SECONDO: An Extensible DBMS Platform for Research Prototyping and Teaching

Ralf Hartmut Güting, Victor Almeida, Dirk Ansorge, Thomas Behr, Zhiming Ding,
Thomas Höse, Frank Hoffmann, Markus Spiekermann, Ulrich Telle

LG Datenbanksysteme für neue Anwendungen
Fachbereich Informatik, Fernuniversität Hagen
D-58084 Hagen, Germany
rhg@fernuni-hagen.de

Abstract

In this document, we describe in a nutshell the SECONDO extensible DBMS.

1. Introduction

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

SECONDO was intended originally as a platform for implementing and experimenting with new kinds of data models, especially to support spatial, spatio-temporal, and graph database models. We now feel, SECONDO has a clean architecture, and it strikes a reasonable balance between simplicity and sophistication. Since all the source code is accessible and to a large extent comprehensible for students, we believe it is also an excellent tool for teaching database architecture and implementation concepts.

The original idea for SECONDO-like database systems was outlined in [Gü93]. A first version of SECONDO was built between 1995 and 2001. A major reimplementation was started in 2001. The current system uses BerkeleyDB as a storage manager, runs on Windows, Linux, and Solaris platforms, and consists of three major components written in different languages:

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

The three components can be used together or independently, in several ways. All three components can be used together in a configuration shown in Figure 1. In this configuration, the GUI can ask the kernel directly to execute commands and queries (queries written as query plans, i.e., term of the implemented algebras). Or it can call the optimizer to get a plan for a given SQL query. The optimizer when necessary calls the SECONDO kernel to get information about relation schemas, cardinalities of rela-

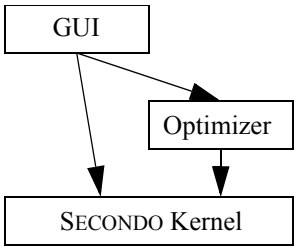


Figure 1: Cooperation of SECONDO Components

tions, and selectivity of predicates. Here the optimizer acts as a server for the GUI and as a client to the kernel. In the following three sections we briefly sketch the three components. A more detailed description of SECONDO can be found in [GBA+00].

2. The SECONDO Kernel

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

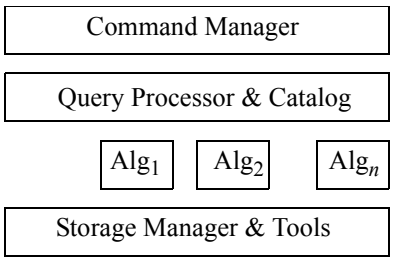


Figure 2: Rough architecture of the kernel

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

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

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

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

The SECONDO kernel manages *databases*. A database is a set of SECONDO objects. A SECONDO object is a triple of the form (*name*, *type*, *value*) where *type* is a type term of the implemented algebras and *value* a value of this type. Databases can be created, deleted, opened, closed, exported to and imported from files. In files they are represented as nested lists (like in LISP) in a text format.

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

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

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

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

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

- StandardAlgebra. Provides data types int, real, bool, string.
- RelationAlgebra. Relations with all operations needed to implement an SQL-like relational language.
- BTreeAlgebra. B-Trees.
- RTreeAlgebra. R-Trees.
- SpatialAlgebra. Spatial data types point, points, line, region.

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

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

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

The following query finds neighbour counties of magdeburg:

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

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

```
windowintersects: rtree(Tuple) x rel(Tuple) x rect -> stream(Tuple)
filter: stream(Tuple) x (Tuple -> bool) -> stream(Tuple)
project: stream(Tuple) x Attrs -> stream(Tuple2)
consume: stream(Tuple) -> rel(Tuple)
bbox: region -> rect
touches: region x region -> bool
contains: string x string -> bool
not: bool -> bool
```

We consider it a major asset of SECONDO that it provides

precise and relatively comfortable notations for query plans like the one shown here. Queries in this notation are completely type-checked by SECONDO. Being able to type query plans interactively is crucial for experimenting with a DBMS. Besides, the notation for query plans is also the interface to the optimizer.

A "live" interaction with SECONDO at the command interface with the query above is shown in Figure 3. It

```
(E) Secondo => query kreis_Gebiet Kreis windowinter-
sects[bbox(magdeburg)] filter[.Gebiet touches magdeburg] fil-
ter[not(.KName contains "Magdeburg")] project[KName] consume
(E) Secondo ->

(query
   (consume
      (project
         (filter
            (filter
               (windowintersects kreis_Gebiet Kreis
                  (bbox magdeburg))
               (fun
                  (tuple1 TUPLE)
                  (touches
                     (attr tuple1 Gebiet)
                     magdeburg)))
            (fun
               (tuple2 TUPLE)
               (not
                  (contains
                     (attr tuple2 KName)
                     "Magdeburg"))))
         (KName))))
Analyze query ...
11:27:11 -> elapsed time 0:00 minutes. Used CPU Time: 0.17 seconds.
Execute ...
11:27:11 -> elapsed time 0:00 minutes. Used CPU Time: 0.22 seconds.

KName: LK Schönebeck

KName: LK Bördekreis

KName: LK Ohre-Kreis

KName: LK Jerichower Land

(E) Secondo =>
```

Figure 3: A query at the command interface

shows how the parser first translates the query into a nested list expression, resolving at the same time some implicit notations for parameter functions.

3. The Optimizer

The optimizer is written in PROLOG, running in the SWI-PROLOG environment which interfaces with C-code. The core functionality is optimization of conjunctive queries. That is, it takes a set of relations and a set of selection and join predicates, and produces a plan.

We have found that PROLOG is an excellent language for implementing query optimizers. The pattern matching and search capabilities of PROLOG make the formulation of optimization rules relatively easy. Execution is very fast. Plans for conjunctive queries with up to 9 predicates are determined in less than a second, from 10 onwards due to the exponential nature of the process, optimization times become noticeable. We feel for an experimental system like SECONDO this is quite sufficient.

The optimizer implements a novel algorithm for query optimization described in [GBA+04]. It uses selectivity estimation and cost estimation to determine good plans.

On top of the conjunctive query optimization capability, we have implemented the essential part of an SQL-like language. The SQL notation was adapted to be able to write queries directly as PROLOG terms. A sample interaction with the optimizer is shown in Figure 4. This is based on a relation Orte containing 506 cities in Germany and a relation plz with 41267 pairs of postal code and city name.

```
2 ?- sql select count(*) from [orte as o, plz as p1, plz as p2] where [o:ort
= p1:ort, p2:plz = p1:plz + 7, (p2:plz mod 5) = 0, p1:plz > 30000, o:ort
contains "o"].
selectivity : 0.000560748
selectivity : 1.6377e-005
selectivity : 0.700935
selectivity : 0.31
Destination node 31 reached at iteration 6
Height of search tree for boundary is 4

The best plan is:

Orte feed {o} filter[(.Ort_o contains "o")] loopjoin[plz_Ort plz exact-
match[.Ort_o] {p1} ] filter[(.PLZ_p1 > 30000)] loopjoin[plz_PLZ plz
exactmatch[(.PLZ_p1 + 7)] {p2} ] filter[((.PLZ_p2 mod 5) = 0)] count

Estimated Cost: 66818.7

Command succeeded, result:

273

Yes
3 ?-
```

Figure 4: Interaction with the optimizer on top of SECONDO

4. User Interface

Besides the command interface, SECONDO has a generic graphical user interface independent of a particular data model. This interface is extensible by viewers. At the bottom left of the next page the GUI is shown. It has a command area (top left), an area for managing the results of queries (top right), and an area for viewers (bottom). Various viewers are available, here a particular viewer is shown. This one, the so-called Hoese-Viewer (named by its author) is able to represent relations with embedded spatial or spatio-temporal objects and offers a rather sophisticated functionality. In itself, it makes a quite interesting demonstration of a user interface for spatial databases. The viewer can also animate moving objects. In the figure, a map of Germany has been created containing cities, rivers, and highways. A query has returned the lines for the river Rhine, and another query has found the counties intersecting the Rhine.

As examples for other viewers, recently three viewers have been written that allow one to manipulate, play and

show data types mp3, jpeg, and midi, for respective algebras.

5. Research Prototyping and Teaching

At this point, it should be obvious that SECONDO is a great platform for experimenting with new data types and even data models. Implementation techniques can be easily tried within the extensible environment. New index structures studied in research can be integrated as an algebra and put to work in a relatively complete DBMS environment. We ourselves use it to study spatial and moving objects databases [CFG+03], network models, fuzzy spatial data types, and optimization techniques.

In addition, we believe SECONDO is an excellent environment for teaching concepts of database systems. It has a clean architecture and an attractive mix of known concepts and implementation techniques and novel features with respect to extensibility. Carefully written user guide, programmer's guide, and installation guide are available.

Of course, SECONDO can be used for writing bachelor and master theses, and has been built in such work to some extent. Recently we have started to offer student projects ("Praktika") for groups of students. The topic of such a project is "Extensible database systems," the duration is one term. We structure a project into two stages: In the first stage, students become familiar with SECONDO by solving a number of exercises:

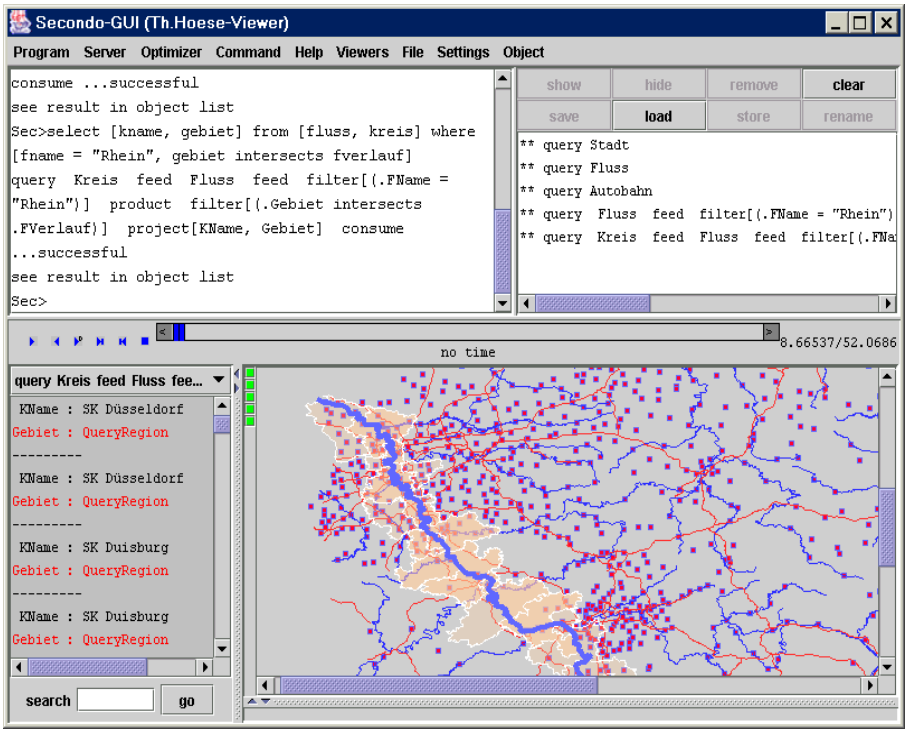
1. Write a small algebra containing data types for point, line segment and triangle with a few operations. Also implement some stream operations.
2. Add a few simple operations to the relational algebra, e.g. duplicate removal by hashing.
3. Learn to manage large objects by implementing a polygon type.
4. Make data types for point, segment, and triangle available as attribute types for relations.
5. An exercise with the storage management interface.
6. Write extensions of the optimizer, for example, rules to use an R-tree and a loopjoin. (This is optional for students with knowledge of PROLOG. Required are programming capabilities in C++ and Java.)
7. Extend the GUI by writing a simple viewer. Also extend the Hoese-Viewer by display classes for point, segment and triangle.

The first stage takes roughly half a term. Here each student works by himself. In the second stage, groups of 3 or 4 students implement together some extension of SECONDO. In the current winter term (2003/04) three groups have been asked to build algebras for midi, mp3, and jpeg data types. We found the outcome rather impressive. All groups have successfully built such algebras, with interesting operations and appropriate viewers/players.

6. What Will be Demonstrated

All the capabilities of SECONDO described in this text can be shown, for example:

- basic commands, writing queries as query plans
- inquiries about the system, e.g. list operators
- using the optimizer
- extensibility: add a rule to the optimizer
- spatial database functionality in the GUI. Managing layers, text-graphics interaction, etc.
- Show animation of a collection of moving points (the underground trains of Berlin).



References

[CFG+03] J. A. Cotelo Lema, L. Forlizzi, R. H. Güting, E. Nardelli, M. Schneider, Algorithms for Moving Objects Databases, *The Computer Journal*, 46(6), 2003.

[DG00] Dieker, S., and R.H. Güting, Plug and Play with Query Algebras: SECONDO. A Generic DBMS Development Environment. Proc. IDEAS 2000, 380-392.

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

[Gü93] R. H. Güting, Second-Order Signature: A Tool for Specifying Data Models, Query Processing, and Optimization. Proc. SIGMOD 1993.