SECONDO: An Extensible DBMS Architecture and Prototype*

Ralf Hartmut Güting, Thomas Behr, Victor Almeida, Zhiming Ding,
Frank Hoffmann, Markus Spiekermann

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

Abstract

We describe SECONDO, an extensible DBMS platform suitable for building research prototypes and for teaching architecture and implementation of database systems. It does not have a fixed data model, but is open for implementation of new models. SECONDO consists of three major components which can be used together or independently: (i) the kernel, which offers query processing over a set of implemented algebras, each offering some type constructors and operators, (ii) the optimizer, which implements the essential part of an SQL-like language, and (iii) the graphical user interface which is extensible by viewers for new data types and which provides a sophisticated viewer for spatial and spatio-temporal (moving) objects. Examples of algebras implemented in SECONDO are relations, spatial data types, R-trees, or midi objects (music files), each with suitable operations. The kernel is extensible by algebras, the optimizer by optimization rules and cost functions, and the GUI by viewers and display functions.

A highlight of the description is a new algorithm for conjunctive query optimization which is remarkably simple, yet performs very well. We also emphasize a technique for selectivity estimation suitable for an extensible environment with complex algebras for non-standard data types.

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.

The strategy to achieve this goal is the following:

- Separate the data model independent components and mechanisms in a database system (the *system frame*) from the data model dependent parts.

(*) This work was partially supported by a grant Gu 293/8-1 from the Deutsche Forschungsgemeinschaft (DFG), project “Datenbanken für bewegte Objekte” (Databases for Moving Objects).

- Provide a formalism to describe the implemented data model, in order to be able to provide clean interfaces between system frame and "contents". This formalism is *second-order signature*, explained below.
- Structure the implementation of a data model into a collection of algebra modules, each providing specific data structures and operations.

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. In addition, the central parts are well documented, with a technique developed specifically for this system (the so-called PD system). 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 has been in the making for a long time, and experiences with earlier attempts to implement extensible systems have helped a lot to design the current system. An earlier prototype was the Gral system [Gü89], a relational DBMS extensible by atomic data types and any kind of operations, including relation operations. However, the core data model was fixed. 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 [BDB04] 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 models. 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. The SECONDO kernel can be used as a single user system or in a client-server mode. As a stand-alone system, it can be linked together with either a simple command interface running in a shell, or with the optimizer. In client-server mode, the kernel can serve clients running the command interface, an optimizer client, or the GUI. The optimizer can be used separately to transform SQL-like queries into query plans that would be executable in SECONDO. The GUI can be used separately to browse spatial or spatio-temporal data residing in files. All three components can be used together in a configuration shown in Figure 1.

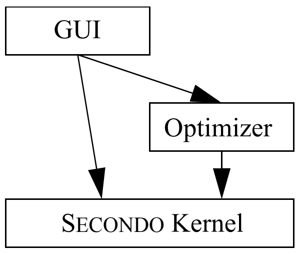


Figure 1: Cooperation of SECONDO Components

In this configuration, the GUI can ask the kernel directly to execute commands and queries (queries written as query plans, i.e., terms 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 relations, 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 describe the three components. In Section 5 we discuss possible uses of SECONDO as a platform for experimental research and for teaching. Related work is considered in Section 6. Conclusions are given in Section 7.

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*.

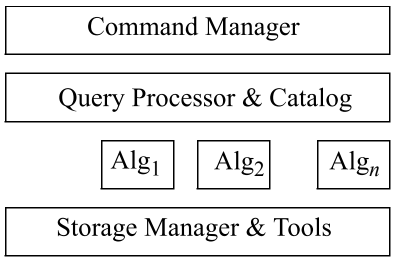


Figure 2: Rough architecture of the kernel

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 [DG00a].

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 appropriate 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.
- DateAlgebra. A small algebra providing a *date* type.
- MidiAlgebra. Providing a data type to represent the contents of Midi files including interesting operations like searching for a particular sequence of keys.

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 which does not try to execute if anything is wrong. 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. The relation "Kreise" has 439 tuples; each contained region has on the average 368 boundary segments. The average tuple size is roughly 50 KByte. The number of segments we computed by a SECONDO query

```
query Kreis feed extend[Segmente: nohalfseg(.Gebiet) / 2] avg[Segmente]
```

```
(E) Secondo => query kreis_Gebiet Kreis windowintersects[bbox(magdeburg)]
filter[.Gebiet touches magdeburg] filter[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

and the tuple size by

```
    query Kreis tuplesize
```

Figure 3 shows how the parser first translates the query into a nested list expression, resolving at the same time some implicit notations for parameter functions. The time for analyzing the query is mainly the time for opening the involved objects.

3 The Optimizer

3.1 Requirements

SECONDO is an extensible DBMS environment for research prototyping and teaching. It should be possible to add algebras without too much effort, in particular algebras with non-standard data types and operations, like a spatial algebra or an algebra for moving objects.

In such an environment, the following requirements for the optimizer arise:

- Like any optimizer, it needs to do its job well, that is, find good query plans fast.
- It should work well also for non-standard data types with lots of operations. In particular, selectivity estimation in such an environment is a big problem.
- The implementation effort for supporting optimization for non-standard data must not be overwhelming; ideally, it would be small.
- Extension mechanisms for the optimizer must be simple and understandable.
- For teaching, it is an advantage if the optimization algorithm is simple and clear, and the code of the optimizer is accessible and well-documented.

The SECONDO optimizer fulfills these requirements.

3.2 Overview

The optimizer is written in PROLOG, running in the SWI-PROLOG environment [SWI04] which interfaces with C-Code. SWI-PROLOG supports the three platforms Windows, Linux and Solaris, like SECONDO.

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 and execution quite fast. To give an idea, plans for conjunctive queries with up to 9 predicates are determined instantly; from 10 onwards, optimization times become noticeable. The PROLOG engine is highly tuned for such work; this optimizer implementation is much faster than anything we tried earlier in C++ (or Modula-2, long ago [BeG92]). We feel for an experimental system like SECONDO this performance is quite sufficient.

The optimizer decomposes the overall problem into three parts, with clearly separated code:

- conjunctive query optimization
- selectivity estimation
- implementing a query language

Currently all of this is formulated for a relational environment. Conjunctive query optimization means the optimizer is given a set of relations and a set of selection and join predicates and produces a plan that can be executed by (the current relational system implemented in) SECONDO. The query language is SQL-like, in a notation adapted to PROLOG so that queries can be entered directly at the PROLOG interface, if desired.

Hence, in contrast to the rest of SECONDO, the optimizer is not data model independent. In particular, queries are limited by the structure of SQL and also the fact that we assume a relational model. On the other hand, the core capability of the optimizer to derive efficient plans for conjunctive queries is needed in any kind of data model.

We will see that the optimizer works strictly as an independent module. It accesses the SECONDO kernel only via its standard interface that can be used by any application. The notation for query plans is the algebra implemented by the kernel; all these plans could be entered manually as well. This means the optimizer module is highly portable to other environments that provide query processing (with a clean interface). As the top level interface, the optimizer provides a predicate `optimize(Query, Plan, Cost)` which can be called from other environments, e.g. the user interface written in Java. Hence, given a query, it returns a plan and a cost estimate.

In the following three subsections we describe (i) a new algorithm for conjunctive query optimization, (ii) our approach to selectivity estimation, and (iii) the user level language.

3.3 A New Optimization Algorithm: Shortest Path Search in a Predicate Order Graph

The optimizer employs an as far as we know novel optimization algorithm which is based on *shortest path search in a predicate order graph*. This technique is remarkably simple to understand and implement, yet efficient.

A predicate order graph (POG) is the graph whose nodes represent sets of evaluated predicates and whose edges represent predicates, containing all possible orders of predicates. Such a graph for three predicates *p*, *q*, and *r* is shown in Figure 4.

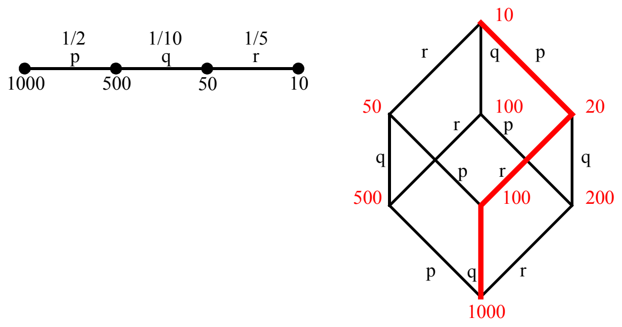


Figure 4: A predicate order graph for 3 predicates

Here the bottom node has no predicate evaluated and the top node has all predicates evaluated. The example illustrates, more precisely, possible sequences of selections on an argument relation of size 1000. One such sequence is shown at the top left of Figure 4. If selectivities of predicates are given (for *p* its is 1/2, for *q* 1/10, and for *r* 1/5), then we can annotate the POG with sizes of intermediate results as shown, assuming that all predicates are independent (not *correlated*). This means that the selectivity of a predicate is the same regardless of the order of evaluation, which of course is not always true.

If we can further compute for each edge of the POG possible evaluation methods, adding a new "executable" edge for each method, and mark the edge with estimated costs for this method, then finding a shortest path through the POG corresponds to finding the cheapest query plan.

Hence the optimization algorithm, due to Güting [Gü02], proceeds in the following steps:

1. For given relations and predicates, construct the predicate order graph and store it as a set of facts in memory.
2. For each edge, construct corresponding executable edges, called *plan edges*. This is controlled by optimization rules describing how selections or joins can be translated.
3. Based on sizes of arguments and selectivities of predicates, compute the sizes of all intermediate results. Also annotate edges of the POG with selectivities.
4. For each plan edge, compute its cost and store it in memory (as a set of facts). This is based on sizes of arguments and the selectivity associated with the edge and on a cost function (predicate) written for each operator that may occur in a query plan.
5. The algorithm for finding shortest paths by Dijkstra is employed to find a shortest path through the graph of plan edges annotated with costs, called *cost edges*. This path is transformed into a SECONDO query plan and returned.

In the sequel, we discuss these steps in more detail.

Step 1: Constructing the POG

The predicate order graph has a very regular structure and can be constructed easily by a recursive algorithm. The recursion is illustrated in Figure 5 which shows POGs for 0, 1, 2, and 3 predicates.

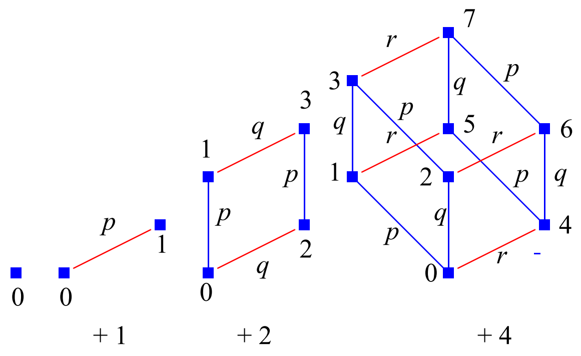


Figure 5: Recursive construction of the POG

We employ a node numbering scheme that encodes in a node number the predicates that have been evaluated in this node. Every bit of a node number corresponds to one predicate. If there are *n* predicates, they are numbered from 0 to n-1. Hence for node 5, binary 101, the predicates with numbers 0 and 2 have been evaluated.

A basic algorithm for constructing the POG for a list of predicates $p_0, ..., p_m$ can be formulated as follows:

```
algorithm pog-simple
    1. The POG for an empty list of predicates consists of a single node labeled 0.
    2. The POG for predicates $p_0, ..., p_m$ is the following:
        a. let $G$ be the POG for predicates $p_0, ..., p_{m-1}$;
        b. let $G'$ be a copy of $G$ where $2^m$ is added to all node numbers;
        c. let $E$ be a set of edges created as follows: for each node of $G$ labeled $j$, create an edge
           labeled $p_m$ from node $j$ to node $j + 2^m$ in $G'$;
        d. the result graph is $G \cup E \cup G'$.
end pog-simple.
```

A few simple facts about the predicate order graph: A POG for n predicates represents all possible orders of the predicates, hence has n! paths from the start node 0 to the destination node 2^n-1. The graph has 2^n nodes and $n \cdot 2^{n-1}$ edges.

The basic algorithm above needs to be extended a bit to obtain a complete solution for step 1 of the optimization algorithm. We need to keep track of

- which relations have been joined already, and
- which node of the POG represents the result of a predicate application.

This is explained further in the sequel.

A conjunctive query can be represented as a *query graph*, whose nodes are relations, edges between distinct nodes represent join predicates, and an edge connecting a node with itself represents a selection. Two examples of query graphs are shown in Figure 6. Both involve two relations A and B. The

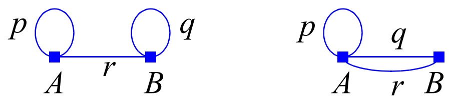


Figure 6: Two query graphs

graph on the left has selection p on A, selection q on B, and a join r between A and B. The graph on the right has one selection on A and two join conditions between A and B. We generally assume the query graph is connected, hence the result is a single relation.

When selections and joins are applied in some order, originally distinct relations are combined into intermediate results. After evaluating all predicates we have a single relation.

We can associate with each node of the POG a *partition* of the set of query relations Q into disjoint subsets, noting for each subset which predicates have been applied. For example, for the graph in Figure 6 (left) some partitions are:

$n_0 = n_\emptyset$: $(\{A\}, \emptyset), (\{B\}, \emptyset)$
$n_1 = n_{\{p\}}$:$(\{A\}, \{p\}), (\{B\}, \emptyset)$
$n_3 = n_{\{p, q\}}$:$(\{A\}, \{p\}), (\{B\}, \{q\})$
$n_5 = n_{\{p, r\}}$:$(\{A, B\}, \{p, r\})$

Furthermore, we associate each element of a partition with a node of the POG where this intermediate result can be found. Let (*Rels*, *Preds*) be an element of a partition. Then the node representing this result is n_{Preds}. Encoding the set *Preds* into a node number, we extend partition elements by this number and obtain for the example shown above:

$n_0 = n_\emptyset$: $(0, \{A\}, \emptyset), (0, \{B\}, \emptyset)$
$n_1 = n_{\{p\}}$:$(1, \{A\}, \{p\}), (0, \{B\}, \emptyset)$
$n_3 = n_{\{p, q\}}$:$(1, \{A\}, \{p\}), (2, \{B\}, \{q\})$
$n_5 = n_{\{p, r\}}$:$(5, \{A, B\}, \{p, r\})$

We now extend the algorithm *pog-simple* as follows: Predicates are represented as terms *pr*(*p*, *a*) for a selection predicate *p* on relation *a*, and *pr*(*p*, *a*, *b*) for a join predicate *p* on relations *a* and *b*. Parameters for the algorithm now called *pog* are a list of relations and a list of predicates. Hence it may be called as

```
pog([a, b], [pr(p, a), pr(q, b), pr(r, a, b)]).
```

Step 1 is extended by associating with node 0 the initial partition which has all relations separate, with no predicates evaluated.

In step 2b a copy of graph *G* is made. Nodes are copied as follows. Besides modifying node numbers, partitions associated with nodes need to be copied and adapted to the predicate p_m. This means:

- For a selection predicate *pr*(*p*, *a*) find the partition element containing *a* and add the predicate *p* to its set of predicates.
- For a join predicate *pr*(*p*, *a*, *b*): If *a* and *b* are found in distinct partition elements, then merge these, forming the unions of relations and predicates and adding *p* to the predicates. If *a* and *b* are found in the same partition element, then just add *p* to its predicates.
- Copy the remaining partition elements unchanged.

Edges are copied in step 2b by recreating them from the new start node. Creation of edges is now based on the partitions in the start node, see the modified step 2c below. It is necessary to recreate edges because a join may switch into a selection, if the involved relations are joined in predicate p_m. For example, this would occur for predicate *q* in Figure 6 (right) when predicate *r* is processed.

In step 2c new edges are created for the current predicate p_m, from node *j* to node $j + 2^m$. Edges are represented as terms *edge*(*Source*, *Target*, *Term*, *Result*) where *Result* is the node where the intermediate result after applying this predicate can be found, and *Term* is created according to these rules:

- For a selection predicate *pr*(*p*, *a*) create an edge with term *select*(*Arg*, *pr*(*p*, *a*)). How arguments like *Arg* are determined is eplained below.
- For a join predicate *pr*(*p*, *a*, *b*): If *a* and *b* are in distinct partition elements of node *j*, then create an edge with term *join*(*Arg1*, *Arg2*, *pr*(*p*, *a*, *b*)). If *a* and *b* are found in the same partition element of node *j*, then create an edge with term *select*(*Arg*, *pr*(*p*, *a*, *b*)).

The arguments written into the *select* or *join* terms are of the form either *arg*(*X*) or *res*(*X*). If the partition element found has a node number 0, hence the form (0, {*A*}, ∅), then no predicate has yet been evaluated and the argument is *arg*(*k*) where *k* is the number of the original argument relation. If the node number is $l > 0$, then we can use an intermediate result associated with node *l* and the term is *res*(*l*).

We illustrate the steps of the optimization algorithm for the example query:

```
select o:ort from [orte as o, plz as p] where [o:ort = p:ort, p:plz = 44225].
```

This is based on a relation `Orte` containing 506 cities in Germany and a relation `plz` with 41267 pairs of city name `Ort` and postal code `PLZ`. Hence the query asks for the city with postal code 44225.

After the first step we can see the constructed POG by commands `writeNodes` and `writeEdges`.

```
4 ?- writeNodes.
Node: 0
Preds: []
Partition: [arp(arg(2), [rel(plz, p, l)], []), arp(arg(1), [rel(orte, o,
u)], [])]

Node: 1
Preds: [pr(attr(o:ort, 1, u)=attr(p:ort, 2, u), rel(orte, o, u), rel(plz, p,
l))]
Partition: [arp(res(1), [rel(orte, o, u), rel(plz, p, l)], [attr(o:ort, 1,
u)=attr(p:ort, 2, u)])]

Node: 2
Preds: [pr(attr(p:pLZ, 1, u)=44225, rel(plz, p, l))]
Partition: [arp(res(2), [rel(plz, p, l)], [attr(p:pLZ, 1, u)=44225]),
arp(arg(1), [rel(orte, o, u)], [])]

Node: 3
Preds: [pr(attr(p:pLZ, 1, u)=44225, rel(plz, p, l)), pr(attr(o:ort, 1,
u)=attr(p:ort, 2, u), rel(orte, o, u), rel(plz, p, l))]
Partition: [arp(res(3), [rel(orte, o, u), rel(plz, p, l)], [attr(p:pLZ, 1,
u)=44225, attr(o:ort, 1, u)=attr(p:ort, 2, u)])]
```

Here a term `rel(orte, o, u)` denotes a relation with an associated variable `o` to be written in upper case, hence as `Orte`. In PROLOG all atoms need to start lower case, since otherwise they would be taken for variables. Therefore we need to encode relation names in this way. The notation `attr(p:pLZ, 1, u)` denotes an attribute of the relation with variable `p` of the first argument relation, also to be written in upper case. These notations are created from the query in a translation step before constructing the POG. Partition elements are denoted as `arp(Arg, Rels, Preds)` terms.

```
5 ?- writeEdges.
Source: 0
Target: 1
Term: join(arg(1), arg(2), pr(attr(o:ort, 1, u)=attr(p:ort, 2, u), rel(orte,
o, u), rel(plz, p, l)))
Result: 1

Source: 0
Target: 2
Term: select(arg(2), pr(attr(p:pLZ, 1, u)=44225, rel(plz, p, l)))
Result: 2

Source: 1
Target: 3
Term: select(res(1), pr(attr(p:pLZ, 1, u)=44225, rel(plz, p, l)))
Result: 3

Source: 2
Target: 3
Term: join(arg(1), res(2), pr(attr(o:ort, 1, u)=attr(p:ort, 2, u), rel(orte,
o, u), rel(plz, p, l)))
Result: 3
```

Step 2: Constructing Plan Edges

Plan edges are constructed from `select` or `join` edges of the POG based on translation rules. A simple rule to translate a selection is:

```
select(Arg, pr(Pred, _)) => filter(ArgS, Pred) :- Arg => ArgS.
```

This says that a term `select(Arg, pr(Pred, _))` can be translated to a term `filter(ArgS, Pred)` if the argument `Arg` can be translated to a term `ArgS` (denoting an argument stream). Here `filter` is a SECONDO operator; these are written in the usual PROLOG prefix notation in these rules. Three rules to translate a join into either a `sortmergejoin` or a `hashjoin` are shown next.

```
join(Arg1, Arg2, pr(X=Y, R1, R2)) => JoinPlan :-
  X = attr(_, _, _),
  Y = attr(_, _, _), !,
  Arg1 => Arg1S,
  Arg2 => Arg2S,
  join00(Arg1S, Arg2S, pr(X=Y, R1, R2)) => JoinPlan.

join00(Arg1S, Arg2S, pr(X = Y, _, _)) => sortmergejoin(Arg1S, Arg2S, attr-
name(Attr1), attrname(Attr2))   :-
  isOfFirst(Attr1, X, Y),
  isOfSecond(Attr2, X, Y).

join00(Arg1S, Arg2S, pr(X = Y, _, _)) => hashjoin(Arg1S, Arg2S, attr-
name(Attr1), attrname(Attr2), 997)   :-
  isOfFirst(Attr1, X, Y),
  isOfSecond(Attr2, X, Y).
```

The plan edges constructed for our example are shown in Figure 7 where already edge costs and estimated result sizes have been added.

```
7 ?- writeCostEdges.
Source: 0
Target: 1
Plan: Orte  feed {o}  plz  feed {p}
product  filter[(.Ort_o = .Ort_p)]
Result: 1
Size: 11709
Cost: 6.14278e+007

Source: 0
Target: 1
Plan: Orte  feed {o}  loopjoin[plz_Ort
plz  exactmatch[.Ort_o] {p} ]
Result: 1
Size: 11709
Cost: 119020

Source: 0
Target: 1
Plan: Orte  feed {o} plz  feed {p} sort-
mergejoin[Ort_o, Ort_p]
Result: 1
Size: 11709
Cost: 161953

Source: 0
Target: 1
Plan: Orte  feed {o} plz  feed {p} hash-
join[Ort_o, Ort_p, 997]
Result: 1
Size: 11709
Cost: 124068

Source: 0
Target: 2
Plan: plz  feed {p}  filter[(.PLZ_p =
44225)]
Result: 2
Size: 96.4182
Cost: 89962.1
```

```
Source: 0
Target: 2
Plan: plz_PLZ plz  exactmatch[44225] {p}
Result: 2
Size: 96.4182
Cost: 973.824

Source: 1
Target: 3
Plan: res(1)  filter[(.PLZ_p = 44225)]
Result: 3
Size: 27.3576
Cost: 19671.2

Source: 2
Target: 3
Plan: Orte  feed {o}  res(2)  product
filter[(.Ort_o = .Ort_p)]
Result: 3
Size: 27.3576
Cost: 143727

Source: 2
Target: 3
Plan: Orte  feed {o} res(2) sortmerge-
join[Ort_o, Ort_p]
Result: 3
Size: 27.3576
Cost: 1350.91

Source: 2
Target: 3
Plan: Orte  feed {o} res(2) hash-
join[Ort_o, Ort_p, 997]
Result: 3
Size: 27.3576
Cost: 2743.31
```

Figure 7: Cost edges for the example

Step 3: Assigning Sizes

Since the sizes of the arguments and all selectivities are known, it is easy to assign expected sizes to all nodes of the POG. Selectivities come from the component described in Section 3.4.

Step 4: Assigning Edge Costs

For each plan edge its cost is computed and a corresponding cost edge is stored in memory. The computation is based on the sizes of the arguments, assigned in the previous step, on the selectivity of the edge, and on cost functions for each operator that may occur in a plan for the egde.

A cost function is a predicate of the form

```
cost(Term, Sel, Size, Cost) :-
```

which means that the cost of an executable `Term` representing a predicate with selectivity `Sel` is `Cost` and the size of the result is `Size`. Cost is computed by descending recursively into the `Term`, computing the costs for the subterms.

Note that a term does not necessarily consist of only a single operator. For example, a join can be translated into a `product - filter` sequence. However, we assume there is exactly one operator which realizes the predicate. The cost function for this operator can use the `Sel` argument to distinguish the size of the stream before and after the operator.

Step 5: Computing the Shortest Path on Cost Edges and Converting It to a Query Plan

We now have a graph with weighted edges. Hence we are ready to apply the algorithm by Dijkstra to find the shortest path from the start node 0 to the destination node 2^n-1 (for n predicates). A call for our example is shown next.

```
13 ?- dijkstra(0, 3, Path, Cost), plan(Path, Plan), plan_to_atom(Plan, Sec-
ondoQuery).
Destination node 3 reached at iteration 2
Height of search tree for boundary is 1

Path = [costEdge(0, 2, rename(exactmatch(plz_PLZ, rel(plz, p, l), 44225),
p), 2, 96.4182, 973.824), costEdge(2, 3, sortmergejoin(rename(feed(rel(orte,
o, u)), o), res(2), attrname(attr(o:ort, 1, u)), attrname(attr(p:ort, 2,
u))), 3, 27.3576, 1350.91)]
Cost = 2324.73
Plan = sortmergejoin(rename(feed(rel(orte, o, u)), o), rename(exact-
match(plz_PLZ, rel(plz, p, l), 44225), p), attrname(attr(o:ort, 1, u)),
attrname(attr(p:ort, 2, u)))
SecondoQuery = 'Orte  feed {o} plz_PLZ plz  exactmatch[44225] {p} sortmerge-
join[Ort_o, Ort_p] '
```

The call `dijkstra(0, 3, Path, Cost)` computes the shortest path `Path` from node 0 to node 3 at cost `Cost`. The predicate `plan(Path, Plan)` connects the two edges 0-2 and 2-3 of the plan into a single term `Plan`. Finally, `plan_to_atom(Plan, SecondoQuery)` translates the `Plan` into a query in SECONDO syntax.

It is very interesting to observe that the algorithm by Dijkstra does not need to visit the entire POG. It stops as soon as the destination node is reached. Hence often a large part of the search space can be pruned away. There are some examples built into the optimizer code with 8, 9, and 10 predicates, respectively, for which we find the numbers of visited nodes shown in Table 1. For example, the

number of preds	*number of nodes*	*number of visited nodes*
8	256	18
9	512	104
10	1024	109

Table 1: Pruning effects of shortest path search

query with 10 predicates is:

```
example17 :- optimize(
select * from [staedte, plz as p1, plz as p2, plz as p3]
where [
    sname = p1:ort, p1:plz = p2:plz + 1, p2:plz = p3:plz * 5,
  bev > 300000, bev < 500000, p2:plz > 50000,
  p2:plz < 60000, kennzeichen starts "W",
  p3:ort contains "burg", p3:ort starts "M"]
  ).
```

In our running example, algorithm Dijkstra stops after visiting two nodes `(Destination node 3 reached at iteration 2)`. See also the example in Section 3.5 where it stops after visiting 6 out of 32 nodes. This behaviour is typical.

3.4 Selectivity Estimation

Cost estimation, even for non-standard applications to be handled by SECONDO, is manageable as long as the sizes of all arguments are known. We know which algorithms are used in plan operators, and can model their costs. The hard problem is selectivity estimation.

The three main approaches in the literature are parametric methods [Ch83, CR94], non-parametric methods (histograms) [PC84, APR99], and sampling [HOT89, LNS90, WAA02]. Sampling has been used for various purposes, e.g., to build approximate histograms, to construct a random subset of a query answer [OR86], to estimate the results of aggregate queries [HOT89], and also to perform selectivity estimation as such [LNS90]. Among these, histograms are the most popular technique.

In an environment like SECONDO, parametric methods do not seem applicable. Histogram techniques work well for standard data types. But observe that the standard types are very simple and essentially only the operations = and ≤ are supported by histograms.

For complex algebras over non-standard data types, e.g. an algebra for moving objects with lots of operations [GBE+00, CFG+03] histograms might be used in principle. However, an enormous research effort to develop new types of histograms and to study their behaviour for estimating selectivity for various operations would be needed. Indeed, this effort is much larger than that of designing the algebra itself and implementing its operations. Whereas this is an interesting topic for research, waiting for the results and implementing them is an extreme impediment in the development of extensible systems.

We have concluded that sampling is the only generally feasible alternative. Our experience with this technique implemented in SECONDO leads us to fully agree with the statements from [LNS90, p. 2]:

> *"The main point is that, unless the query itself can be computed extremely efficiently (...), or the answer is very small, the size of the query can be estimated accurately in a small fraction of the time it takes to compute the query."*
> *"The results presented in this paper argue that size estimation through sampling could be easily added to database systems and can provide what is perhaps surprisingly good performance."*

We now describe how sampling actually works in SECONDO. Three main ideas are:

- materialized samples,
- samples in memory,
- stored selectivities.

Materialized Samples. For each relation in a database, at least when it is used for the first time in a query to be optimized, SECONDO creates and stores a small sample which is a random subset of the tuples of that relation. For a relation named `X`, the sample is named `X_sample`. Samples can be created using an operator `sample` with signature:

```
sample: rel(Tuple) x int x real -> stream(Tuple)
```

Arguments are a relation, a number of tuples m, and a fraction f of tuples desired. The sample will have size $\min(n, \max(m, f \cdot n))$ where n is the size of the relation. For example, the SECONDO command

```
let plz_sample = plz sample[100, 0.01] consume
```

creates a sample of size 412 because relation plz has 41267 tuples.

Samples can be created either manually or automatically by the optimizer. The optimizer uses an existing sample if it is available, otherwise creates one on the fly by sending a command to the SECONDO kernel. Samples are built quite fast, in a single scan of the relation.

The optimizer uses the default policy shown above, i.e., a sample is a fraction of 1% of the relation, but has at least 100 tuples (except if the relation has less than 100 tuples). Of course, the literature on sampling derives in detail what is the optimal sample size to achieve some precision with some confidence. For us at the moment the ad-hoc parameters above seem sufficient.

Samples in Memory. Like a histogram data structure, a sample relation is kept in memory. More precisely, it is loaded into memory when it is needed for the first selectivity query involving this relation. For large relations, this may incur a small delay at this time. To avoid this, one could easily add a "warm-start" feature which would load all samples into memory when opening a database.

Keeping relations in memory and performing queries on them is particularly easy in SECONDO, as the relational algebra was implemented from the beginning with two different storage models. In the memory model basically a relation is a vector of tuples; a tuple is a vector of pointers to attribute values. In the persistent model, a relation is a file and a tuple a record of the storage manager. There is a common interface to both implementations so that operators are usually implemented only once, for both models. A few operators need to have separate implementations for the two models.

Stored Selectivities. Selectivities of predicates can be stored hard-coded as facts in PROLOG, for example:

```
sel(plz:ort = staedte:sName, 0.0031).
sel(plz:pLZ = (plz:pLZ)*5, 0.0000022).
sel(plz:pLZ < 60000, 0.64).
sel((plz:pLZ mod 5) = 0, 0.17).
sel(plz:ort contains "burg", 0.060).
```

This facility is very useful for developing the optimizer and testing and was available long before anything about samples and selectivity queries to SECONDO was implemented. If the optimizer finds such a fact when trying to determine the selectivity of a predicate, fine.

When the optimizer is started, it reads a file with stored selectivities, like the one shown next, and stores the contents as facts in memory.

```
/* Automatically generated file, do not edit by hand. */
storedSel(fluss:fName= "Rhein", 0.05).
storedSel(kreis:gebiet intersects fluss:fVerlauf, 0.0058).
storedSel(aktive:matNr=zaehlbar:matNr, 0.0004).
storedSel(autobahn:aName= "A 45", 0.03).
storedSel(thousand:no>995, 0.011236).
storedSel(thousand:no>900, 0.134831).
storedSel(thousand:no>200, 0.786517).
storedSel(plz:ort contains "dorf", 0.0630841).
storedSel(plz:ort contains "Klein", 0.00700935).
```

Note that PROLOG automatically indexes facts by their first argument value, so access is quite efficient. Hence if the optimizer finds such a stored (dynamic) fact, it also returns.

Otherwise, it sends a selectivity query to the SECONDO kernel which evaluates the selectivity on the samples by either a scan (for a selection) or a loopjoin (for a join). The resulting selectivity is also stored as a fact in memory. When the optimizer is shut down at the end of a session, it stores these facts back into the file.

The time required for a selectivity query depends on whether it is a selection or a join, the size of the involved relations (resp. samples), and whether the predicate evaluation is cheap (for standard types) or more expensive (e.g. for spatial types). Times for selection queries on standard types are only a fraction of a second, for joins on larger relations the time needed may get into the range of a few seconds. See also the example evaluations in Section 3.5. Of course, if really expensive predicates are evaluated, times may get unpleasantly long. There exist two effects which together make the time requirements acceptable:

- Selection predicates involve constants, hence there exist many different selection predicates of interest. Therefore it is not so likely that the selectivity is found in the stored selectivities. But the time requirements for selection selectivity queries are low.
- The time requirements for join selectivity queries are higher. But join predicates compare two attributes and only a few such comparisons make sense in practice. Hence after working with a database for some time, all relevant join selectivities will be stored, hence there is no time required any more for such queries.

Regarding the quality of the selectivity estimation, we made a small evaluation for the predicates that we happended to find in the file `storedSels` shown above at the time of writing. The result is shown in Table 2. Whereas this is not a thorough experimental study, it seems that the results are of reasonable quality to support optimization decisions.

Predicate	*Estim. sel.*	*Res. size*	*Size arg1*	*Size arg2*	*Real sel.*
fluss:fname = "Rhein"	0,05	17	375		0,0453
kreis:gebiet intersects fluss:fverlauf	0,0058	1025	439	375	0,0062
aktive:matnr = zaehlbar:matnr	0,0004	4940	2241	6206	0,00035
autobahn:aname = "A 45"	0,03	10	325		0,0307
thousand:no > 995	0,0112	5	1000		0,005
thousand:no > 900	0,1348	100	1000		0,1
thousand:no > 200	0,7865	800	1000		0,8
plz:ort contains "dorf"	0,0630	2284	41267		0,0553
plz:ort contains "Klein"	0,0070	200	41267		0,0048

Table 2: Estimated and real selectivities

Note that the algorithm for conjunctive query optimization is entirely independent from the used technique for selectivity estimation. The big advantange of the sampling technique is that it works automatically for all data types and we get it basically for free. As soon as we can process a query, we can also process the corresponding selectivity query. No additional effort is required to implement histogram techniques. However, nothing prevents one from additionally implementing histograms for those data types for which they are available, e.g. standard data types, and certain spatial types and operations. Such implementations can easily be added in the PROLOG environment which automatically tries existing alternatives. Hence, if it can derive a selectivity from a histogram, it will do so, otherwise fall back to the sampling technique.

3.5 Query Language

The third component of the optimizer is an implementation of the core of an SQL-like language, adapted to PROLOG notation so that a query can be written directly as a PROLOG term. This means that lists of attributes, relations, and predicates are written comma-separated in square brackets.

Because the "." has a special meaning in PROLOG, we use a colon instead to separate tuple variables and attribute names, hence instead of writing `e.name` we write `e:name`.

A sample interaction with the optimizer is shown in Figure 8. This is based on a relation `Orte` containing 506 cities in Germany and a relation `plz` with 41267 pairs of city name `Ort` and postal code `PLZ`.

```
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  exact-
match[(.PLZ_p1 + 7)] {p2} ]  filter[((.PLZ_p2 mod 5) = 0)]  count

Estimated Cost: 66818.7

Command succeeded, result:

273

Yes
3 ?-
```

Figure 8: Interaction with the optimizer on top of SECONDO

The optimizer writes some information about its working. One can see that it sends 4 selectivity queries to SECONDO for which the resulting selectivities are printed. For the third predicate the selectivity was known already. Note that some of the predicates are (deliberately, of course) of a kind that can hardly be supported by histograms.

One can also observe that in the shortest path search through the predicate order graph only 6 out of the $32 = 2^5$ nodes need to be visited. The plan uses exact match queries on B-tree indexes on the `Ort` and `PLZ` attributes of the postal code relation `plz`.

To give some idea about performance: The query of Figure 8 was just run on the author's computer[1] after starting SECONDO and ensuring that none of the needed selectivities are stored (except the hard-coded one for the third predicate). The total time for processing the query was 4.29 seconds which is divided as shown in Table 3. Milliseconds were rounded to the next 10 msec. Time measurements are elapsed time received from the PROLOG environment.

1. This is a some years old standard PC running Windows ME, not a very fast machine (900 MHz AMD Processor).

Action	*Time [sec]*
selectivity query o:ort = p1:ort	0.440
selectivity query p2:plz = p1:plz + 7	1.380
selectivity query p1:plz > 30000	0.050
selectivity query o:ort contains “o”	0.060
optimization	0.060
plan execution	2.300

Table 3: Running times for query of Figure 8

Running the query a second and third time resulted in running times of 1.98 and 1.92 seconds, respectively. Since selectivities are stored now, only the times for optimization and plan execution remain. Plan execution is a bit faster than the first time, probably due to caching effects in the storage manager.

Recently the implementation of the query language has been extended beyond the basic select-from-where statement. In the current version also groupby and orderby clauses are available, it is possible to create new attributes in the result based on expressions in the select-clause, and union and intersection of several select-from-where clauses are available.

3.6 Discussion

In this section we briefly discuss some properties, inherent, and current limitations of the optimization algorithm.

The algorithm does not manage properties of intermediate results such as *interesting orders*. Keeping track of the order of a tuple stream would allow one to use a *mergejoin* algorithm instead of a *sortmergejoin*, hence to omit sorting if the arguments are sorted already. We have considered to add such a feature to the POG (having for each node a collection of “minor” nodes with different properties), but found the additional complication not worthwhile. It would also slow down optimization as the graph would become much larger.

The algorithm considers all orders among predicates that have no order dependency, such as 5 selection predicates on 5 different relations. We feel the simple overall structure and the capability to find plans by shortest path search compensates for that.

The algorithm does not handle Cartesian products, as some other methods do.

Currently no projections are done on base relations; this is a feature that could easily be added. It is especially relevant for tuples containing large objects as attributes.

Translation rules are applied to each edge of the POG independently. So the same predicate is translated many times. One could reduce this to a few cases distinguishing whether one of the arguments is a base relation, and then use a shared version of plan edges. This would speed up the algorithm and save space so that larger numbers of predicates could be handled.

Currently cost functions are a bit simplistic. They do not model and distinguish the cost of evaluating predicates. At the moment they also do not consider tuple sizes, but this is a feature that will shortly be added. These are not inherent problems, just a matter of continuing the work. The same holds for yet missing optimization rules, e.g. for using spatial indexes.

The criteria for determining sample sizes are also simplistic. For larger relations, e.g. a million tuples, we do not need 1% of the relation size. It might also make sense to maintain for each relation two samples of different sizes, one for evaluating selection and a smaller one for evaluating join selectivity queries.

On the positive side again, the algorithm is guaranteed to find the optimal plan if the following conditions hold:

- Cartesian products are not needed,
- all relevant optimization rules are implemented,
- cost functions are sufficiently precise,
- predicates are independent (not correlated).

That means, no parts of the search space are incorrectly pruned away. In particular, the algorithm also creates bushy plans.

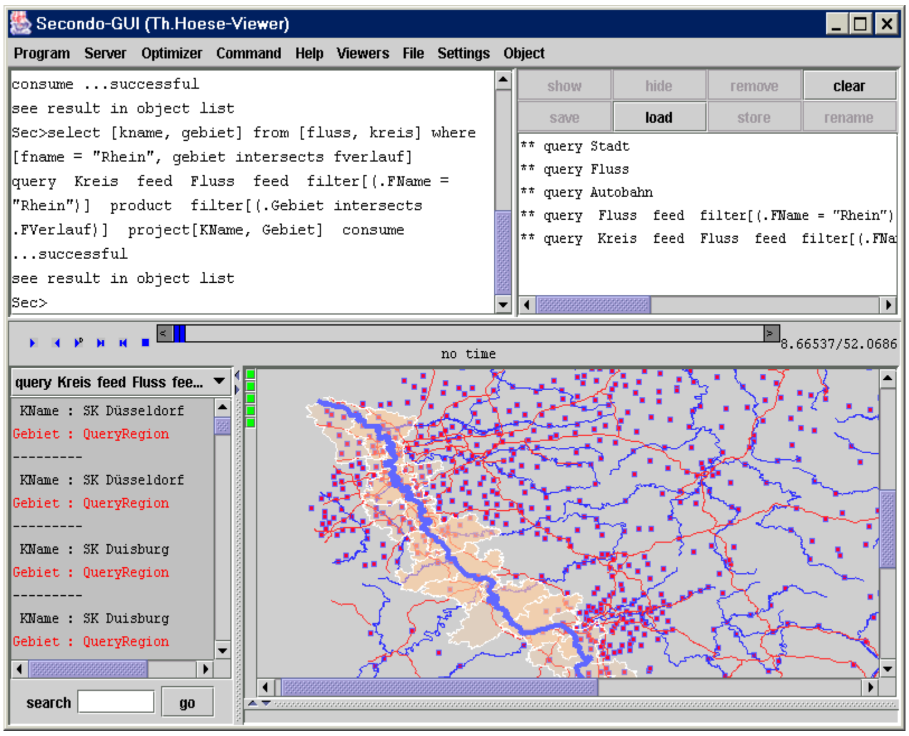
4 User Interface

Besides the command interface, SECONDO has a generic graphical user interface independent of a particular data model. This interface is completely written in Java. For this reason it can be used on many different platforms. It communicates with SECONDO via TCP/IP using nested lists for transferring objects.

The GUI can also be used offline, independently from SECONDO, as a browser for the data types representable by one or more of its viewers. In this case objects are loaded from files, where they are given in the nested list representation of SECONDO. Some other formats are supported, see below.

The interface uses viewers to display objects. It is extensible in two ways. The first possibility is to add a new viewer in the framework. The second one is to extend a viewer itself.

In Figure 9 the GUI is shown. It has a command area (top left), an object manager (top right), and an area for viewers (bottom). The main purpose of the command area is to enter queries for SECONDO. But it can also be used for controlling the GUI itself by prefixing the keyword `gui` to a command. The available commands can be found in the `Help` menu. Furthermore the GUI can communicate with an optional optimizer server. If a query starts with `select` or `sql`, the optimizer is used to find the best plan for this SQL query. After that, this plan is sent to SECONDO for execution. A prefixed keyword `optimizer` provides the facility to send arbitrary commands in PROLOG syntax to the optimizer server.


Secondo-GUI (Th.Hoese-Viewer)
Program Server Optimizer Command Help Viewers File Settings Object
show hide remove clear
save load store rename
8.66537/52.0686
no time
query Kreis feed Fluss fee...
search
go

```
consume ...successful
see result in object list
Sec>select [kname, gebiet] from [fluss, kreis] where
[fname = "Rhein", gebiet intersects fverlauf]
query  Kreis  feed  Fluss  feed  filter[(.FName =
"Rhein")]  product  filter[(.Gebiet intersects
.FVerlauf)]  project[KName, Gebiet]  consume
...successful
see result in object list
Sec>
```

```
** query Stadt
** query Fluss
** query Autobahn
** query  Fluss  feed  filter[(.FName = "Rhein")
** query  Kreis  feed  Fluss  feed  filter[(.FNa
```

```
KName : SK Düsseldorf
Gebiet : QueryRegion
---------
KName : SK Düsseldorf
Gebiet : QueryRegion
---------
KName : SK Duisburg
Gebiet : QueryRegion
---------
KName : SK Duisburg
Gebiet : QueryRegion
```

Figure 9: The graphical user interface

The object manager contains all results of successful finished SECONDO queries as well as objects loaded from files. The contained objects can be removed from the current viewer (`hide`), removed completely (`remove`, `clear`), added to a viewer (`show`), saved into a file (`save`) or stored into the currently opened database (`store`). The interface can import several file formats (`load`), like shape files, dbase III files or files containing nested lists. With the aid of the storing mechanism we can create new database objects from such files.

In the current version of the interface more than ten viewers exist. They allow one to display a large set of different data types. Especially all types defined in the currently existing algebras can be represented. By the concept of viewers it is possible to show an object in different ways. Examples of available viewers are:

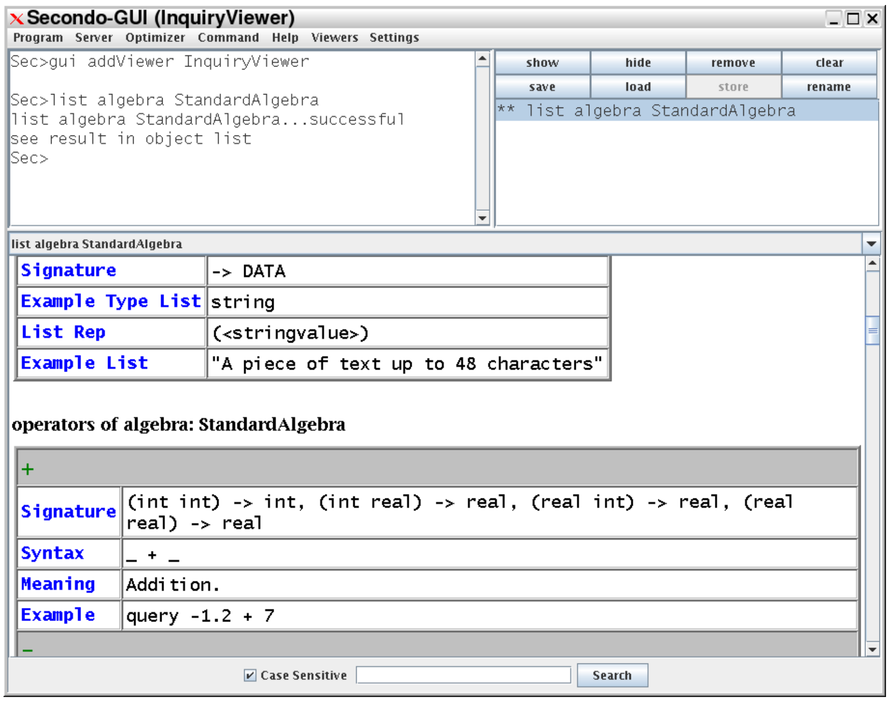
- HoeseViewer, described later in detail
- InquiryViewer: displays results of inquiries to SECONDO in a formatted manner
- RelViewer: displays relations containing textual information
- Fuzzy2D, Viewer3D: display fuzzy spatial objects in different ways
- JPEGViewer, MP3Viewer and MidiViewer: show (play) the corresponding data type and additional information
- StandardViewer: shows simply the nested list representation of an object as text (possible for any object).

When the GUI is started, a set of frequently used viewers (given in a configuration file) is loaded. But a new viewer can also be loaded at runtime.

In Figure 9 a particular viewer is shown. This one, the so-called HoeseViewer (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. The HoeseViewer itself can be extended for displaying new object types containing textual, geographical or temporal information. To do that, just a display class implementing the interface `DsplBase`, `DsplGraph` and `Timed`, respectively, has to be written. It is not necessary to change the source code of the HoeseViewer. A display class can be added at runtime. It will be found automatically. There one only needs to describe how a given object should be displayed. All other features of the HoeseViewer are for free. Such features are, for example:

- zooming, scrolling, searching text
- setting rendering attributes, e.g. colour, transparency
- selecting: when a graphical object is selected (by a mouse click), the corresponding textual representation is also marked and vice versa

The HoeseViewer is well suited to display the data types described above, but not all objects are representable in this viewer. For example, the representation of data types like midi or mp3 is not possible. The missing control of the representation does not allow one to show objects with three dimensions because the order of painting is important. For such objects new viewers have to be designed. In Figure 10 the InquiryViewer filled with the result of the inquiry `list algebra StandardAlgebra` is shown. Writing a new viewer is not very difficult. A viewer has to provide methods


Secondo-GUI (InquiryViewer)
Program Server Optimizer Command Help Viewers Settings
show hide remove clear
save load store rename
list algebra StandardAlgebra
Case Sensitive
Search

```
Sec>gui addViewer InquiryViewer

Sec>list algebra StandardAlgebra
list algebra StandardAlgebra...successful
see result in object list
Sec>
```

** list algebra StandardAlgebra

Signature	-> DATA
Example Type List	string
List Rep	(<stringvalue>)
Example List	"A piece of text up to 48 characters"

operators of algebra: StandardAlgebra

+	
Signature	(int int) -> int, (int real) -> real, (real int) -> real, (real real) -> real
Syntax	_ + _
Meaning	Addition.
Example	query -1.2 + 7
–	

Figure 10: Another viewer in the user interface

for adding, removing and selecting `SecondoObjects`. An object is given by its nested list representation together with a name. Furthermore, a viewer implementor has to write methods which return whether and how well a given object can be displayed. Optionally a viewer writer can extend the main menu of the user interface with own menu entries for controlling the viewer. For example, in Figure 10 the `Settings` entry comes from the InquiryViewer. When a new object is added, the viewer analyzes the nested list representation and converts it into its own data structure, which can be displayed. Here the decision about when, where and how the object is drawn is left to the implementor. This can be done in a few lines of code (e.g. in the StandardViewer) or it can require a lot of classes with complex computations (e.g. in the Viewer3D for fuzzy objects). In all cases the source code of the framework remains untouched.

Another viewer, the QueryViewer, can help to develop a new one. The QueryViewer can process all relations coming from SECONDO. If an object is not a relation, it is wrapped into one. From a relation can be extracted single objects, all objects of a given attribute, tuples and the whole relation. The extracted objects are added to another viewer selected as a subviewer. So, when a viewer can display a certain data type, this type can also be used in an environment using relations containing such objects.

Because this interface is extensible, it fulfills the requirements of SECONDO in an excellent way. When an algebra adds new data types to SECONDO, the user interface should be extended to display objects of these types.

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 [GBE+00, 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. The source code is, at least in the essential parts, well documented. 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. Exercises are roughly the following:

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 Related Work

SECONDO continues a long tradition of extensible systems that have been studied since the mid-eighties. There are many important and interesting prototypes of which we can mention only a few here, namely GENESIS [BBG+88], EXODUS [CDG+90], Starburst [HCL+90], Postgres [SRH90], Volcano [Gr94], Predator [Se98], and CONCERT [BRS96]. Our own group has developed the Gral system [BeG92]. Some of these systems are data-model independent and favor a toolkit approach for building database systems, including components like a storage manager and an optimizer generator (e.g. GENESIS, EXODUS, Volcano). Others provide a fixed, for example, an object-relational, data model with well-defined interfaces for extensions such as data types or indexes (e.g. Starburst, Postgres, Gral). More recently, commercial systems like Oracle, DB2 provide interfaces for collections of data types with operations called data blades, cartridges, or extenders. These are similar to SECONDO's algebra modules. However, such data type additions do not change the core data model of the respective system. Some unique aspects of SECONDO are

- the existence of a meta model, second-order signature, to define a data model and algebra over it, which allows one to build clean interfaces,
- the concept of algebra modules which covers the entire implementation of the data model, not only extensions for attribute data types,
- a clean and precise notation for describing data types and especially query plans, including the possibility to enter type-checked plans at the user interface.

The main emphasis of this paper has been on query optimization in SECONDO. Classical work on query optimization [SAC79] has focused on determining a good join order by a dynamic programming algorithm, evaluating selections first, according to the classical heuristics. Later in [HS93, H94] it was observed that modern applications may use *expensive predicates*, for example, a check whether an image attribute contains a sunset, and techniques were developed to insert such expensive predicates into the query plan at a later stage. Subsequent work treated selections and joins equally, as we do here. In particular, the paper [SM98] has influenced us a lot, where all possible arrangements of selections and joins in a query tree were considered, again by dynamic programming. The work in [SM98] is based on techniques for rapid enumeration of join orders in [VM96]. Other work that tries to derive an optimal plan in the presence of expensive predicates is [CS96].

To our knowledge, all of the previous work on query optimization tries to enumerate systematically the possible query trees. Our approach to represent all possible orders of selections and joins in a graph and then perform optimization by shortest path search on this graph seems to be entirely new and quite promising.

Approaches to selectivity estimation have already been discussed in Section 3.4. The ideas of materialized samples, keeping samples in memory, and storing selectivities may be new.

Regarding the user interface, we are not aware of a published description of a generic interface for an extensible DBMS that would be extensible like our GUI. Requirements for spatial user interfaces are discussed in [Eg94].

More details about the SECONDO kernel system can be found in the following papers: [DG00a] discusses the extensible architecture (however, the concepts described there for supporting query optimization such as models and model mappings are outdated now). [GDF+99] presents the query processor in detail, and [DG00b] describes the concept for large object management in SECONDO.

7 Conclusions

The goal in writing the paper has been twofold: (i) to present SECONDO as a complete extensible system including kernel, optimizer, and user interface, and (ii) to describe in detail our novel algorithm for query optimization using shortest path search in a predicate order graph.

We feel the SECONDO system as a whole is a nice platform for experimenting with DBMS implementations. It is not only possible, but even relatively easy to integrate, for example, a new kind of index structure, or a collection of data types and operations for new applications. The core data model is not exempt from such changes, as in many other systems, but can be modified as well. For example, an algebra for nested relations has recently been implemented as a master thesis project. To make this possible, it is important that the system is still simple enough to be accessible for students. In designing SECONDO, we have tried to maintain a good balance between simplicity and sophistication when necessary for efficient execution.

The algorithm for query optimization is, to our knowledge, quite new and a radical departure from previous techniques. It is remarkably simple to implement, but fast and produces good plans. Together with the concept for selectivity estimation by sampling it works quite well for an extensible system like SECONDO.

Future work on the query optimizer includes

- a systematic experimental evaluation investigating aspects such as the running times for varying numbers of predicates, dependencies on the number of available rules or the complexity of cost functions, pruning effects of shortest path search, quality of plans,
- completing the set of optimization rules, especially for spatial and spatio-temporal applications,
- providing more precise cost functions, especially including the cost of predicate evaluation,
- implementing some of the improvements mentioned in Section 3.6.

Our future work on SECONDO as a whole will mostly focus on implementing algebras and indexes for new applications. We are currently implementing a data model for objects moving in networks, as described in [GAD04], using the index from [AG04]. This model requires one to change the core data model in order to accomodate networks.

Acknowledgments

We thank Helge Bals, Thomas Höse, and Ulrich Telle for their contributions to developing and implementing the SECONDO system. The work of Stefan Dieker, Miguel Rodriguez de Luaces, Jose Antonio Cotelo Lema, and Nicole Richter on the first version of SECONDO (developed until 2001) is also gratefully acknowledged.

References

[AG04] Almeida, V.T. de, and R.H. Güting, Indexing the Trajectories of Moving Objects in Networks. Fernuniversität Hagen, Informatik-Report 309, 2004.

[APR99] Acharya, S., V. Poosala, and S. Ramaswamy, Selectivity Estimation in Spatial Databases. Proc. ACM SIGMOD, 1999, 13-24.

[BBG+88] Batory, D.S., J.R. Barnett, J.F. Garza, K.P. Smith, K. Tsukuda, B.C. Twichell, and T.E. Wise, GENESIS: An Extensible Database Management System. *IEEE Trans. on Software Engineering,* 14, 1988, 1711-1730.

[BDB04] BerkeleyDB. http://www.sleepycat.com/

[BeG92] Becker, L, and R.H. Güting, Rule-Based Optimization and Query Processing in an Extensible Geometric Database System. *ACM Trans. on Database Systems*, 17(2), 1992, 247-303.

[BRS96] Blott, S., L. Relly, and H.J. Schek, An Open Storage System fr Abstract Objects. Proc. ACM SIGMOD Conf., 1996, 330-340.

[CDG+90] Carey, M.J., D.J. DeWitt, G. Graefe, D.M. Haight, J.E. Richardson, D.T. Schuh, E.J. Shekita, and S.L. Vandenberg. The EXODUS Extensible DBMS Project: An Overview. In: S.B. Zdonik and D. Maier (eds.), Readings in Object-Oriented Database Systems, 474-499. Morgan-Kaufmann Publishers, 1990.

[CFG+03] Cotelo Lema, J.A., L. Forlizzi, R. H. Güting, E. Nardelli, and M. Schneider, Algorithms for Moving Objects Databases, *The Computer Journal*, 46(6), 2003, 680-712.

[Ch83] Christodoulakis, S., Estimating Record Selectivities. *Information Systems*, 8(2), 1983, 69-79.

[CR94] Chen, C.M., and N. Roussopoulos, Adaptive Selectivity Estimation Using Query Feedback. Proc. ACM SIGMOD, 1994, 161-172.

[CS96] Chaudhuri, S., and K. Shim, Optimization of Queries with User-Defined Predicates. Proc. VLDB, 1996, 87-98.

[DG00a] Dieker, S., and R.H. Güting, Plug and Play with Query Algebras: SECONDO. A Generic DBMS Development Environment. Proc. Int. Database Engineering and Applications Symposium (IDEAS, Yokohama, Japan), 2000, 380-392.

[DG00b] Dieker, S., and R.H. Güting, Efficient Handling of Tuples with Embedded Large Objects. *Data & Knowledge Engineering*, 32, 2000, 247-269.

[Eg94] Egenhofer, M., Spatial SQL: A Query and Presentation Language. *IEEE Trans. on Knowledge and Data Engineering*, 6(1), 1994, 86-95.

[GAD04] Güting, R.H., V.T. de Almeida, and Z. Ding, Modeling and Querying Moving Objects in Networks. Fernuniversität Hagen, Informatik-Report 308, 2004.

[GBE+00] Güting, R.H., M. H. Böhlen, M. Erwig, C. S. Jensen, N. A. Lorentzos, M. Schneider, and M. Vazirgiannis, A Foundation for Representing and Querying Moving Objects. *ACM Trans. on Database Systems*, 25(1), 2000, 1-42.

[GDF+99] Güting, R.H., S. Dieker, C. Freundorfer, L. Becker, and H. Schenk, SECONDO/QP: Implementation of a Generic Query Processor. 10th Int. Conf. on Database and Expert Systems Applications (Florence, Italy), 1999, 66-87.

[Gr94] Graefe, G., Volcano – An Extensible and Parallel Query Evaluation System. *IEEE Trans. on Knowledge and Data Engineering*, 6, 1994, 120-135.

[Gü02] Güting, R.H., A Query Optimizer for SECONDO. Source Code and Documentation for the SECONDO Optimizer, from December 2002 on, file `optimizer.pl` within SECONDO.

[Gü89] Güting, R.H., Gral: An Extensible Relational Database System for Geometric Applications. Proc. of the 15th Intl. Conf. on Very Large Data Bases, Amsterdam, 1989, 33-44.

[Gü93] Güting, R.H., Second-Order Signature: A Tool for Specifying Data Models, Query Processing, and Optimization. In: Proc. ACM SIGMOD Conference. Washington, USA, 1993, 277-286.

[H94] Hellerstein, J.M., Practical Predicate Placement. Proc. ACM SIGMOD, 1994, 325-335.

[HCL+90] Haas, L.M., W. Chang, G.M. Lohman, J. McPherson, P.F. Wilms, G. Lapis, B. Lindsay, H. Pirahesh, M. Carey, and E. Shekita, Starburst Mid-Flight: As the Dust Clears. *IEEE Transactions on Knowledge and Data Engineering*, 2, 1990, 143-160.

[HOT89] Hou, W.C., G. Ozsoyoglu, and B. Taneja, Processing Aggregate Relational Queries with Hard Time Constraints. Proc. ACM SIGMOD, 1989, 68-77.

[HS93] Hellerstein, J.M., and M. Stonebraker, Predicate Migration: Optimizing Queries with Expensive Predicates. Proc. ACM SIGMOD, 1993, 267-277.

[LNS90] Lipton, R.J., J.F. Naughton, and D.A. Schneider, Practical Selectivity Estimation through Adaptive Sampling, Proc. ACM SIGMOD, 1990, 1-11.

[OR86] Simple Random Sampling for Relational Databases. Proc. VLDB, 1986, 160-169.

[PC84] Piatetsky-Shapiro, G., and C. Connell, Accurate Estimation of the Number of Tuples Satisfying a Condition. Proc. ACM SIGMOD, 1984, 256-276.

[SAC79] Selinger, P.G., M.M. Astrahan, D.D. Chamberlin, R.A. Lorie, and T.G. Price. Access Path Selection in a Relational Database System. Proc. ACM SIGMOD, 1979, 23-34.

[Se98] Seshadri, P., Enhanced Abstract Data Types in Object-Relational Databases. *VLDB Journal*, 7(3), 1998, 130-140.

[SM98] Scheufele, W., and G. Moerkotte, Efficient Dynamic Programming Algorithms for Ordering Expensive Joins and Selections. Proc. EDBT, 1998, 201-215.

[SRH90] Stonebraker, M., L.A. Rowe, and M. Hirohama, The Implementation of POSTGRES. *IEEE Transactions on Knowledge and Data Engineering*, 2, 1990, 125-142.

[SWI04] SWI-Prolog. http://www.swi-prolog.org/

[VM96] Vance, B., and D. Maier, Rapid Bushy Join-Order Optimization with Cartesian Products. Proc. ACM SIGMOD, 1996, 35-46.

[WAA02] Wu, Y.L, D. Agrawal, and A.E. Abbadi, Query Estimation by Adaptive Sampling. Proc. ICDE, 2002, 639-648.