

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, Fernuniversität Hagen
D-58084 Hagen, Germany

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 accommodate 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.

SECONDO runs on Windows, Linux, and Solaris platforms, and consists of three major components *SECONDO kernel*, *optimizer*, and *graphical user interface* described briefly in the following sections. More detailed descriptions can be found in [3, 4].

2. The SECONDO Kernel

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++.

A data model is implemented as a set of data types and operations. These are grouped into *algebras*.

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 [2].

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>      let <ident> = <value expr>
delete type <ident>            delete <ident>
create <ident>: <type expr>     query <value expr>
update <ident> := <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, the available databases, or the contents of the open database. Objects can also be exported into and restored from files. Finally there are commands for transaction management.

Currently there exist about twenty algebras implemented within SECONDO. All algebras include appropriate operations. Some examples are the *StandardAlgebra*, providing data types int, real, bool, string, the *RelationAlgebra*, providing relations with all operations needed to implement an SQL-like relational language, the *R-tree algebra*, or the *SpatialAlgebra*, offering data types point, points, line, region.

The following example query is based on these algebras. It involves some SECONDO objects: (i) a relation County containing the regions of counties in Germany, (ii) an object magdeburg of type region, containing the geometry of county “Magdeburg”, and (iii) an object county_CArea which is an R-tree on the CArea attribute of relation County.

The following query finds neighbour counties of magdeburg:

```

query county_CArea County
windowintersects[bbox(magdeburg)]
filter[CArea touches magdeburg] filter[not(CName contains
"Magdeburg")] project[CName] consume

```

The query uses the R-tree index to find tuples for which the bounding box (MBR) of the CArea 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 CName attribute and the stream is collected into a result relation. Some of 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

```

For each operation a syntax can be specified; for relation and stream operations usually a postfix notation is used. 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.

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.

The optimizer implements a novel algorithm for query optimization described in [3]. 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.

4. User Interface

The graphical user interface (GUI) is an extensible interface for an extensible DBMS such as SECONDO. It is extensible by *viewers* for new data types or models. The GUI is written in Java.

One of the viewers, the so-called Hoeser-Viewer, 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. For a screenshot see [4].

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

SECONDO is a great platform for experimenting with new data types and even data models. For example, new index structures studied in research can be integrated as an algebra. We ourselves use it to study spatial and moving objects databases [1], 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.

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 (optional for students with knowledge of PROLOG).
7. Extend the GUI by writing a simple viewer. Also extend the Hoeser-Viewer by display classes for point, segment and triangle.

The first stage takes roughly half a term. In the second stage, groups of 3 or 4 students implement together some extension of SECONDO. In the winter term 2003/04 three groups have built rather impressive algebras for midi, mp3, and jpeg data types, with interesting operations and appropriate viewers/players.

References

- [1] Cotelma Lema, J. A., L. Forlizzi, R. H. Güting, E. Nardelli, M. Schneider, Algorithms for Moving Objects Databases, *The Computer Journal*, 46(6), 2003, 680-712.
- [2] Dieker, S., and R.H. Güting, Plug and Play with Query Algebras: SECONDO. A Generic DBMS Development Environment. Proc. IDEAS 2000, 380-392.
- [3] 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.
- [4] Güting, R.H. et al., SECONDO: An Extensible DBMS Platform for Research Prototyping and Teaching. Full version of this demo paper, available at <http://www.informatik.fernuni-hagen.de/import/pi4/papers/DemoPaperICDE.pdf>.