object Creation` menu. After pressing the unlabeled button (right of the time line) the object creation starts. A rectangle can be drawn by holding the left mouse button pressed and dragging the mouse. A point is created just by clicking on its location. For creating other objects, a sequence of points has to be defined by left mouse button clicks. To finish the creation of such more complex objects, the object creation button has to be pressed again. If an object is defined, it is stored into the currently open database and inserted into the `Object Manager`.

Instead of creating a lot of single objects, it is also possible to write a set of objects having the same type into a single relation. To use this feature, as a first step, a relation with schema

```
rel((tuple[Name:string, T : type]))
```

must be defined in the currently opened database. The type is the same that is to be created. It is also possible to let the gui create this relation (choose this from the `Object Creation` menu). To automatically insert newly created objects into this relation, the corresponding check box `Store in relation` must be activated within the `Object Creation` menu.

Online This feature enables Javagui to process and visualize realtime position data sent from an Android application, for example. To get this show on the road, there are more configurations and further applications needed that are beyond the scope of this manual.

6.4.3 UpdateViewer/UpdateViewer2

This section describes how to use the UpdateViewer and the UpdateViewer2. The UpdateViewer2 supports all the display and editing options offered by the UpdateViewer: displaying relations, editing attribute values, deleting and inserting new tuples. UpdateViewer2 also supports formatting and viewing structured text documents.

UpdateViewer With this viewer you can manipulate the contents of relations. It is possible to insert and delete tuples and to modify attribute values. When you have started the UpdateViewer you will see eight buttons on the top of the viewer panel (Figure 16).

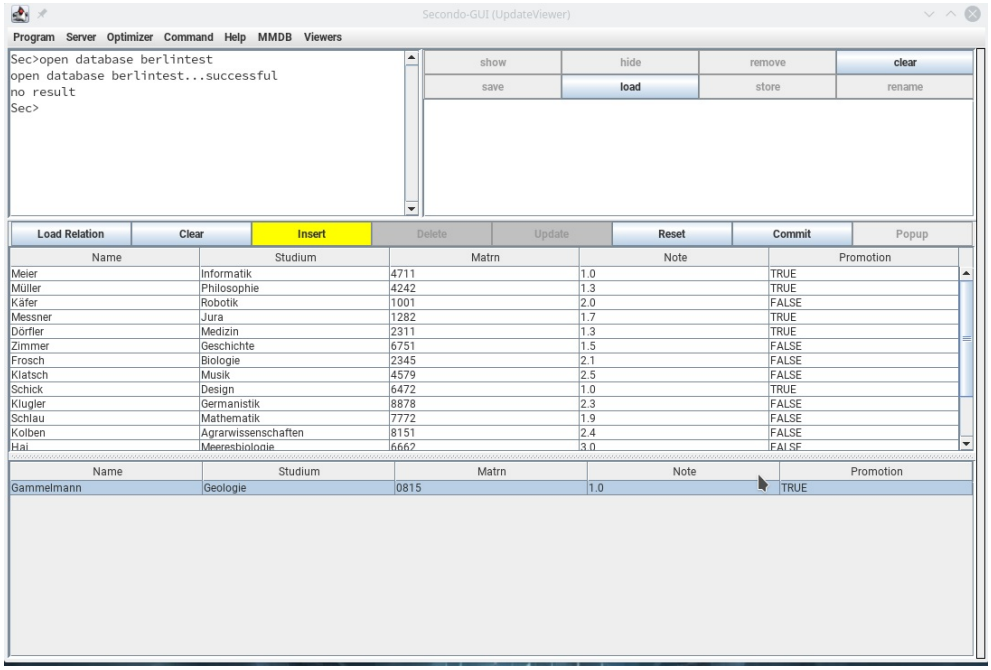

Figure 16: UpdateViewer

Button	Description
`Load Relation`	Press that button to load a relation from an open database. A window pops up where you can type in the name of the relation. Additionally you can add a filter. In the example relation of Figure 16 you can filter the relation with: `filter[.Note < 2.0]`. Press `Commit` to execute your selections.
`Clear`	Clear the whole viewer window.
`Insert`	You can insert a tuple (see Figure 16) by selecting this button. An empty tuple appears. Click in the corresponding location to enter a suitable attribute value (otherwise a type error will be prompted). Press the `Commit` button to make your entries persistent. The new tuple is appended to the relation.
`Delete`	Use this feature to delete tuple(s). After selecting `Delete` you are able to mark tuples. Use `CTRL` and the left mouse button to select more than one tuple, or use the pressed left mouse button to select consecutive tuples. Press `Commit` after your selection.
`Update`	With `Update` it is possible to change single attribute values. Press the `Update` button and then click on the value you want to modify. You can insert a suitable value directly or you can use `Popup` to do this in a new window. Click on `Commit` to commit your changes.

Table 10: Features of Update Viewer ⋮..

Button	Description
`Reset`	You can cancel an insert, delete or update operation as long as you have not yet committed it.
`Popup`	see `Update`

Table 10: Features of the Update Viewer

UpdateViewer2 - Introduction The idea of the `UpdateViewer2` is to support the editing and formatting of text documents that are generated from the information stored in several relations. A motivating example would be the handbook of modules of a university curriculum combining information from several tables. Therefore this viewer provides features to define so-called document profiles based on several relations and to search and edit text fields across relations. It also offers a comfortable presentation and editing of large text fields.

In the `UpdateViewer2` several relations can be displayed at the same time. Each of them is presented individually in a separate tab. The tuples are displayed sequentially in blocks of arranged pairs. The tuple id and attribute names are displayed on the left. The respective attribute values are shown on the right and are editable (Figure 18 or 19).

As mentioned above, the UpdateViewer2 supports all the features from the UpdateViewer. The `Clear`, `Insert`, `Delete` (see Figure 19), `Update`, `Reset` and `Commit` features work in a similar way as described in Table 10. Notice that there is no `Popup` button anymore (see Figure 19). With the new `Undo` option you are able to incrementally unmark selected tuples or to take back attribute changes unless you have executed a Commit.

There are also two new buttons: `Load` and `Format`. With `Load` one can load relations analogously to the `Load relation` feature of the UpdateViewer. To do this, hit the `Load` button. If no profile has been created yet, you will be asked to do so. The name assignment is mandatory here, but you do not have to specify the other profile parameters to use `Load directly`. Click on `Load directly` and choose your relation. Only the relations of the types `rel`, `mrel` and `orel` are offered for selection. Press `OK` to load the specified relation (see Figure 17). But there is more to say about `Load`. We discuss `Load` and `Format` later in this section.

Direct query results from the command panel of the type `rel`, `arel`, `mrel orel`, `nrel` and `trel` can be displayed in the viewer, too. Loaded relations in this way can only be viewed and searched, but not edited.

UpdateViewer2 - Search, update and replace At the bottom of the UpdateViewer2 panel, you can find the search and replace features. They work across the displayed relations. With the arrow keys `previous search hit`, `next search hit`, `first search hit` and `last search hit` one can navigate through the search results. The selection `case sensitive` will pay attention to upper and lower case letters.

After entering a search key and pressing the `Search` button, the number of hits is displayed. The hits are highlighted in the display area and the cursor jumps to the first hit. If there is no hit in any of the loaded relations, the search field is deleted.

To activate the `Replace` and `Replace all` options you must bring the viewer into update mode. Then the user can replace the search key with the string in the replace field by pressing the `Replace` button and jump to the next hit. An automatic substitution can be initiated, or the user can jump from hit to hit to decide on a case-by-case basis whether or not to replace at that position. Use `Commit` to save your changes.

The attribute values can also be changed directly in the update mode in the display area. You just have to select the value you want to edit and enter your changes. Changed attribute values are highlighted in blue. By pressing the `Undo` button, changes can be undone incrementally starting

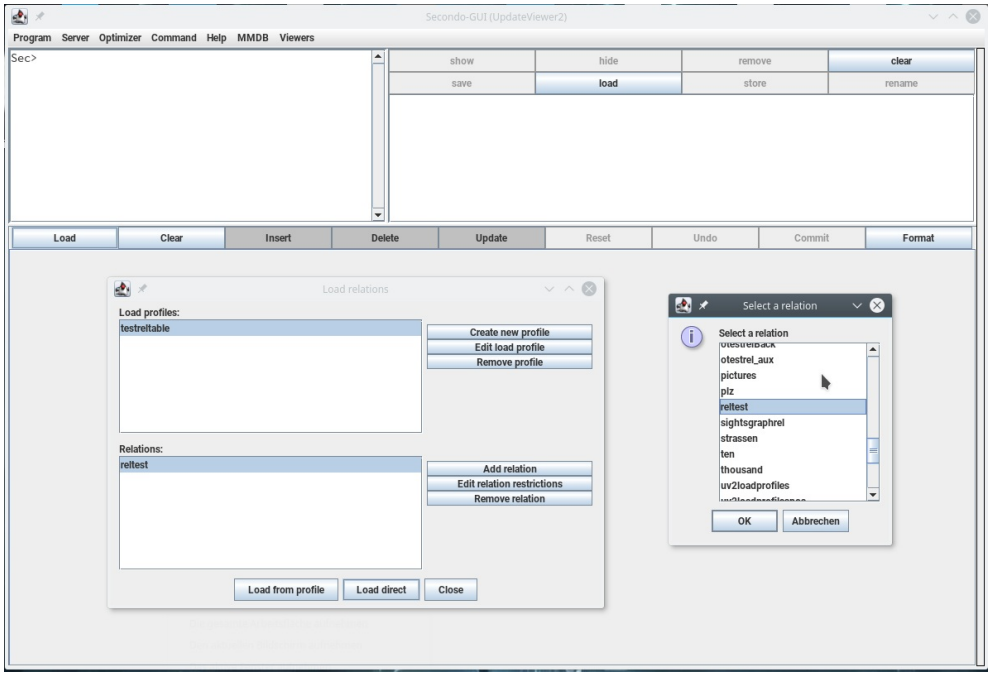

Figure 17: UpdateViewer2 - Loading a relation directly

from the last change.

With `Commit` the changes to all relations are executed. If this is successful, the update mode will be terminated, otherwise an error will occur.

`Reset` discards all changes and exits the mode.

UpdateViewer2 - Insert and Delete A new tuple is inserted in a separate area, which is displayed with the button `Insert`. Here a tuple block with empty (attributes value) text fields is shown, where you can enter the corresponding attribute values. The button `Commit` saves the new tuple persistently. If entered values have a wrong format, the user will be notified. Normally a new tuple is appended at the end of the relation unless a sorting criterion is used.

To delete a tuple, the viewer must be switched to delete mode by pressing `Delete`. Now tuples can be marked for deletion by clicking in the table areas. The block(s) will be highlighted (Figure 19). The respective last mark can be undone with `Undo`.

`Reset` discards the new tuple or the markings and exits the respective mode.

UpdateViewer2 - Formatting text documents (example 1) As noted above, the loading dialog has more features than `Load Direct`. Several relations can be loaded at the same time. For this a (document) profile and one or more relations must be selected. Use `Load from Profile` to proceed (see Figure 17). Use `Add relation` to select another relation (only the types `rel`, `mrel` and `orel` are supported) and `Remove relation` to remove a relation from the dialog. With `Edit relation restriction` it is possible to modify and define some relation restriction parameters. Profiles can also be edited, removed and of course created (Figure 17). In the dialog in Figure 17, existing document profiles are displayed. A profile contains the relations belonging to the document, the document structure and the formatting settings and it has to have a unique name. In the editor window, a window with a help text and an example is available for each parameter. The help window is displayed by right-clicking on the corresponding value area (Figure 22).

The name assignment is mandatory to create a profile, while the other parameters must be set for formatting purposes. If only `FormatType`, `FormatAliases`, `FormatQuery` and `OutputDir` are

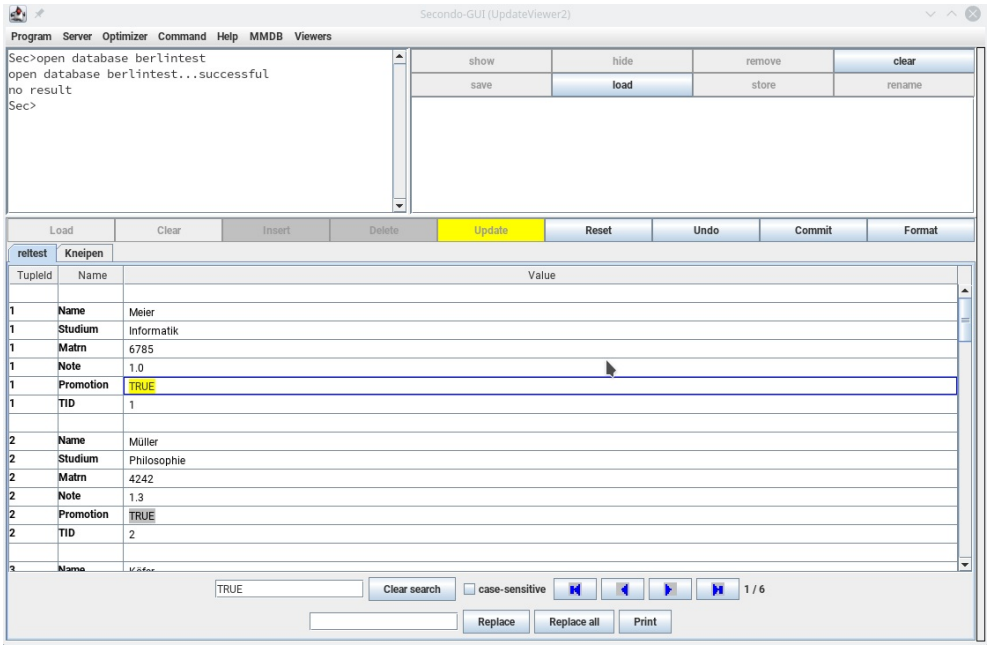

Figure 18: UpdateViewer2 - Searching and replace

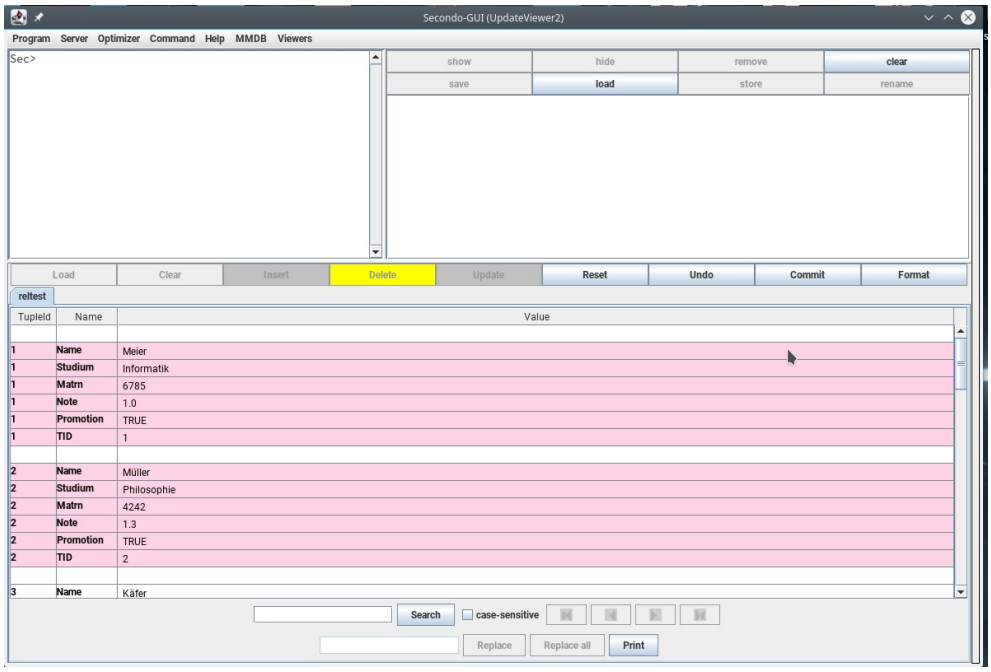

Figure 19: UpdateViewer2 - Delete

specified, a default format is used. Table 11 gives an explanation of all parameters. We will come back to format issues later, when we discuss a little example. For now we just outline the meaning of the parameters offered in the `edit load profile` (respectively `create profile`) window. Use the help function to get more details.

Parameter	Description
`ProfileName`	Name of the profile. It is freely assigned by the user.
`FormatType`	Type of the desired format.
`FormatAliases`	List of names of the relations involved in the document and the respective aliases (renames) used in `Format Query`.
`FormatQuery`	A SECONDO query that delivers the data for the document in form of a relation with attribute names that match the placeholders used in `FormatTemplateBody`.
`FormatScript`	If a document is to be reworked with the help of external tools, the path to a script is given here. To apply the script select `apply script` in the `Format Document` window.
`OutputDir`	Absolute or relative path to the directory where the formatted document is stored.
`FormatTemplateHead`	Markup template for the beginning of the top level of the document.
`FormatTemplateBody`	Markup template for the body of the document with placeholders for relation fields, e.g. attribute values to be inserted. (`<<placeholder>>`).
`FormatTemplateTail`	Markup template for the end of the top level of the document.

Table 11: Parameters of a document profile

After creating a profile, it is possible to add related relations and to set different relation settings. For example you can define filters and specify projections and orders. These relation profiles can be created or modified with `Edit relation restrictions`. Figure 20 shows the configuration possibilities.

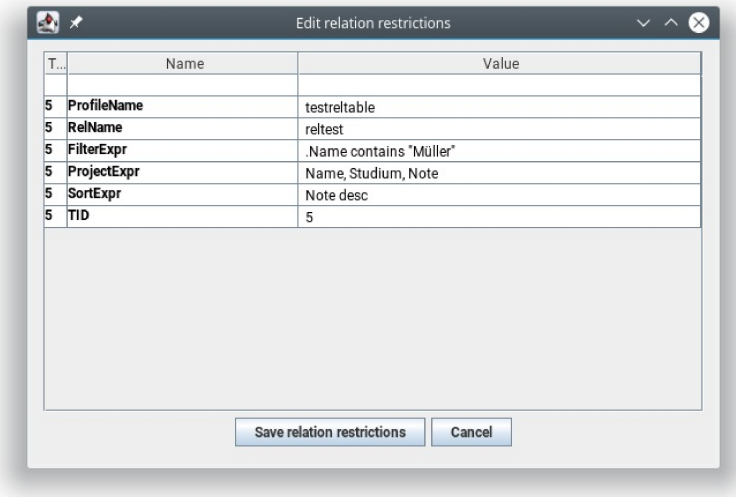

Figure 20: Relation restrictions

If you want to load only certain tuples of a relation, one or more filters can be specified. These filters have the same syntax as the *filter* operator and are separated by semicolons. For projecting on some attributes, a `ProjectExpr` has to be specified, consisting of the attribute names separated

by commas. Normally, the tuples are unordered, respectively in the order in which they are supplied by SECONDO. So with `SortExpr` a sort criterion can be defined. The syntax corresponds to the expression to be passed to the operator *sortby*.

Introducing a small example, we now demonstrate the format feature of the `UpdateViewer2` using a relation listed in Figure 21.

```
( OBJECT  reltest
   ()
   ( rel
      ( tuple
         (
            ( Name  string )
            ( Studium  string )
            ( Matrn  int )
            ( Note  real )
            ( Promotion  bool ))))
   (
      ( "Meier"  "Informatik"  6785  1.0  TRUE )
      ( "Müller"  "Philosophie"  4242  1.3  TRUE )
      ( "Käfer"  "Robotik"  1001  2.0  FALSE )
      ( "Messner"  "Jura"  1282  1.7  TRUE )
      ( "Dörfler"  "Medizin"  2311  1.3  TRUE )
      ( "Schick"  "Design"  6472  1.0  TRUE )
      ( "Klugler"  "Germanistik"  8878  2.3  FALSE )
      ( "Schlau"  "Mathematik"  7772  1.9  FALSE )
      ( "Kolben"  "Agrarwissenschaften"  8151  2.4  FALSE )
      ( "Hai"  "Meeresbiologie"  6575  3.0  FALSE )
      ( "Raum"  "Astrophysik"  3412  1.0  TRUE )
      ( "String"  "Physik"  8888  2.6  FALSE )
      ( "Tausend"  "Volkswirtschaft"  8767  3.0  FALSE )
      ( "Müller"  "Pharmazie"  7  1.7  FALSE )))
```

Figure 21: SECONDO Object (example 1)

The profile is named *testreltable* and the other format parameter values are given in Figure 22. Currently only the *html* format is supported. After saving the *html* specifications we add the relation *reltest* in the load dialog (see Figure 17). We do not use relation restrictions. After clicking on `Load from profile` we hit the `Format` button in the main window of the viewer. A `Format document` window appears. Select `Separate pages` if you like, or use `Apply script`, if a path is specified in the `FormatScript` parameter of the document profile. We select none of these and hit `Format`. The result is shown in Figure 23. The corresponding *html* file will be found in the path that was set in `OutputDir`.

An example of a relation restriction configuration is shown in Figure 20. In our little example relation the restrictions

```
.Name contains "Müller"
Name, Studium, Note
Note desc
```

will restrict the relation to

```
"Müller"  "Philosophie"   1.3
"Müller"  "Pharmazie"     1.7
```

after executing `Load from Profile`.

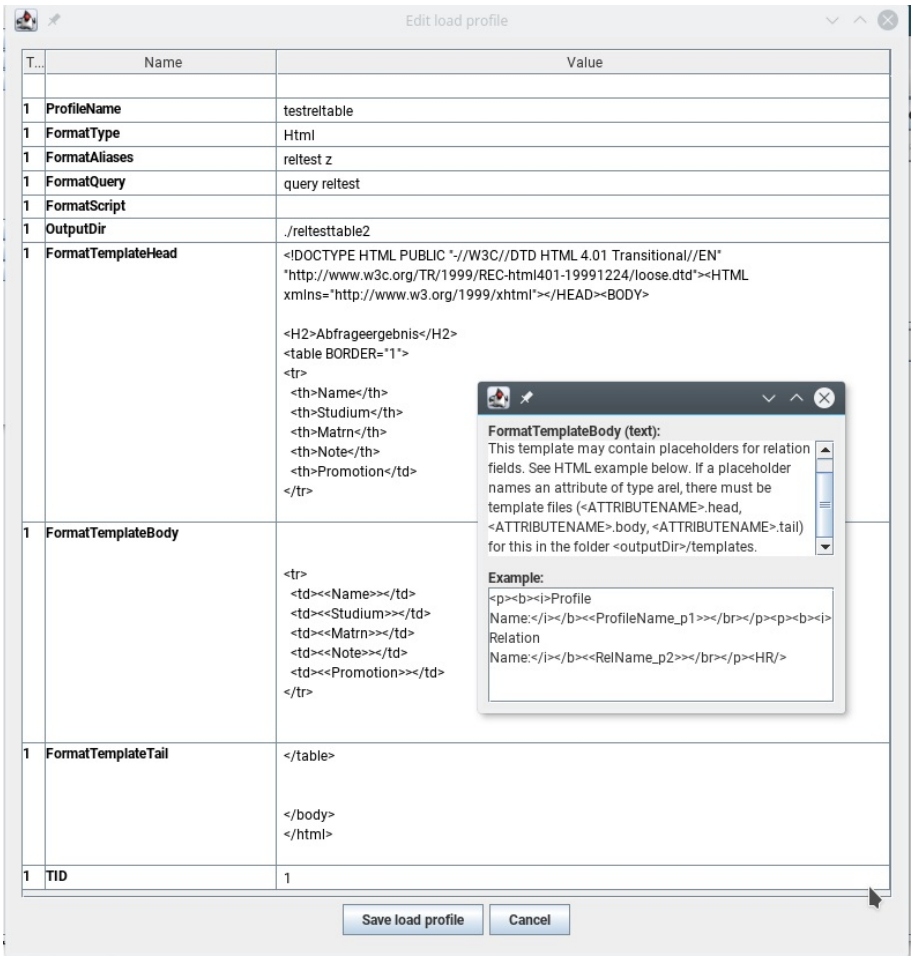

Figure 22: Edit load profile

Figure 23: Formatting example 1

UpdateViewer2 - Formatting text documents (example 2) In the second example, we format a document using a nested relation. A nested relation (type `nrel`) is a relation that may contain subrelations (type `arel`) with further attributes. Figure 24 shows an example of a nested relation with one subrelation. The top level attributes are `Course` and `Id`. The third relation field contains the subrelation with the three second level attributes `Semester`, `Grade` and `Number`.

As we already know, `FormatTemplateBody` may contain placeholders for relation fields. If a placeholder names an attribute of type `arel` (`Rating` in our example), there must be template files (`<ATTRIBUTENAME>.head`, `<ATTRIBUTENAME>.body`, `<ATTRIBUTENAME>.tail`) for this in the folder `<outputDir>/templates.` These additional files (Figure 27) define format specifications similar to the top level formatting (Figure 25). The result is shown in Figure 26.

```
( OBJECT nestedrel
   ()
   ( nrel
      ( tuple
         (
            ( Course string )
            ( Id int )
            ( Rating
               ( arel
                  ( tuple
                     (
                        ( Semester string )
                        ( Grade real )
                        ( Number int )))))))
   (
      ( "Metaphysics"  12345
         ( 0
            ( "WS17/18"  1.7  20 )
            ( "WS16/15"  2.0  11 )
            ( "SS14"     2.3  42 )))
      ( "Ethics"  10815
         ( 0
            ( "SS18"  1.3  22 )
            ( "SS17"  1.7  21 )
            ( "SS09"  2.0  12 )))
      ( "Philosophy of the mind"  47110
         ( 0
            ( "WS17/18"  1.0  21 )
            ( "WS16/17"  1.3  11 )
            ( "WS15/16"  1.3  13 )))))
```

Figure 24: Another SECONDO Object (example 2)

6.4.4 RelationViewer

The RelationViewer is a convenient way to import and export *csv* files without using any SECONDO operators directly. This viewer displays SECONDO relations as tables. It is not suitable for displaying relations with many attributes or relations containing large objects. It is possible to print out a specified relation. Refer to Figure 28 for an overview of the RelationViewer.

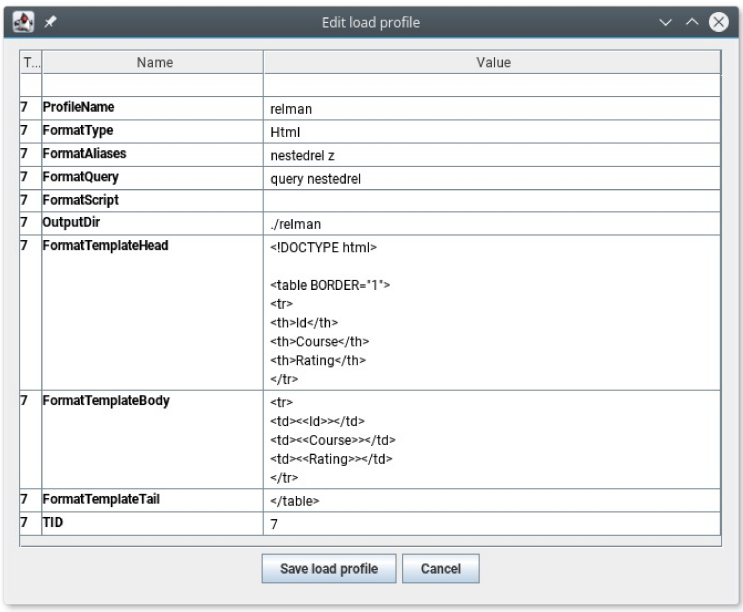

Figure 25: Example 2 Profile

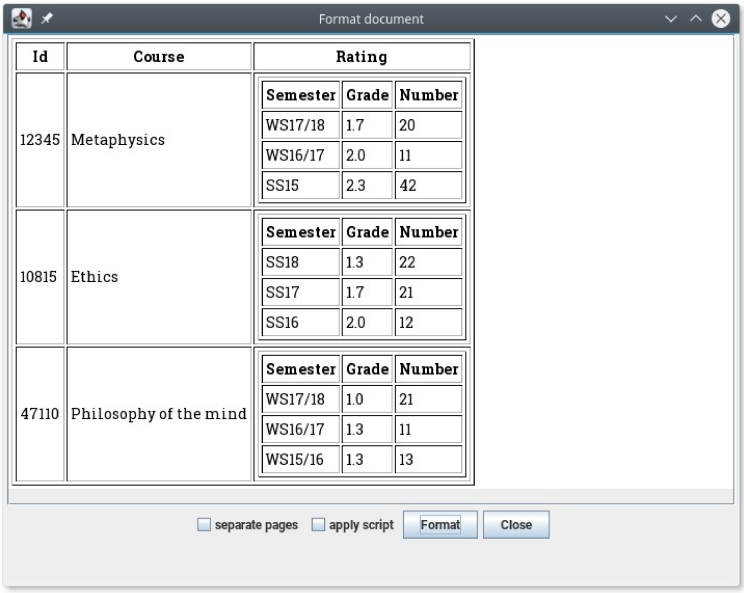

Figure 26: Formatting example 2

Rating.head	Rating.body	Rating.tail

`<table BORDER="1">`
`<tr>`
`  <th>Semester</th>`
`  <th>Grade</th>`
`  <th>Number</th>`
`</tr>`
	`<tr>`
	`  <td><<Semester>></td>`
	`  <td><<Grade>></td>`
	`  <td><<Number>></td>`
	`</tr>`
		`</table>`

Figure 27: Code of the additional files (example 2)

Visualization of a relation A relation that is a result of a query executed in the `Command Panel` will automatically be displayed. When you want to visualize a relation that is already stored in your database, you just have to use the command:

```
query <relationname>
```

You can use the `Object Manager` to switch between different query results by selecting the object you want to display.

Import: Introduction To import data from a *csv* file, a matching relation (dummy) must be available in the database. If this is not the case, you have to create it before importing. The data types must match those in the *csv* file. For example, the code

```
let reltest = [const rel(tuple
               ([Name: string, Studium: string,
                Matrn: int, Note: real, Promotion: bool]))
                value ()]
```

creates an empty relation *reltest*, the first two attributes have the type *string*, the third is an *int*, the fourth attribute has a *real* value and attribute `Promotion` expects the type *bool*. There are three buttons: `Export`, `Import` and `Print`. Before selecting `Import` to import, you must select the relation into which you want to insert data. You can do that with:

```
query reltest
```

The result of that query is the empty relation *reltest*.

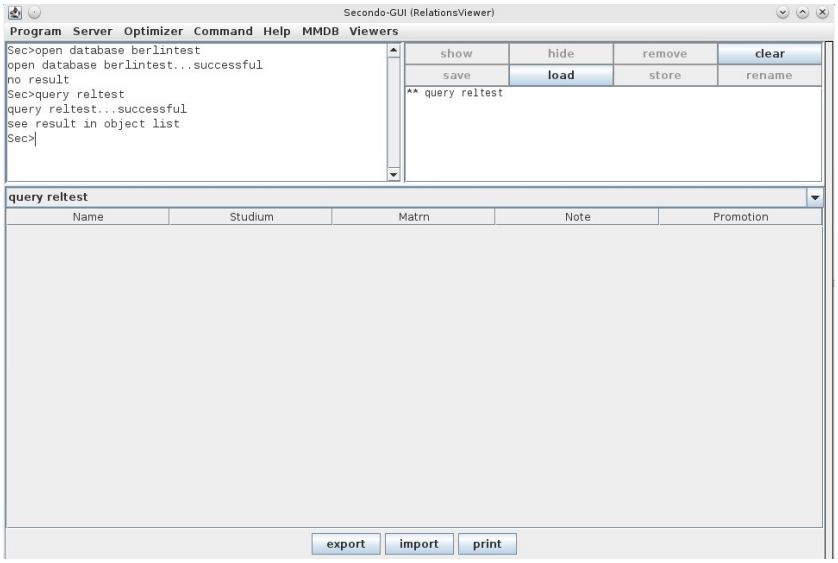

Figure 28: Relation Viewer

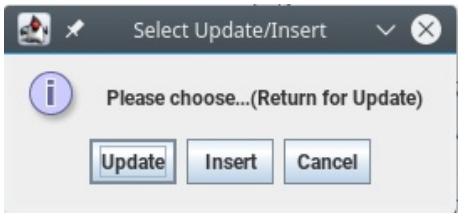

Figure 29: Update Insert Selection

Import: Update and Insert After pressing the `Import` button you will be prompted to select the file. If the file has been selected and `Open` was pressed, another window appears (Figure 29). The selection `update` means that the selected relation is overwritten with the data from the file, so all possible entries in the relation are replaced by the new data. `Insert`, however, appends the new data to existing data in the relation. The old data will be extended with the new data. In our example we choose `Update` first. A request appears to enter the delimiter (see Figure 30). Here, you have to put in exactly the separator, which is used in the *csv* file, otherwise the import will not work.

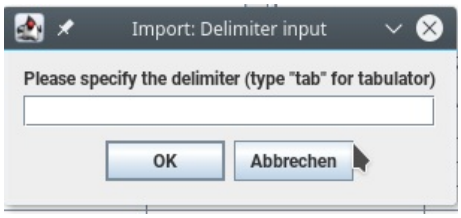

Figure 30: Delimiter Input Request

Next, you can choose to skip lines at the beginning of your import file (Figure 31). The selection `OK`, or the entry 0 followed by `OK` does not skip a line. Otherwise, of course, only as many lines can be skipped as there are in the file. Comment lines at the beginning, initiated with #, are automatically skipped because they do not contain data for the relation.

Now the name of the relation into which data is to be imported has to be confirmed again. In our case *reltest* has to be entered (see Figure 33).

Figure 31: Skip lines

Figure 32: Confirmation

After the execution the added data are shown. To check the data in the relation itself, you can enter `query reltest` again. The relation now contains the data from the file (Figure 21). Next, an `Insert` is executed with the relation *reltest* and a file that contains three more data tuples:

```
Rudolph, Pädagogik, 4243, 1.3, TRUE
Klein, Biologie, 1001, 2.0, FALSE
Meer, Jura, 1282, 1.7, FALSE
```

After entering `query reltest`, you can see that the new data now has been added to the existing data at the end of the relation (see Figure 34).

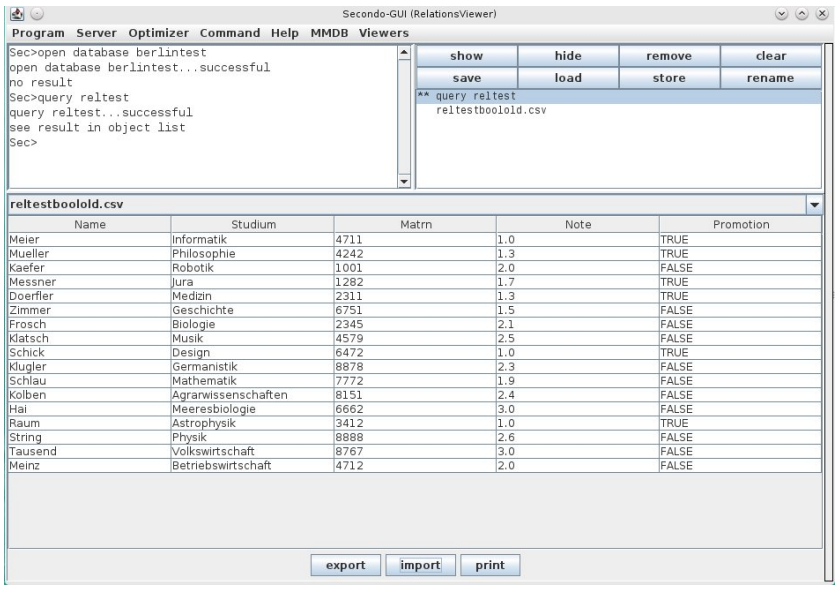

Figure 33: Imported data (update)

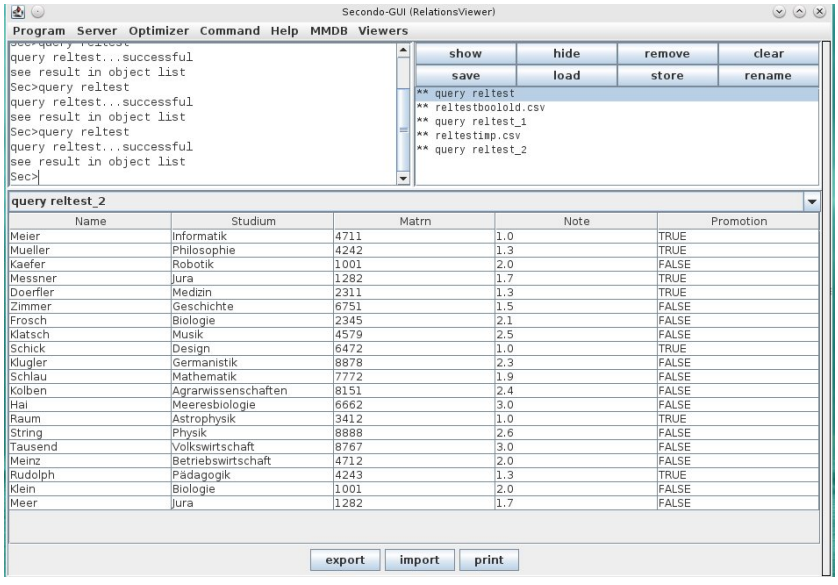

Figure 34: Relation with inserted data

Import: Error messages Finally, we list some common error messages:

"*no table selected*":

This error message is generated if you are trying to import without having previously selected a relation (see above).

"*Table mismatch or delimiter mismatch*";

Here, either the entered delimiter does not match that delimiter in the file, or the schema of the table into which you want to insert data does not match the structure of the data in the file.

"*Insert or update failure-Secondo error*":

For example, this error occurs, if the name of the selected relation does not match the name of the confirmed relation. In particular, this may occur if someone tries to insert data into a relation that does not exist.

Export Select the relation you want to export with `query relname`. After that press the `export` button (see Figure 28). Finally choose a location and a file name and enter a delimiter when you are asked for it.

Printing Just select a relation by querying it and press the print button (see Figure 28). Then a print window will appear. Set the printer parameters according to your wishes and choose a printing device or print to a *ps* file if available in your system.

6.4.5 Other Viewers

Standard Viewer The StandardViewer simply shows a SECONDO object as a string representing the nested list of this object (Figure 35). In the viewer area only one object is displayed at the same time. To show another object in this viewer it must be selected in the `Object Manager` at the top right of this viewer. You can remove the current (or all) object(s) in the extension of the `Menu Bar`. Make sure to load the StandardViewer by default to be able to display any SECONDO object.

FormattedViewer The FormattedViewer shows the results of inquiries sent to SECONDO in a similar way as `SecondoTTY` does (see Figure 36).

InquiryViewer The InquiryViewer shows objects of the same types as a colorized table (Figure 37). In the default configuration file, this viewer is not included. It can be loaded by using by the `gui addViewer` command, for example. Other ways to get this viewer are to insert it into the `gui.cfg` configuration file or to use the `Menu Bar`.

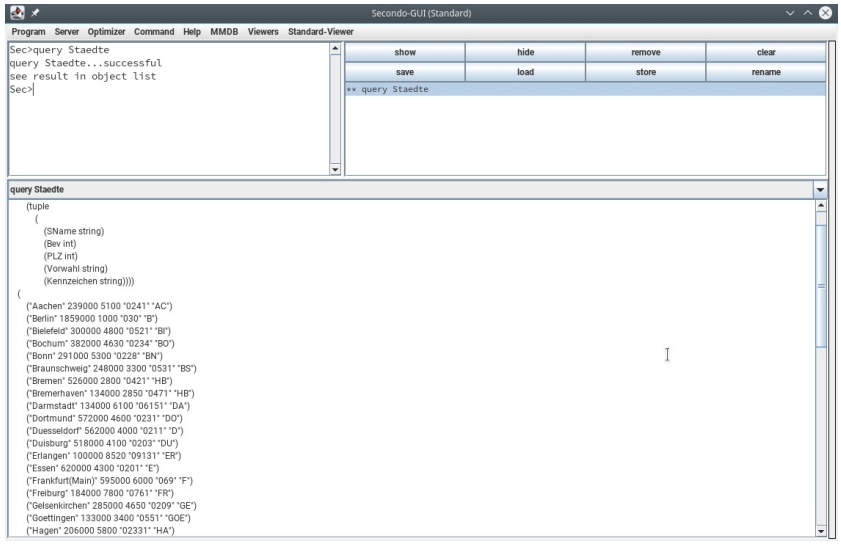

Figure 35: StandardViewer

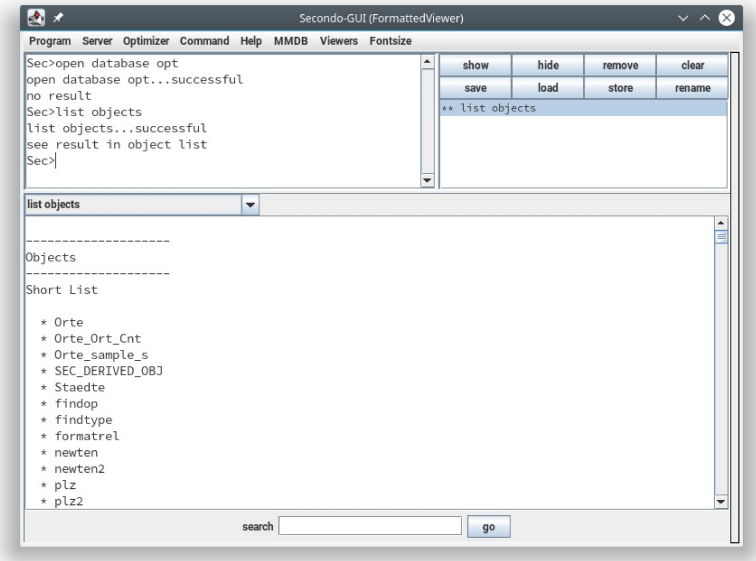

Figure 36: FormattedViewer

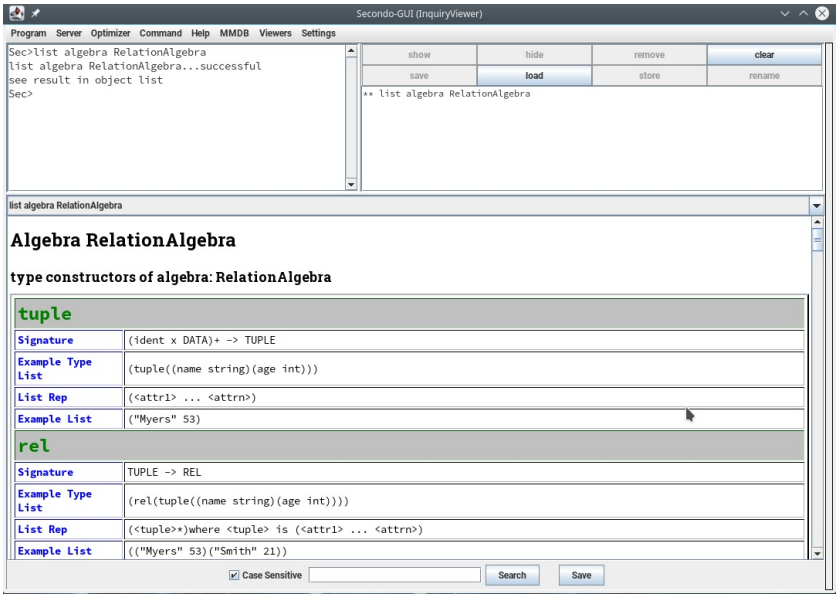

Figure 37: InquiryViewer

There are, in fact, many more viewers for special purposes available; too many to explain them in this introductory user manual. They can be found by selecting the menu item (`Viewers`→`AddViewer`) which will start a dialog to browse through the directory `secondo/Javagui/viewer` containing all the viewer implementations. Some interesting viewers are the following:

- **PictureViewer** Presents jpeg images based on the PictureAlgebra.
- **mp3V** Plays mp3 audio data.
- **PDFRelCreator** Allows one to create relations from directories containg pdf documents or jpeg images.
- **Optics** Presents a diagram resulting from the OPTICS clustering algorithm.
- **SpaceTimeCube** Shows trajectories of moving objects in a 3d representation called the space-time-cube.
- **RTree** Allows one to visualize R-trees.
- **Chess** Visualizes chess games in connection with a chess algebra.
- **V3D** Visualizes moving regions or their units as 3d objects.

7 Customization

SECONDO can be customized in several ways. Most of the preferences affect the extent of functionality and/or the performance of the system. Many other parameters can be adjusted, too.

7.1 Changing the Set of Algebra Modules

As mentioned in Section 1, a running SECONDO system consists of the kernel extended by several algebra modules. Currently, more than 100 different modules are available. They can (almost) arbitrarily be included or excluded when compiling and linking the system by running the `make` utility.

The file `makefile.algebras`, located in the main directory of the SECONDO installation, contains (at least) two lines for every algebra module. The first line references the name of the directory in which the algebra module is present, and the second line defines the name of the algebra module, such as in the excerpt below:

```
ALGEBRA_DIRS += Polygon
ALGEBRAS     += PolygonAlgebra
```

A new algebra module can be activated by inserting the two corresponding lines into the file `makefile.algebras`. If desired, a module can be deactivated by removing or commenting out both lines. However, some of the modules depend on others, so that after the deactivation of an algebra module, another one may not be able to work anymore. In this case, the user will be provided with an error message by the compiler mentioning the unavailable module.

There are modules that require certain libraries to be installed in the underlying operating system. One of them is as follows:

```
ALGEBRA_DIRS += MapMatching
ALGEBRAS     += MapMatchingAlgebra
ALGEBRA_DEPS += xml2
```

Finally, for some algebra modules it is necessary to add a flag in case of deactivation, for example:

```
#ALGEBRA_DIRS += ImageSimilarity
#ALGEBRAS     += ImageSimilarityAlgebra
CCFLAGS       += -DNO_IMAGESIMILARITY
```

In order to activate such a module, uncomment the first two lines and comment out the third line instead.

There are further specifications that are relevant for certain algebras. Linker options that are executed with every linking process can be added with `COMMON_LD_FLAGS`. If the respective option is desired only if algebras are processed (e.g., for building only `SecondoTTYCS`, no algebra is linked), it can be specified with the help of `ALGEBRA_LINK_FLAGS`. Finally, directories containing required header files can be added via `ALGEBRA_INCLUDE_DIRS`. The file `makefile.algebras.sample` provides application examples for these definitons.

After any changes in the `makefile.algebras` file, the SECONDO system has to be recompiled by invoking the `make` command.

7.2 Configuration of Parameters

General parameters affecting the SECONDO system configuration can be altered by editing the file `SecondoConfig.ini` located in the `bin` subdirectory of the SECONDO directory. All application

variants (i.e., `SecondoTTYBDB`, `SecondoPL`, `SecondoTTYCS`, `SecondoMonitor`, and `SecondoPLTTY`, for details please refer to Section 3) read the configuration settings from this file. Optionally, the user may redefine the `SECONDO_CONFIG` environment variable to another file name.

Most of the parameters are described briefly in the file itself, hence we only provide explanations for the most important ones. The first configuration option has the default value

```
SecondoHome=$(HOME)/secondo-databases
```

and defines the location where SECONDO stores its databases. If the specified path does not exist, SECONDO tries to create it. The program execution is aborted if the directory creation cannot be accomplished. In any case, the user should make sure that the storage space available at the database storage path is large enough for the desired data to be processed.

The parameter

```
#RTFlags += SI:UsePasswd
```

is deactivated by default, which means that the SECONDO installation can be started without any authorization procedure. If the user desires to restrict the access to SECONDO, this line has to be uncommented, and a file containing password information has to be specified.

The use of transactions is essential for recovering data that was lost due to a system crash or other unexpected program termination. Consequently, transactions are activated by default. However, particularly for large-scale operations, e.g., importing OpenStreetMap data of Germany or California, the processing times may become unpleasant because large logfiles are written to disk. Transactions can be deactivated by uncommenting the line

```
#RTFlags += SMI:NoTransactions
```

clearly accelerating most operations. Existing logfiles that will not be used anymore are automatocally removed if BerkeleyDB version 4.8 (or later) is applied and the line

```
RTFlags += SMI:AutoRemoveLogs
```

is active. Otherwise, the logfiles can be deleted by executing a script named `rmlogs` located in the `bin` subdirectory of the SECONDO installation.

The global amount of memory available for all SECONDO operations amounts to 512 MBytes by default. The corresponding parameter

```
GlobalMemory=512
```

may be adjusted to any desired value that does not exceed the machine's memory capacity.

7.3 Command Line Parameters

Some of the parameters defined in the `SecondoConfig.ini` file can be overwritten if command line parameters are applied. For example, host and port of a running SECONDO server to be accessed by the current SECONDO client can be altered by invoking the following command in the `bin` subdirectory:

```
SecondoTTYCS -h [IP address] -p [port number]
```

Please execute

```
SecondoTTYBDB --help
```

for a comprehensive list of the supported command line parameters.

References

[DBG09] C. Düntgen, T. Behr, and R. H. Güting. Berlinmod: A benchmark for moving object databases. *VLDB Journal*, 18(6):1335–1368, 2009.

[Güt88] Ralf Hartmut Güting. Geo-relational algebra: A model and query language for geometric database systems. In Joachim W. Schmidt, Stefano Ceri, and Michele Missikoff, editors, *Advances in Database Technology - EDBT'88, Proceedings of the International Conference on Extending Database Technology, Venice, Italy, March 14-18, 1988*, volume 303 of *Lecture Notes in Computer Science*, pages 506–527. Springer, 1988.

[Ore86] Jack A. Orenstein. Spatial query processing in an object-oriented database system. In Carlo Zaniolo, editor, *Proceedings of the 1986 ACM SIGMOD International Conference on Management of Data, Washington, DC, USA, May 28-30, 1986.*, pages 326–336. ACM Press, 1986.