Plug and Play with Query Algebras: SECONDO A Generic DBMS Development Environment[*]

Stefan Dieker and Ralf Hartmut Güting
Praktische Informatik IV, FernUniversität Hagen
D-58084 Hagen, Germany
{stefan.dieker, gueting}@fernuni-hagen.de

Abstract

We present SECONDO, a new generic environment supporting the implementation of database systems for a wide range of data models and query languages. On the one hand, this framework is more flexible than common extensible and object-relational systems, offering the full extensibility of *second-order signature*, the formal basis for data and query language definitions in SECONDO. On the other hand, it is much more complete and structured than database system toolkits. Extensibility is provided by the concept of *algebra modules* defining and implementing new types (type constructors, in fact) and operators. Support functions are used to register them with the system frame.

After a review of second-order signature essentials, this paper presents the system functionality, given by a uniform set of user commands valid for all data models, and the extensible system architecture. All common DBMS features are implemented in the system frame; only purely data model dependent functionality is coded in algebra modules, supported by a variety of tools. Furthermore, we describe the key strategies for extensible query processing in the SECONDO environment and explain the structure of algebra modules.

1 Introduction

Conventional relational database systems cannot meet the requirements from modern database application domains like CAD, GIS, spatio-temporal information systems, or the large and still growing field of multimedia processing. As a consequence, new application-specific data models arise. Common facilities for the development of systems implementing new data models are toolkits like EXODUS [CaDF+86] or extensible systems as developed in the Postgres [StR86] project.

While toolkits are very generic and can be used for the implementation of many different data models, they leave too much expenditure at the implementor to be accepted as a suitable means for fast system implementation. Extensible systems, on the other hand, pre-implement as much functionality as possible at the expense of flexibility, since they usually prescribe a specific data model, e.g. an object-relational model.

The goal of SECONDO is to offer a generic "database system frame" that can be filled with implementations of a wide range of data models, including, for example, relational, object-oriented, graph-oriented or

* This work was partially supported by the CHOROCHRONOS project, funded by the EU under the Training and Mobility of Researchers Programme, Contract No. ERB FMRX-CT96-0056.

sequence-oriented DB models. The strategy to achieve this goal is to separate the data-model independent components and mechanisms in a DBMS (the *system frame*) from the data-model dependent parts. Nevertheless, the frame and the "contents" have to work together closely. With respect to the different levels of query languages in a DBMS, we have to describe to the system frame:

- the *descriptive algebra*, defining a data model and query language,
- the *executable algebra,* specifying a collection of data structures and operations capable of representing the data model and implementing the query language,
- *rules* to enable a query optimizer to map descriptive algebra terms to executable algebra terms, also called *query plans* or *evaluation plans.*

A general formalism serving all of these purposes has been developed earlier, called *second-order signature (SOS)* [Güt93]. It is reviewed in Section 3.

On top of the descriptive algebra level may be some syntactically sugared language, e.g. in an SQL-like style. We assume that the top-level language and the descriptive algebra are entirely equivalent in "expressive power"; only the former may be more user-friendly whereas the latter is structured according to the SOS formalism. A compiler transforming the top-level language to descriptive algebra can be written relatively easily using compiler generation tools, since it just has to perform a one-to-one mapping to the corresponding data definitions and operations.

At the system level, definitions and implementations of type constructors and operators of the executable algebra are arranged into *algebra modules*, interacting with the system frame through a small number of well-defined support functions for manipulation of types and objects as well as operator invocation. Those algebra support functions dealing with type expressions will be created automatically from the corresponding SOS specification.

The remainder of this document is structured as follows. Section 2 discusses related work. In Section 3 we focus on how second-order signature allows one to define a descriptive or executable algebra. Section 4 presents the system functionality, the extensible architecture, and the key techniques implemented for query processing. Section 5 explains the structure of algebra modules and shows how inter-module interaction can be organized. In Section 6 we briefly discuss the implementation of user interfaces on top of SECONDO, and Section 7 concludes the paper.

2 Related Work

The first approach to open up the type system of a relational DBMS with abstract data types (ADTs) was pioneered by the Ingres project [OnFS84]. It was followed by Postgres [StR86], primarily focusing on query optimization with ADTs and support for complex objects. Another important project implementing an extensible relational system was Starburst [ScCF+86], with query processing and a clean architecture for storage and indexing of complex objects as key goals. More recently, also commercial extensible systems are available, now known as *object-relational* systems [CaD96]. Informix Universal Server [Inf98] is just one example.

Other object-relational systems provided extensibility of the data model, but emphasized a particular application field. Quite early, for instance, the need for extensibility was observed in the area of geographical information systems. Projects range from the Gral system [Güt89], emphasizing structured data model extension by algebraic specifications, to the Paradise project [PaYK+97], whose main research interest is increasing the efficiency of query evaluation by parallelization, motivated by the huge amount of geographical data available through modern satellite systems.

So far we exclusively considered systems which, though extensible, are still tightly bound to the relational model. A much more radical approach to support the development of application-specific non-standard database systems can be identified as the database system toolkits/components approach [CaD96]. Toolkits do not prescribe any data model, but rather identify a common subset of functionality which all database systems for whatever data model must provide, e.g. transaction management, concurrency control, recovery, and query optimization. Key projects of that category were GENESIS [BaBG+88] and EXODUS [CaDF+86]. The storage manager component of SHORE [CaDF+94], the successor of EXODUS, in fact is used as the storage manager for SECONDO.

Both the object-relational and the toolkits thread of research meet at SECONDO. Since SECONDO is not bound to any data model, it is more extensible than the presented extensible systems. On the other hand, it offers a precise framework to describe varying data models and execution systems which is reflected in well-defined interfaces for registering corresponding support functions. Thus, SECONDO is as comfortable to extend as any object-relational system and almost as flexible as a database toolkit: an extensible algebraic term execution engine on top of a state-of-the-art storage manager with support for persistent objects.

Volcano [Gra94] and PREDATOR [SeLR97] are two other systems which cannot be classified according to the above categories, but are closer in spirit to SECONDO. Volcano is currently a more complete system than SECONDO, including mechanisms for parallelization and the Volcano optimizer generator [GrM93]. SECONDO, on the other hand, is superior to Volcano in terms of generic query processing, query algebra specification, and simplicity and clarity of algebra module implementation.

PREDATOR advocates the concept of enhanced ADTs (E-ADTs) for data model extension. An E-ADT is an ADT with additional information on the semantics of the ADT, supporting query optimization. In contrast to common object-relational systems, the relational functionality is not hard-coded, but rather provided by an exchangeable relational E-ADT. A sequence database system has been implemented [SeLR96]. Compared to SECONDO, PREDATOR is even more generic since it is not restricted to the (very small) limitations of the expressive power of second-order signature. As a consequence, however, E-ADT implementation for non-standard data models is a much more complex issue than in SECONDO, since query processing details have to be implemented by the E-ADT implementor.

3 Second-Order Signature

Since the SECONDO system tries to implement the framework of second-order signature (SOS) developed in [Güt93], it is necessary to recall the essential concepts here. The basic idea of SOS is to use two coupled signatures to describe first, a data model and second, an algebra over that data model. A signature in general has *sorts* and *operators* and defines a set of *terms*.

3.1 Specifying a Descriptive Algebra

In SOS, the first signature has so-called *kinds* as sorts and *type constructors* as operators. The terms of the first signature are called *types*. In the sequel we show example specifications for the relational model and a relational execution system. Although the purpose of SECONDO is not to reimplement relational systems, it makes no sense to explain an unknown formalism using examples from an unknown data model. The structural part of the relational model can be described by the following signature:

kinds IDENT, DATA, TUPLE, REL
type constructors

	$\rightarrow$ DATA	$\underline{int}$, $\underline{real}$, $\underline{string}$, $\underline{bool}$
$(\text{IDENT} \times \text{DATA})^+$	$\rightarrow$ TUPLE	$\underline{tuple}$
TUPLE	$\rightarrow$ REL	$\underline{rel}$

Here $\underline{int}$, $\underline{real}$, $\underline{string}$, and $\underline{bool}$ are type constructors without arguments, or *constant* type constructors, of a result kind called DATA. A kind stands for the set of types (terms) for which it is the result kind. For DATA this set is finite, namely DATA = $\{\underline{int}, \underline{real}, \underline{string}, \underline{bool}\}$. In contrast, there are infinitely many types of kind TUPLE or REL. For example,

$\underline{tuple}$([(name, $\underline{string}$), (age, $\underline{int}$)])
$\underline{rel}$($\underline{tuple}$([(name, $\underline{string}$), (age, $\underline{int}$)]))

are types of kind TUPLE and REL, respectively. The definition of the $\underline{tuple}$ type constructor uses a few simple extensions of the basic concept of signature that are present in the SOS framework, for example, if $s_1, \ldots, s_n$ are sorts, then $(s_1 \times \ldots \times s_n)$ is also a sort (*product* sort), and if s is a sort, then s^+ is a sort (*list* sort). The term $(t_1, \ldots, t_n)$ belongs to a product sort $(s_1 \times \ldots \times s_n)$ iff each t_i is a term of sort s_i; the term $[t_1, \ldots, t_m]$, for $m \geq 1$, is a term of sort s^+ iff each t_i is a term of sort s. The kind IDENT is predefined (and treated in a special way in the implementation in SECONDO); its type constructors are drawn from some infinite domain of "identifiers"; hence they can be used as attribute names here.

The notion of a relation *schema* has been replaced by a relation type, and "relation" is not considered to be a single type, but a type constructor. Hence operations like selection or join are viewed as polymorphic operations. Note that the choice of kinds and type constructors is completely left to the designer of a data model. We are not offering a toolbox with a fixed set of constructors such as *tuple, list, set*, etc., but instead a framework where new constructors can be defined. Hence the SECONDO system frame as such knows nothing about $\underline{rel}$ or $\underline{tuple}$ constructors.[1] To demonstrate that this is a general framework, let us briefly show a complex object type system (similar to [BaK86]):

1. The only predefined kinds and type constructors in SECONDO are IDENT with its "type constructors" and the type constructor $\underline{stream}$ which plays a special role in query evaluation (see Section 4.2.3).

kinds IDENT, OBJ

type constructors

	$\to$ OBJ	$\underline{\textit{bottom}}$, $\underline{\textit{top}}$, $\underline{\textit{int}}$, $\underline{\textit{real}}$, $\underline{\textit{string}}$, $\underline{\textit{bool}}$
$(\text{IDENT} \times \text{OBJ})^+$	$\to$ OBJ	$\underline{\textit{tuple}}$
OBJ	$\to$ OBJ	$\underline{\textit{set}}$

Here the $\underline{\textit{tuple}}$ and $\underline{\textit{set}}$ type constructors can be applied independently. For example, a type describing *person* objects is:

$\underline{\textit{tuple}}$([(name, $\underline{\textit{string}}$),
(children, $\underline{\textit{set}}$($\underline{\textit{string}}$)),
(address, $\underline{\textit{tuple}}$([(city, $\underline{\textit{string}}$), (street, $\underline{\textit{string}}$), (number, $\underline{\textit{int}}$)]))])

In summary, the terms of the first signature of an SOS specification define a *type system*, which within the descriptive algebra is equivalent to a DBMS data model.

Now the second signature is used to define operations on the types generated by the first signature. Whereas a signature normally has only a small, finite number of sorts, the second level of signature in SOS generally has to deal with infinitely many sorts. Because of this, we write *signature specifications*. The basic tool is *quantification over kinds*. We define a few example operations for the relational model above:

operators

$\forall$ *data* in DATA.

$data \times data$	$\to \underline{\textit{bool}}$	$=, \neq, <, \leq, \geq, >$

Here *data* is a type variable ranging over the types in kind DATA. Hence it can be bound to any of these types which is then substituted in the second line of the specification. So we obtain comparison operators on two integers, two reals, etc. Relational selection is specified as follows

$\forall$ *rel*: $\underline{\textit{rel}}$(*tuple*) in REL.

$rel \times (tuple \to \underline{\textit{bool}})$	$\to rel$	**select**

:Here $\underline{\textit{rel}}$(*tuple*) is a *pattern* in the quantification which is used to bind the two type variables *rel* and *tuple* simultaneously. Hence the first argument to **select** is a relation of some type *rel*, and the second argument is a function from its tuple type to $\underline{\textit{bool}}$, that is, a predicate on this tuple type. The result has the same type as the first argument.

The second argument of **select** is based on the following extension of the concept of signature defined in SOS: If, for $n \geq 0$, $s_1, \ldots, s_n$ and s are sorts, then $(s_1 \times \ldots \times s_n \to s)$ is a sort (*function* sort). Furthermore,

fun $(x_1\colon s_1, \ldots, x_n\colon s_n) \quad t$

is a term of sort $(s_1 \times \ldots \times s_n \to s)$ iff t is a term of sort s with free variables $x_1, \ldots, x_n$ of sorts $s_1, \ldots, s_n$, respectively.

An operator **attr** allows us to access attribute values in tuples:

$\forall$ *tuple*: $\underline{\textit{tuple}}$(*list*) in TUPLE, *attrname* in IDENT, *member*(*attrname*, *attrtype*, *list*).

$tuple \times attrname$	$\to attrtype$	**attr**

Here *member* is a *type predicate* that checks whether a pair (x, y) with x = *attrname* occurs in the *list* making up the tuple type definition. If so, it binds *attrtype* to y. Hence **attr** is an operation that for a given tuple and attribute name returns a value of the data type associated with that attribute name. – Type predicates are implemented "outside" the formalism in a programming language.

Syntax. The standard syntax for terms over signatures is prefix notation $op(arg_1, \ldots, arg_n)$. Whereas type terms are always written in prefix notation, SOS allows one to specify syntax patterns for operators, to obtain more readable query expressions. For the operators defined above, these might be defined as

$data \times data$	$\to$ $\underline{bool}$	$=, \neq, <, \leq, \geq, >$	_ # _
$rel \times (tuple \to \underline{bool})$	$\to$ *rel*	**select**	_ # [_]
$tuple \times attrname$	$\to$ *attrtype*	**attr**	# (_, _)

Here "_" denotes an argument, "#" the operator; parentheses, brackets, and commas have to be put as shown. So with this syntax specification, comparison operators can be written in infix notation, **select** in postfix with the parameter predicate following in square brackets, and attribute access in prefix notation. The resulting syntax we call *SOS syntax*.

With the few operations defined above we can now write a query. Assuming a relation "people" of type $\underline{rel}$(person) is given and "person" is the name of a tuple type $\underline{tuple}$([(name, $\underline{string}$), (age, $\underline{int}$)]), the query "Find people older than 20" can be written as

people **select** [**fun** (p: person) **attr**(p, age) > 20]

The parser tool implemented in SECONDO can actually infer the tuple type *person* from the type of relation *people* supplied as a first argument to **select** (using an additional specification given with the **select** operator before parser generation), so that this query can be written as

people **select** [**attr**(., age) > 20]

To distinguish the two levels of signature, we call the first *type signature* and the second *value signature.*

3.2 Specifying an Executable Algebra

Precisely the same formalism (although we have not yet seen all of it) can be used to specify an execution system for some data model. In this case, type constructors represent data structures and operators represent query processing algorithms implemented in the system. We show a small part of a relational execution system.

kinds IDENT, DATA, ORD, TUPLE, RELREP

type constructors

	$\to$ DATA	$\underline{int}$, $\underline{real}$, $\underline{string}$, $\underline{bool}$
	$\to$ ORD	$\underline{int}$, $\underline{real}$, $\underline{string}$
$(\text{IDENT} \times \text{DATA})^+$	$\to$ TUPLE	$\underline{tuple}$
TUPLE	$\to$ RELREP	$\underline{srel}$, $\underline{relrep}$

Here we have introduced an additional kind ORD to represent types corresponding to one-dimensional, ordered domains suitable to be indexed in a B-tree. Type constructor $\underline{srel}$ stands for a simple sequential representation of a relation which might be constructed as a result of a query. Type *relrep* will be used to generalize over different relation representations.

We would like to define a type constructor representing a B-tree relation representation ordered by one of the attributes. The type description should also contain the attribute name and data type by which the B-tree is organized. Basically, the signature might be defined as follows.

$$\mathrm{TUPLE} \times \mathrm{IDENT} \times \mathrm{ORD} \rightarrow \mathrm{RELREP} \quad \underline{\mathit{btree}}$$

However, this would not ensure that the attribute name and data type given as the second and third argument do actually occur within the tuple type given as the first argument

The purpose of a signature is to describe what are valid arguments for an operator. Signatures are simple, but a bit limited, since they define any combination of arguments that conform to the given argument sorts to be valid. Sometimes, as for the *btree* constructor, it is necessary to introduce further restrictions to ensure certain relationships between the arguments. We have already seen in the operator specifications above how this can be done. SOS also allows one to write *type constructor specifications*, using the same mechanisms as for operator specifications. In fact, all the signatures for type constructors shown above can be viewed as a shorthand for such specifications. For example, the *srel* specification can also be written as:

$\forall$ *tuple* in TUPLE.
    $\mathit{tuple} \rightarrow \mathrm{RELREP} \quad \underline{\mathit{srel}}$

The *btree* constructor can be defined, using the *member* predicate introduced earlier, as:

$\forall$ *tuple*: *tuple*(*list*) in TUPLE, *attr* in IDENT, *dtype* in ORD, *member*(*attr*, *dtype, list*).
    $\mathit{tuple} \times \mathit{attr} \times \mathit{dtype} \rightarrow \mathrm{RELREP} \quad \underline{\mathit{btree}}$

Now we would like to introduce operators applicable to any existing relation representation. SOS offers *subtype specifications* for this. We can make *srel* and *btree* subtypes of *relrep* by saying:

subtypes

srel(*tuple*)	< *relrep*(*tuple*)
btree(*tuple*, *attr*, *dtype*)	< *relrep*(*tuple*)

Type variables appearing on the left must also appear on the right hand side; hence we have generalization from left to right. We now define a few example operators (comparison operators and **attr** are specified as in the descriptive algebra):

operators

$\forall$ *tuple* in TUPLE, *attr* in IDENT, *dtype* in ORD.

relrep(*tuple*)	→ *stream*(*tuple*)	**feed**	_ #
stream(*tuple*) × (*tuple* → *bool*)	→ *stream*(*tuple*)	**filter**	_ # [_]
stream(*tuple*)	→ *srel*(*tuple*)	**consume**	_ #
btree(*tuple, attr, dtype*) × *dtype* × *dtype*	→ *stream*(*tuple*)	**range**	_ # (_, _)

The **feed** operator scans a relation representation and feeds its tuples into a stream. The **filter** operator applies a predicate to each tuple in a stream; **consume** collects tuples from a stream into a temporary relation. The **range** operator implements a range query on a B-tree, feeding the qualifying tuples into a stream. Syntax patterns are specified as explained in Section 3.1.

Suppose now that the *people* relation from Section 3.1 is represented as a B-tree of type

> *btree*(*tuple*([(name, *string*), (age, *int*)]), age, *int*)

We can then express the query “Find people older than 20” in executable algebra in two different ways. The first query plan scans the relation representation, the second performs a range query.

> people **feed filter** [**fun** (p: person) **attr**(p, age) > 20] **consume**
> people **range**(21, 120) **consume**

3.3 Commands

In the SOS framework, a *database* is a pair (*T*, *O*), where *T* is a finite set of *named types* and *O* is a finite set of *named objects*. A named type is a pair, consisting of an identifier and a type of the current (descriptive or executable) algebra. A named object is a pair, consisting of an identifier and a value of some type of the current algebra. SOS defines six basic commands to manipulate a database, regardless of – or parameterized by – the data model:

> **type** <identifier> = <type expression>
> **delete type** <identifier>
> **create** <identifier> : <type expression>
> **update** <identifier> := <value expression>
> **delete** <identifier>
> **query** <value expression>

A command can be given at the level of descriptive or executable algebra. In the first case, it is subject to optimization before execution; in the second, it is executed directly. In these commands, a *type expression* is a type of the current type signature, possibly containing names of (previously defined) types in the database. A *value expression* is a term of the current value signature, which may also contain constants and names of objects in the database. The **type** command adds a new named type, **delete type** removes an existing type. The **create** command creates a new object of the given type; its value is yet undefined. The **update** command assigns a value resulting from the value expression which must be of the type of the object. The **delete** command removes an object from the database. The **query** command returns a value resulting from the value expression to the user interface or application. Here are some example commands at the executable algebra level:

> **type** city = *tuple*([(name, *string*), (pop, *int*), (country, *string*)])
> **type** city_rel = *srel*(city)
> **create** cities: city_rel
> **update** cities := {enter values into the cities relation, omitted here}
> **query** cities **feed filter** [**fun** (c: city) **attr**(c, pop) > 1000000] **consume**

More details about the SOS framework can be found in [Güt93]. In [BeG95] a descriptive algebra has been defined for GraphDB [Güt94], an object-oriented data model that integrates a treatment of graphs, which shows that the framework is powerful enough to describe complex, advanced data models.

4 The SECONDO System

Second-order signature is the formal basis for specifying data-models and query languages. In this section, we present the SECONDO system frame, providing a clean extensible architecture, implementing all data-model independent functionality for managing SOS type constructors and operators, and supporting persistent object representations. Extending the frame with algebra modules results in a full-fledged database system. In addition to the basic commands presented in Section 3.3, SECONDO provides several other commands, e.g. for transaction management, system configuration, administration of multiple databases, and file input and output.

4.1 Architecture

Figure 1 shows a coarse architecture overview of the SECONDO system. We discuss it level-wise from bottom to top. White boxes are part of the fixed *system frame*, which is independent of the currently implemented data model. Grey-shaded boxes represent the extensible part of the SECONDO system. Their contents differ with specific database implementations.

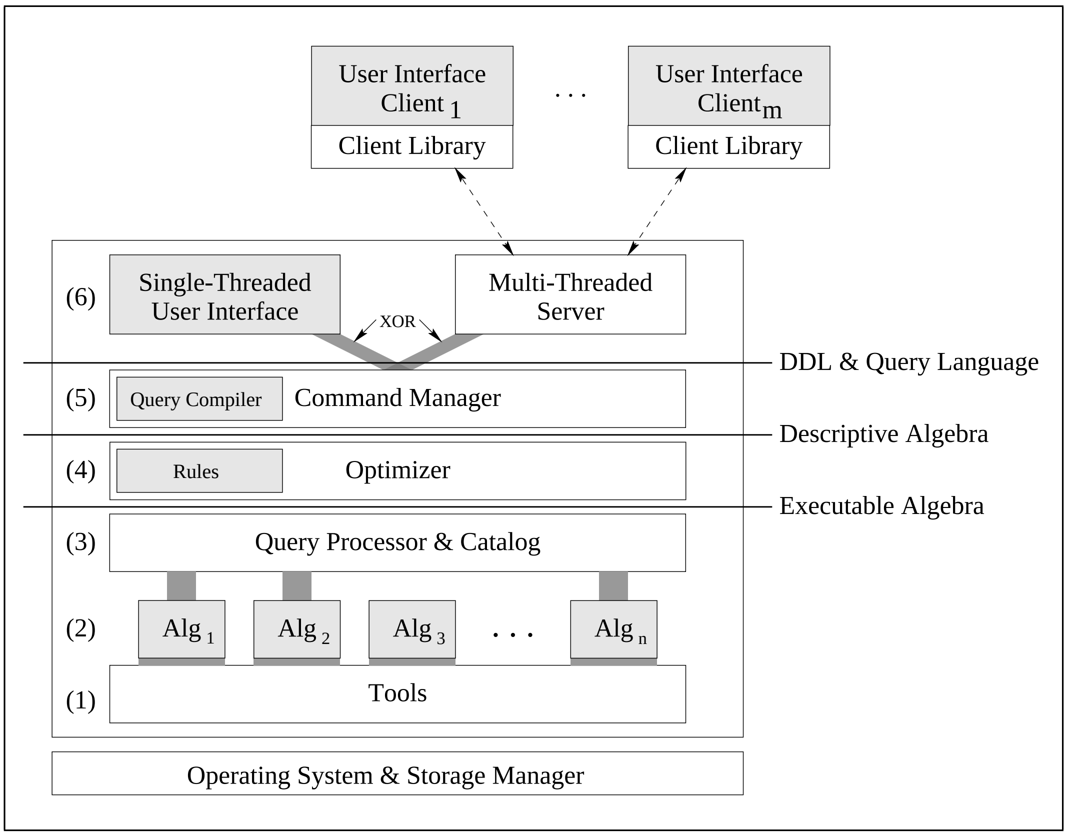

Figure 1: SECONDO architecture

The system is built on top of the Solaris operating system. Since we want to offer a full-fledged database system with features like transaction management, concurrency control, and recovery, using a storage manager for dealing with persistent data is essential. Actually, we use the storage manager component of the SHORE [CaDF+94] system.

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

Nested Lists, a library of functions for easy handling of nested lists, the generic format to pass values as well as type descriptions. Section 4.2.1 describes nested lists in more detail.

SecondoSMI, a simplified storage manager interface to the SHORE functions used most often. It can be used together with original SHORE function calls whenever the simplified functionality is not sufficient.

Catalog Tools for easy creation of system catalogs and algebra-specific catalogs.

Tuple Manager, an efficient implementation for handling tuples with embedded large objects [DiG98]. Often the size of values represented using the large object abstraction provided by modern storage managers actually varies between tuple instances from very small to large. Our approach swaps out embedded objects that are really large, but stores them within the tuple byte string if they are smaller than a suitable threshold size.

SOS parser. As we have seen in the “Syntax” paragraph within Section 3.1, the SOS syntax for query expressions is defined through operator syntax patterns. The SOS parser, extensible by those syntax specifications, transforms an SOS term to the generic nested list format used in the system.

SOS specification compiler. This tool creates the source code for the `TypeCheck` and `TransformType` algebra support functions (see Section 5.1) from a valid SOS specification.

Level 2 is the algebra module level. To some extent, an algebra module of SECONDO is similar to ADTs of PREDATOR [SeLR97] or data blades of Informix Universal Server [Inf98]. Using the tools of level 1, a SECONDO algebra module defines and implements type constructors and operators of an executable query algebra. While there are some good reasons to use C++ for algebra module implementation, SECONDO allows for implementations in Modula-2 and C, too. To be able to use a module’s types and operators in queries, the module must be registered with the system frame, thereby enabling modules in upper levels to call specific support functions provided by the module. In Figure 1, modules 1, 2, and *n* are connected to the frame, thus *active*, while module 3 is *inactive*. C++ modules are activated by linking them to the system frame. In addition to that, activation of C and Modula-2 modules requires insertion of some standardized lines into the body of a predefined startup function.

Level 3 contains the query processor, the system catalog of types and objects (remember that a database is just a set of named types and named objects), and the mechanism for module registration. During query execution, the query processor controls which support functions of active algebra modules are executed at which point of time. Input to the query processor is a query plan, i.e. a term of the executable algebra defined by active algebra modules.

The query optimizer depicted in level 4 transforms a descriptive query into an efficient evaluation plan for the query processor by means of transformation rules. For each algebra module, the database implementor provides a corresponding set of rules as well as algebra support functions supplying information on estimated query execution costs.

The command manager in level 5 provides a procedural interface to the functionality of the lower levels, described in more detail in Section 6. Depending on the command level, the query (or other command) is passed either to the query compiler provided by the database implementor, to the optimizer, or to the query processor.

In level 6 we find the front end of a SECONDO installation, providing the user interface. In general, there are two mutually exclusive alternatives: Either the user interface is linked with the frame and active algebra modules to a self-contained program, or the SECONDO process is made a server process serving requests of an arbitrary number of client processes which implement the user interfaces. In the first case, SECONDO is a single-user, single-process system, in the latter case SECONDO is a multi-user capable client-server system, exploiting the multithreaded environment offered by SHORE. To support the implementation of user clients, SECONDO provides comfortable client libraries for C++ and Java.

4.2 Query Processing

4.2.1 Representation of Type Expressions, Values, and Value Expressions

During query processing, type expressions, values, and value expressions are passed between different modules and functions. We exploit the concept of nested lists, well known from functional programming languages, as a generic means to represent type expressions, queries, and constant values in queries, query results, or external files. A nested list is either a value of an *atom type* (*integer*, *real*, *boolean*, *string*, *text*, or *symbol*), or a list of arbitrary length. Each list element in turn is either an atomic value or a nested list.

The textual representation of a nested list consists of a left parenthesis, followed by an arbitrary number of elements separated by blanks, followed by a right parenthesis. For instance, the list expression `(1 2.01 ((int) (″XX″ TRUE)))` represents a nested list of 3 elements: The integer atom 1 is the first element, the real atom 2.01 is the second one, and the list `((int) (″XX″ TRUE))` is the third one.

The tools layer of SECONDO provides efficient Modula-2, C, C++, and Java libraries for managing nested lists. Using this library, a nested list is a pointer structure in main memory which is referenced by a single word of storage. Thus, passing nested lists as function parameters is very inexpensive. Furthermore, the library provides a persistent version of nested lists and functions to convert a main memory nested list representation to a persistent one and vice versa.

Nested lists are a suitable means to represent *type expressions* of the second-order signature formalism. Let us consider two examples:

- The SOS type expression <u>*rel*</u>(<u>*tuple*</u>([(name, <u>*string*</u>), (pop, <u>*int*</u>)])) corresponds to the list expression `(rel (tuple ((name string) (pop int))))`.

- The argument type specification of a relational selection operator, taking as arguments a value of type `rel` and a function mapping a tuple to boolean, is defined by `[`<u>rel</u>`(`*x*`), (`*x* ⟶ <u>bool</u>`)]`. The corresponding list expression is `((rel x) (map x bool))`.

During query processing, different kinds of types must be considered. *User defined types* and *types of objects* are persistently stored in the system catalog. *Types of constant values* are in textual or in-memory format, depending on the format of query input. *Types of intermediate results* are produced and accepted by the query processing support functions implemented within algebra modules. Thus, whenever a persistent or textual type expression is encountered, it is transformed to in-memory representation before further processing.

Concerning *values*, we distinguish *internal values* and *external values*. Internal values are values of objects and intermediate results. They are always represented as a single word of storage. Both system catalog and query processor use just this word value, regardless of the actual value implementation. If the main memory representation of a value does not fit into a single word, a pointer referencing the value is used instead. For persistent value representations, typically an integer offset into an algebra-specific catalog is used.

External values are found in queries, query results, and the files used by the SECONDO commands for file input and output. External values are represented in nested list format. The algebra implementor specifies the list structure, which is quite straightforward in most cases. Consider a relation value of type <u>rel</u>(<u>tuple</u>([(name, <u>string</u>), (pop, <u>int</u>)])). Its nested list representation is a list of tuple values, each a list consisting of a string and an integer element, for instance `((“New York” 7322000) (“Paris” 2175000))`. For an atomic data type such as <u>polygon</u>, the representation might be a list of pairs of vertex coordinates.

Value expressions define queries or occur as right hand sides of update commands and are essentially terms of a value signature. They can be written either in *SOS syntax* (see Section 3.1) or in *list syntax*, i.e., as nested lists. Expressions in SOS syntax are transformed by the *SOS parser* tool into list syntax, hence for processing, queries are always represented as nested lists. For example, the query plan

cities **feed filter** [**fun** (c: city) **attr**(c, pop) > 1000000] **consume**

can be typed into the system either in SOS syntax

```
cities feed filter[fun (c: city) attr(c, pop) > 1000000] consume
```

or in list syntax

```
(consume (filter (feed cities) (fun (c city) (> (attr c pop) 1000000))))
```

A value expression can in general be a *constant*, an *object* name, an *abstraction* (i.e. a term of a function type), a *list* of value expressions (matching an operator’s argument of a list type), or an *application* of an operator to some value expressions. Constants are either of the predefined nested list atom types *integer*, *real*, *string*, or *boolean*, which can be written directly in the usual notation and are interpreted as values of types `int`, `real`, `string`, and `bool`, if there is an algebra module in the system providing these types. Or they can be generic constants written in the form (<type expression> <value list>), e.g. `(polygon ((1.0 3.8) (4.0 3.8) (2.5 6.0)))`, where *value list* is the external representation of the value.

4.2.2 Processing Type Expressions: Kind Checking

The task of kind checking is to validate type expressions given as argument of the **type** command (see Section 3.3) for the definition of a new named type. This in turn requires checking if all type constructors are applied correctly. In principle, how a type constructor can be applied is defined by its SOS specification. Of course, if the argument types are given in some representation, one can also check this procedurally. In fact, kind checking is implemented in this way: For each type constructor there is a support function, called `TypeCheck`, checking for the type expressions supplied as arguments whether they meet the respective SOS type constructor specification or not. At the moment, the `TypeCheck` function is coded manually by the algebra implementor. Later on, the SOS specification compiler will generate it automatically from the SOS specification.

SOS type constructor specifications contain quantifications over kinds. To reflect quantification over kinds within the implementation of the `TypeCheck` function, the system frame offers the catalog function `CheckKind`, returning the disjunction of all `TypeCheck` functions pertaining to a given kind. Setting up the association of type constructors and kinds takes place in the startup routines (see Section 5.1) of active algebra modules.

4.2.3 Processing Value Expressions: Type Checking and Evaluation

A query (or value expression) is evaluated in three steps. First, the expression, given as a nested list, is *annotated*, which includes type checking. Second, from the annotated expression (also a nested list), an operator tree is built. Third, evaluation is called for the root of the operator tree. We discuss each step in turn.

Annotating the Query. Given a value expression such as

```
(consume (filter (feed cities) (fun (c city) (> (attr c pop) 1000000))))
```

a recursive procedure *annotate* essentially processes the tree represented by the nested list bottom-up, annotating each node with its type. More precisely, for each atom or sublist *s* in the query, a list of the form ((*s*, *class*, …), *type*) is returned, where *class* is a keyword classifying the element, and after *class* some specific information for this kind of element follows. Classes are, for example, *constant*, *operator*, *abstraction*; more specific information is for a constant where its value can be found, or for an operator its *algebra number* and *operator number* which in the system identify an operator uniquely. The *type* of the element is, of course, also given as a nested list. For example, for the abstraction above the returned type would be

```
(map (tuple ((name string) (pop int) (country string))) bool)
```

The essential step in annotation is the type checking of operator applications, that is, checking whether the argument types are correct, and determining the result type. Like for type constructors, how an operator can be applied is determined by its SOS specification. Again, if the argument types are given in some representation, one can also check procedurally whether they are correct and what the result type is. And indeed, in our implementation for each operator there is a support function `TransformType` which, given the argument types as a list of nested lists, returns the result type of the operator application, or an atom *error.* Once the SOS specification compiler is available, these support functions will be generated automatically; currently, they are still hand-coded.

Annotation also resolves the *overloading* of operators by means of another support function `Select`. In general, for each operator there may be an array of *evaluation* support functions, dealing with specific combinations of argument types. The `Select` function takes a list of argument types and returns a number which is an index into the array of evaluation functions. Annotation calls `Select` with the actual types in the operator application and puts the returned *evaluation function number* into the annotated expression.

There are some other facilities in annotation that cannot be explained here in detail for lack of space, such as the possibility to compute *derived arguments* by type mapping functions, or *derived function argument types*.

Building the Operator Tree. Given the annotated value expression, it is relatively easy to build an operator tree. Such a tree has three kinds of nodes, namely

1. *Object* nodes, representing a database object or a constant value,
2. *Indirect object* nodes, representing a variable in the expression of an abstraction,
3. *Operator* nodes, representing the application of an operator to some arguments.

Object and indirect object nodes are leaves, operator nodes internal nodes of the tree. Object nodes contain the value of the object or constant, represented in a single word of storage. Indirect object nodes contain a reference to an argument vector for the abstraction (parameter function), and an index into that vector. The argument vector is an array of WORD and contains single word value representations. Hence an indirect object also refers to a value. Operator nodes contain an array of pointers to subtrees, algebra number and evaluation function number for the operator, a flag indicating whether the operator is the root of a subtree representing an abstraction (and if it is, a pointer to the abstraction's argument vector), and a flag whether the operator returns a result of type <u>*stream*</u>. Figure 2 shows an operator tree for the example query above.

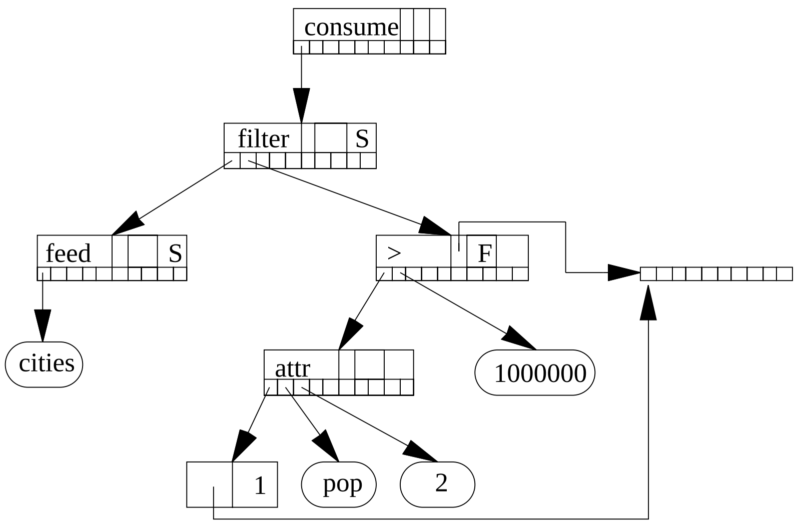

Figure 2: Operator tree

In an operator node, we have represented the operator by its name rather than its pair of numbers. An “S” represents a true “stream” flag, “F” the “root of abstraction” (function) flag. The right subtree of the **filter** operator represents as a whole the abstraction subexpression

```
(fun (c city) (> (attr c pop) 1000000))
```

The argument vector for the abstraction is shown to the right. Note that the third argument for the **attr** operator, the attribute number of “pop” within a city tuple, is a *derived argument* added in annotation by the operator’s type mapping function. Hence the evaluation function can use this number for tuple access instead of the attribute name.

Building the operator tree is easy because annotation has collected all the relevant information. For example, the “stream” flag is set if the outermost type constructor of the operator’s result type is `stream`; or the symbol “c”, the first argument to **attr**, has been analyzed to be the first parameter of the abstraction which leads to setting up an indirect object node.

Evaluation. Evaluation is done in close cooperation between a recursive function *eval* of the query processor, applied to the root of the operator tree, and the operators’ evaluation functions. The basic strategy is, of course, to evaluate the tree bottom-up, calling an operator’s evaluation function with the values of the argument subtrees that have been determined recursively by *eval*. Interesting aspects are the treatment of abstractions (parameter functions of operators) and stream processing. To support this, function *eval* has the basic structure shown in Figure 3.

```
function eval (t : node): WORD;
input: a node t of the operator tree;
output: the value of the subtree rooted in t;
method:
    if t is an object or indirect object node then lookup the value and return it
    else {t is an operator node}
        for each subtree $t_i$ of t do {evaluate all subtrees that are not functions or streams}
            if the root of $t_i$ is marked as function (F) or stream (S) then $arg_i := t_i$
            else $arg_i := eval(t_i)$
            end
        end;
        Call the operator’s evaluation function with argument vector arg and return its result
    end
end eval
```

Figure 3: Function *eval*

The generic interface of an operator evaluation function basically has the form

function *alpha* (*arg*: *argVector*; **var** *result*: WORD): *integer*;

where *argVector* is an array of WORD. The returned *integer* value is used for error messages. As we have just seen, the arguments supplied to the evaluation function are either values, or in case of arguments that are parameter functions (abstractions) or streams, pointers to the respective subtrees which we call *suppliers*. An operator with a function argument needs to pass parameter values to that function and then to call for its evaluation. The query processor supports this by offering two primitives *argument* and *request*. For a given supplier, *argument* returns a pointer to its argument vector. The caller can then put values into that vector. *Request* calls for evaluation of a supplier which is implemented by calling *eval*. For example, the evaluation function for the **filter** operator in Figure 2 gets the argument vector for its second argument and puts the current tuple value into its first field. It then calls *request* for this second argument which returns the boolean value resulting from evaluating the parameter function.

An operator with a stream argument (e.g. **consume**) uses *request* in the same way to get an element of the stream. In addition, two more primitives *open* and *close* are available for stream suppliers. The query processor implements these as calls of the supplier's evaluation function with a *message* from the set {OPEN, REQUEST, CLOSE}. Hence the generic interface of an evaluation function is in fact a bit larger:

function *alpha* (*arg*: *argVector*; **var** *result*: WORD; *m*: *message*): *integer*;

An evaluation function returning a stream (e.g. for **feed**, **filter**) is structured into three parts according to the three possible messages. Usually for *open* some initialization actions have to be performed and for *close* some cleaning up. A stream evaluation function returns (in the integer return value) one of two special values CANCEL and YIELD to inform the consuming operator whether it did deliver a stream element or the stream was exhausted. Via a final primitive *received* the consumer can check for a supplier whether it sent CANCEL or YIELD.

A description of query processing in SECONDO at a more technical level, including many code examples, e.g. for evaluation functions, can be found in [GüFB+97].

5 Algebra Modules

5.1 Structure

In addition to the data structures and support functions already mentioned, an algebra module contains components to support optimization. For each type constructor, a *model* may be registered which is a data structure containing summary information about a value of the type. The model is a place to keep statistical information such as (expected) number of tuples, histograms about attribute value distribution, etc. For maintaining models there are three support functions `InModel`, `OutModel` and `ValueToModel`, explained in Table 1.

Furthermore, for each operator, there are two support functions related to optimization called `MapModel` and `MapCost`. For an operator application, `MapModel` takes the models of the operator's arguments and returns a model for the result. `MapCost` also takes the models of the operator's arguments as well as the estimated costs for computing the arguments and, based on these, estimates and returns the cost for the operator application. By calling these support functions, the optimizer can estimate properties of intermediate results and the cost of query plans or subplans.

Table 1 lists all support functions for type constructors implemented in algebra modules. In addition to these functions, also the type constructor name is passed to the system frame.

`In/Out`	Conversion from nested list to internal value representation and vice versa.
`Create/Delete`	Allocate/deallocate memory for internal value representation.
`TypeCheck`	Validation of type constructor applications in type expressions.
`InModel/OutModel`	Conversion from nested list to internal model representation and vice versa.
`ValueToModel`	Computes a model for a given value.

Table 1: Support functions for type constructors

Table 2 presents all support functions for operators. For each operator, also its name and the number of evaluation functions are passed to the system frame.

`TransformType`	Computes the operator’s result type from given argument types.
`Select`	Selects the correct evaluation function in case of overloading by means of the actual argument types.
`Evaluate`	Computes the result value from input values. In case of overloading, several evaluation functions exist.
`TransformType`	Computes the operator’s result type from given argument types.
`MapModel`	Computes the result model from argument models. Optional.
`MapCost`	Computes the estimated cost of an operator application from argument models and costs.

Table 2: Support functions for operators.

Furthermore, there is a startup routine for each algebra module which is used to associate type constructors with the kinds containing them, to perform initializations of global arrays, etc.

The registration mechanism for support functions differs with the implementation language. Registration is most comfortable in C++: For each type constructor, an instance of the predefined class `TypeConstructor` must be defined, passing operator support functions as constructor arguments. The same happens with operators and a predefined class `Operator`. For a complete algebra, an instance of a class derived from the predefined class `Algebra` is defined. The constructor of the derived class is the startup routine of the modules.

5.2 Module Interaction

Algebra modules need not only to cooperate with the system frame, but also with other algebra modules. Modules implement certain signatures. At the type level, *kinds* are the junctions between different signatures. For instance, each type in kind DATA will be a valid attribute type for the *tuple* type constructor. Thus, a type constructor *polygon* is made a type constructor for attribute types by simply adding *polygon* to the type constructors for DATA.

At the implementation level, the interface between system frame and algebra modules does not impose any specific inter-module interaction conventions on the algebra implementation, but rather the algebra implementor is free to define the protocol for interaction with type constructors and operators of his algebra. For C++ implementations there is a general strategy, based upon the inheritance and virtual method mechanisms provided by C++, which allows one to define generic interfaces between modules in a uniform manner as follows.

The basic observation is that the relationship between kinds and type constructors corresponds to the relationship between base classes and derived classes. For each kind K, the algebra module *alg* requiring an interface to values of types in K defines an abstract base class *k_base*. For the implementation of operators in *alg*, typically some support functions for dealing with values of kind K will be necessary. Just these support functions are defined as abstract virtual methods of *k_base*. Whenever a class *tc* in any algebra module is defined to implement a type constructor in kind K, *tc* must be derived from *k_base*: `class tc : public k_base`. For instance, the base class `Data` corresponding to kind DATA contains a virtual method `Compare` which has to be defined within all attribute data type implementations, thereby enabling the generic implementation of the `sort` operator of the relational algebra module.

6 User Interfaces

User interfaces communicate with the system frame through a single procedure called `Secondo`. Its input parameters are (i) the user command, given as an SOS command string, a textual list expression, or an in-memory nested list representation, (ii) a parameter specifying the command level: executable algebra, descriptive algebra, or specific data definition and query language, and (iii) a parameter which determines whether the result is returned in textual or in-memory nested list representation. Procedure `Secondo` returns the query result type and value as well as error information. Additional algebra-specific error information is returned if the error has occured in one of the support functions provided by algebra modules during query processing.

We distinguish two kinds of user interfaces: application specific interfaces and generic ones. Specific interfaces provide user interaction for a DBMS with a fixed data model. The representation of input or result values can be hard-coded for the specific set of data types. This makes user interface implementation a quite straightforward issue; on the other hand, extending the data model requires an expensive reimplementation of the former interface, or even the complete implementation of a new one.

In contrast to specific user interfaces, generic interfaces pre-implement most of the user interface functionality. Only the algebra-specific details of value representation must be provided by the algebra implementor. Again, the concept of support functions provided by algebra implementors can be employed. Since not only the result *value*, but also the result *type* is returned by procedure `Secondo`, the decision which support function to call can be made on the basis of the outermost type constructor in the actual type expression.

In fact, we have implemented two simple generic interfaces in C++ and Java. Query results are presented in list form as returned by the system. However, if a specific display function for an actual result type has been registered with the user interface, the display function determines the formatting of the result value. For instance, a result value of type *rel*(...) is printed in a tabular style.

7 Conclusions

We have presented SECONDO, a new generic environment supporting the implementation of database systems for a wide range of data models and query languages. The second-order signature formalism is the basis for data type and operator specification on the descriptive as well as the executable algebra level. We have shown the correspondence between abstract SOS specifications and data type and operator implementation, arranged into small algebra modules which extend the system frame in a clear and easy manner.

At the moment, architecture levels 1, 3, 5, and 6 in Figure 1 are operational as described in Section 4.1. As regards level 2, an algebra module with type constructors *int*, *real*, *bool*, and *string* and a comprehensive set of operations on these types is implemented. Furthermore, a relational algebra algebra module offers type constructors *rel* and *tuple* with all standard relational operators plus some non-standard operators. Currently, several projects are running to extend SECONDO:

- The implementation of a new algebra module providing the functionality of GraphDB [Güt94], a sophisticated graph-oriented data model.
- The implementation of an algebra module providing index data types and operations.
- Design and implementation of modules for spatio-temporal data types like “moving point” and “moving region” [ErGSV97].
- A generic evaluation plan generator.

For the time being, algebra support functions dealing with type expressions have to be coded manually. In the future, the SOS specification compiler will create those functions automatically. Other future work will focus on the completion of the extensible rule-based query optimizer.

Acknowledgments

For their contribution to the design and implementation of the SECONDO system we thank Friedhelm Becker, Ludger Becker, Jose Antonio Cotelo Lema, Claudia Freundorfer, Miguel Rodriguez Luaces, and Holger Schenk.

References

[BaBG+88] D. S. Batory, J. R. Barnett, J. F. Garza, K. P. Smith, K. Tsukuda, B. C. Twichell, T. E. Wise: GENESIS: An Extensible Database Management System. *IEEE Transactions on Software Engineering 14(11)*: 1711-1730 (1988).

[BaK86] F. Bancilhon and S. Khoshafian. A Calculus for Complex Objects. Proc. ACM Symposium on Principles of Database Systems, pp. 53-59, Cambridge, Mass., 1986.

[BeG95] L. Becker and R. H. Güting. The GraphDB Algebra: Specification of Advanced Data Models with Second-Order Signature. FernUniversität Hagen, Informatik-Report 183, May 1995.

[CaD96] M. J. Carey and D. J. DeWitt. Of Objects and Databases: A Decade of Turmoil. In *Proc. of the 22nd VLDB Conference*, pp. 3-14, Mumbai (Bombay), September 1996.

[CaDF+86] M. J. Carey, D. J. DeWitt, D. Frank, G. Graefe, M. Muralikrishna, J. E. Richardson, E. J. Shikita. The Architecture of the EXODUS Extensible DBMS. In *Proc. of the 1st Intl. Workshop on Object-Oriented Database Systems*, pp. 52-65, Pacific Grove, September 1986.

[CaDF+94] M. J. Carey, D. J. DeWitt, M. J. Franklin, N. E. Hall, M. L. McAuliffe, Je. F. Naughton, D. T. Schuh, M. H. Solomon, C. K. Tan, O. G. Tsatalos, S. J. White, M. J. Zwilling. Shoring Up Persistent Applications. In *Proc. of the 1994 ACM SIGMOD Intl. Conference*, pp. 383-394, Minneapolis, May 1994.

[DiG98] S. Dieker and R. H. Güting. Efficient Handling of Tuples with Embedded Large Objects. Informatik-Report 236, FernUniversität Hagen, August 1998.

[ErGSV97] M. Erwig, R. H. Güting, M. Schneider, and M. Vazirgiannis. Spatio-Temporal Data Types: An Approach to Modeling and Querying Moving Objects in Databases. Informatik-Report 224, FernUniversität Hagen, December 1997. To appear in *GeoInformatica*.

[Gra94] Goetz Graefe. Volcano - An Extensible and Parallel Query Evaluation System. *IEEE Transactions on Knowledge and Data Engineering*, 6(1):120-135, February 1994.

[GrM93] G. Graefe and W. J. McKenna. The Volcano Optimizer Generator: Extensibility and Efficient Search. In *Proc. of the 9th IEEE Intl. Conf. on Data Engineering*, pp. 209-218, Vienna, April 1993.

[GüFB+97] R. H. Güting, C. Freundorfer, L. Becker, S. Dieker, and H. Schenk. Secondo/QP: Implementation of a Generic Query Processor. Informatik-Report 215, FernUniversität Hagen, February 1997.

[Güt89] R. H. Güting. Gral: An Extensible Relational Database System for Geometric Applications. In *Proc. of the 15th VLDB Conference*, pp. 33-44, Amsterdam, August 1989.

[Güt93] R. H. Güting. Second-Order Signature: A Tool for Specifying Data Models, Query Processing, and Optimization. In *Proc. of the 1993 ACM SIGMOD Conference*, pp. 277-286, Washington, June 1993.

[Güt94] R. H. Güting. GraphDB: Modeling and Querying Graphs in Databases. In *Proc. of the 20th VLDB Conference*, pp. 297-308, Santiago, September 1994.

[Inf98] Informix Software, Inc. *Extending INFORMIX-Universal Server: Data Types, version 9.1*, 1998. Online version (http://www.informix.com).

[OnFS84] J. Ong, D. Fogg, and M. Stonebraker. Implementation of Data Abstraction in the Relational Database System Ingres. *SIGMOD Record*, 14(1):1-14, March 1984.

[PaYK+97] J. M. Patel, J. Yu, N. Kabra, K. Tufte, B. Nag, J. Burger, N. E. Hall, K. Ramasamy, R. Lueder, C. Ellman, J. Kupsch, S. Guo, D. J. DeWitt, J. F. Naughton. Building a Scaleable Geo-Spatial DBMS: Technology, Implementation, and Evaluation. In *Proc. of the 1997 ACM SIGMOD*, pp. 336-347, Tucson, May 1997.

[ScCF+86] P. M. Schwarz, W. Chang, J. C. Freytag, G. M. Lohmann, J. McPherson, C. Mohan, and H. Pirahesh. Extensibility in the Starburst Database System. In *Proc. of the 1st Intl. Workshop on Object-Oriented Database Systems*, pp. 85-92, Pacific Grove, September 1986.

[SeLR96] P. Seshadri, M. Livny, and R. Ramakrishnan. The Design and Implementation of a Sequence Database System. In *Proc. of the 22nd VLDB Conference*, pp. 99-100, Mumbai (Bombay), September 1996.

[SeLR97] P. Seshadri, M. Livny, and R. Ramakrishnan. The Case for Enhanced Abstract Data Types. In *Proc. of the 23rd VLDB Conference*, pp. 66-75, Athens, August 1997.

[StR86] M. Stonebraker and L. A. Rowe. The Design of Postgres. In *Proc. of the 1986 ACM SIGMOD Conference*, pp. 340-355, Washington, June 1986.