Gral: An Extensible Relational Database System for Geometric Applications

Ralf Hartmut Güting

Fachbereich Informatik, Universität Dortmund
D-4600 Dortmund 50, West Germany

Abstract: We describe the architecture of a relational database system that is extensible by user-defined data types and operations, including relation operations. The central concept is to use languages based on many-sorted algebra to represent queries as well as query execution plans. This leads to a simple and clean extensible system architecture, eases the task of an application developer by providing a uniform framework, and also simplifies rule-based optimization. As a case study the extensions needed for a geometric database system are considered.

1. Introduction

Much of the database research of recent years was aimed at providing a better support for non-standard applications such as office information systems, geographic information systems, CAD databases, etc. A common need of these applications is the representation and manipulation of more complex objects than those representable by a tuple of a relation in the traditional relational model, for example, an office form, a complete map or a river, say, in a geographic information system, or the design of a VLSI circuit.

A fundamental choice for the representation of a complex object is whether its structure should be visible or hidden at the level of the data model. Speaking in terms of the relational model, for representation the question is whether the object should be described by a collection of tuples from various relations, or by a single attribute value from a specific domain for this kind of objects. For manipulation, the distinction is whether the internal structure is accessible to the general facilities of the query language (selection of subobjects, for example) or only to domain-specific operations. The two ways of handling complex objects have been called structural and behavioural object orientation, respectively [Di86].

Both approaches are obviously needed and are appropriate for certain applications. For example, one should be able to define the internal structure of an office document within the data model and to access it through the query language. On the other hand, a river can well be represented as an atomic value of an abstract data type LINE with suitable operations, for instance, a function returning the length.

A lot of work has been done to support the modeling of visible object structures. Enhancements to the relational model have been proposed by linking together tuples to represent an object either explicitly [Co79, HaL82] or implicitly, through the use of nested relations [ScS86]. Most of the more recent data models and system proposals do also support structural object orientation, for example [MaD86, CaDV88, PiT86].

The idea of allowing application-specific abstract data types as base types, or attribute domains, of a database system was perhaps first put forward in [StRG83]. Since base types need to be implemented in a programming language and because they are application-specific, a user must be able to implement such a type and to add it to a database system. This observation has led to efforts by several groups to construct *extensible* database systems. Two directions can be distinguished. One is to select a data model and to implement for this data model a system with well-defined interfaces for user extensions. This is the approach chosen by the POSTGRES [StR86] and Starburst [Schw86] projects, based on the relational model, and within the PROBE project [Daya87] for an extended functional data model. A different view is taken in the EXODUS [Care86] and GENESIS [Bato86] projects where a collection of powerful tools for *building* a database system is provided. In this case an application developer ("users" providing extensions have also been called "database implementor" [Care86] or "database customizer" [Lo88]) implements the whole database system, aided by the tools. With this approach there is more flexibility for the application even to select a specific data model.

In this paper we describe an (implemented and running) relational database system, called Gral[1], extensible by user-defined data types and operations. This means that an application developer (AD) can

- add data types and operations to the query language,
- add the representation of a type to the system,
- add new relation representations or index structures to support type-specific storage or searching,
- add the implementation of type-specific operations, and
- add rules to the system to describe the implementation of a new query language operation and to support optimization for this operation.

The key concept embodied in Gral, which distiguishes it from other work, is *many-sorted algebra*. Gral uses many-sorted algebra as a query language and also as an executable language to describe query plans (access plans). The whole system is centered around this formal concept and we show that this leads to a very simple and clean extensible system architecture. For example, there is a layer in the system corresponding to the sorts of the executable algebra and another layer corresponding to the operators.

---

[1] Gral stands for "**G**eo-**R**elational **Al**gebra" which is the query language of the Gral system.

Algebraic query languages have been used before [Gr84] and are perhaps now viewed as old-fashioned. However, they have clear advantages for the use in an extensible system, because they are inherently modular and extensible. SQL- or QUEL-like languages accomodate easily new functions or predicates for new atomic types (e.g. in where-clauses). Yet it is awkward to add new relation operators to such languages. As we shall see, non-standard applications do require special relation operations (see also [OsH86]). Many-sorted algebra and the Gral system allow to add any kind of user-defined relation mapping. The particular kind of algebra that we use includes very powerful retrieval operations such as selection or join with an arbitrary algebra expression as condition and the embedding of derived values (described by algebra expressions) into relations.

The optimizer translates query language expressions to query execution plans. In Gral both are algebraic languages with the same structure. This eases considerably the application developer´s task who needs to be familiar with both language levels. In particular it makes it relatively easy to formulate the rules driving the query optimizer.

The Gral project started with the goal of building an efficient database system for geometric applications. While this is still our goal, the need for extensibility was soon recognized and the focus shifted initially to provide a clean and simple extensible architecture. We find that having in mind a specific application is quite helpful to see clearly what kind of extensibility is required.

We consider an extensible system to have a model and a system part and, orthogonally, an application-independent and an application-specific part. We use the following terminology. The application-independent parts are called the *Gral model frame* and the *Gral kernel system.* Speaking of application extensions in general we use the terms *Gral application model* and *Gral application (system)*. Finally, the running Gral prototype is one specific application; its data model is *geo-relational algebra* and we call it the *Gral system.* The following three sections describe Gral's query language, its executable language, and the system architecture. Related work is mentioned in Section 5.

2. An Extensible Data Model and Query Language

Basically Gral uses an extended relational algebra as a query language. More precisely, Gral's data model and query language are based on the formal concept of a *many-sorted algebra*. A many-sorted algebra is a collection of sets and functions between these sets; it is described by an *S-sorted signature* Σ where S is a set of *sorts* (names for the sets) and Σ a family of sets $\Sigma_{w,s}$ of operator symbols (names for the functions), where $w \in S^*$ and $s \in S$ describe the functionality of operators in $\Sigma_{w,s}$ [GoTW78]. For Gral, this algebra has one sort for relations and further sorts for atomic data types; the operations include relational operators such as selection or join and operations on atomic data types such as integer multiplication or a test whether two polygons intersect. More precisely, again, the structured data objects in Gral are not relations (sets of tuples) but *sequences of tuples*, or ordered relations. This allows, for example, to include sequence and sorting operators in the algebra.

Since Gral is an extensible database system together with one specific extension we can distinguish between the application-independent *model frame* and the application-specific data model called *geo-relational algebra*. The model frame describes any application as a many-sorted algebra; it defines the application-independent sorts and operators and is abstract with respect to the application-specific sorts and operators. The set of sorts is $S = \{s_1, \ldots, s_n\}$ with s_1 = REL and s_2 = BOOL. The collection of operators (signature) of the model frame is $\Sigma = \Sigma_G \cup \Sigma_A$ (G for "general", A for "application-specific") of which Σ_G is defined within the model frame. Some operators of Σ_G are shown in the following table.

REL × REL	→ REL	**union**, **product**, **join**, **concat**	_ _ #
REL	→ REL	**select**, **project**, **extend**, **ord**, **head**	_ #
REL	→ ATOM	**extract**	(_ #)
BOOL × BOOL	→ BOOL	**and**, **or**	(_ # _)
BOOL	→ BOOL	**not**	# (_)

The model frame contains sorts for relations (sequences of tuples) and Boolean values and their associated operations. The rightmost column shows the syntax for using an operator within an expression (a query); "_" denotes an operand and "#" the operator, parentheses are to be used as indicated. For example "rel1 rel2 **union**" and "(x **and** (y **or** z))" are well-formed expressions. Some operators take parameters which appear in square brackets behind the operator symbol, as, for example, in "cities **select** [cpop > 500000] **project** [cname, cpop]", for a "cities" relation with attributes for city name and population.

The set of sorts S has a subset ATOM = $\{s_2, \ldots, s_n\}$ which contains the atomic data types. The model frame provides also a set ORD $\subseteq$ ATOM containing the atomic data types that are one-dimensional ordered domains so that a relation can be sorted by attributes of such a type.

Any application-specific database system must specify its atomic data types and may provide within Σ_A any operations on these atomic types, but also operations involving relations. Standard types such as integer, real, and string with suitable operations are likely to occur in most applications. Sorts INT, REAL, and STR are also part of the geo-relational algebra together with the following operations:

$STD_i \times STD_i$	→ BOOL	=, ≠, <, ≤, ≥, >	(_ # _)
INT × INT	→ INT	+, -, *, **div**, **mod**	(_ # _)
INT × INT	→ REAL	/	(_ # _)
REAL × NUM	→ REAL	+, -, *, /	(_ # _)
NUM × REAL	→ REAL		
REL	→ INT	**count**	(_ #)
NUM*	→ REAL	**avg**	(_ #)
NUM_i*	→ NUM_i	**sum**, **min**, **max**	(_ #)

In this listing some abbreviations are used. STD and NUM denote sets of sorts, namely, STD = {INT, REAL, STR} and NUM = {INT, REAL}. The notation X_i with such a set identifier means that X_i is to be consistently replaced by the same element of set X. So the first line allows comparisons of any two operands of equal type. The notation X* denotes a relation operand where an attribute of type X is specified, or, intuitively, a set of X values. The attribute is given as a parameter of the operator, for example, "(cities **avg** [cpop])" returns the average population of cities (within the cities relation) as a real number. The last line is also an abbreviation for two lines with the functionalities INT* → INT and REAL* → REAL.

We can now illustrate the operators of the model frame by a few examples.

(1) Determine the five cities with the highest population numbers.

cities **ord** [cpop -] **head** [5]

The **ord** operator sorts cities descending by population; from the resulting sequence **head** returns the first five tuples.

(2) Add to each city tuple an attribute giving its population in percent of the maximal population.

cities
  extend [(cpop/(cities **max** [cpop]))*100 {percent}]

The **extend** operator takes a list of algebra expressions as parameters, each expression is followed by a new attribute name. For each expression, the relation will have a new attribute whose value is determined for each tuple by evaluating the expression.

(3) Find pairs of cities whose population number differs by less than 50000.

cities cities {*->*2} **join** [(cpop - cpop2) < 50000]

For the second operand relation, the attributes are first renamed (appending a suffix "2") before the **join** operator is applied. Note that the **select** as well as the **join** operator take arbitrary algebra expressions with Boolean result as parameters.

(4) How many people live in Dortmund, in thousands?

(cities **extract** [cname="Dortmund"; cpop]) **div** 1000

The **extract** operator allows one to extract from a relation a single atomic value which is then available for further operations. Parameters of **extract** are an algebra expression (as a selection condition which must return exactly one tuple) and an attribute name.

An application may decide to choose the collection of standard types and operators shown here "from the shelf". However, since all atomic types, except for BOOL, are considered application-specific, it may instead also use very long integers, text fields containing formatting instructions, etc.

An application will then introduce its own more specific data types. Gral's data model, the geo-relational algebra, is designed to support geometric applications in the plane, perhaps with a bias towards supporting geographic information systems. Gral has therefore types for points, lines, and regions in the plane. A line is conceptually a finite sequence of straight line segments. A region is a hole-free, non-selfintersecting polygon. Actually there are two types of regions called PGON and AREA; for type AREA the restriction holds that no two polygons that are attribute values of the same relation may intersect. Type AREA is used to model subdivisions of the plane as they often occur in maps. So the complete set of sorts of the geo-relational algebra is S = {REL, BOOL, INT, REAL, STR, POINT, LINE, PGON, AREA}. To simplify the signature there is also a hierarchy within the geometric types: REG = {PGON, AREA} (regions), EXT = {LINE} ∪ REG (extended objects), and GEO = {POINT} ∪ EXT (any geometric object). A selection of the operators is shown below:

GEO × REG	→ BOOL	**inside**	(_ # _)
EXT × EXT	→ BOOL	**intersects**	(_ # _)
LINE* × LINE*	→ POINT*	**intersection**	_ _ #
LINE* × REG*	→ LINE*		
PGON* × REG*	→ PGON*		
POINT* × POINT	→ REL	**closest**	_ _ #
POINT × POINT	→ REAL	**dist**	# (_ , _)
LINE	→ REAL	**length**	# (_)
REG	→ REAL	**perimeter**, **area**	# (_)

The **inside** operator tests whether any kind of geometric object is wholly contained within a region. The **intersects** operator checks extended objects for intersection. The **intersection** operator (with functionality LINE* × LINE* → POINT*) is applied to two relation operands; for each operand an attribute of type LINE is specified. It returns a relation which "looks like" the cartesian product of the operand relations extended by a POINT attribute. To be precise, the result relation contains one tuple for each intersection point of a line of the first operand with a line of the second operand. This tuple consists of all attributes of the (tuple with the) first line, all attributes of the second line, and the intersection point. Intuitively, the functionality should be read "The intersection of two sets of lines is a set of points". Forming the intersection of lines is embedded in this way into a relation operation because even the intersection of two single lines is generally a set of points, so one cannot define an atomic operator with functionality LINE × LINE → POINT. The **intersection** operator can be applied similarly to a set of lines and a set of regions, or two sets of regions. The **closest** operator is given a relation with a point attribute specified, and a (query) point; it returns the tuple (or those tuples) of the operand relation whose point attribute has minimal distance to the query point. Finally, our selection shows operators to compute the distance between two points, the length of a line, and perimeter and area of a region. Some example queries, based on relations representing cities, states, rivers, and highways of West Germany, illustrate the use of these operators.

cities	cname	center	cpop
	STR	POINT	INT
states	sname	region	spop
	STR	AREA	INT
rivers	rname	route	
	STR	LINE	
highways	hname	way	
	STR	LINE	

(5) Associate with each city the state containing it.

cities states **join** [center **inside** region]
  project [cname, sname]

(6) Find rivers passing through the state of Bayern (Bavaria).

states **select** [sname="Bayern"] rivers **join** [route
  intersects region] **project** [rname]

(7) Which city with more than 100000 inhabitants is closest to the point where highway A1 passes the Rhine?

(rivers **select** [rname="Rhein"]
  highways **select** [hname="A1"]
  intersection [route, way, {crossing}]

extract [true; crossing]) {crosspoint};
cities **select** [cpop > 100000] crosspoint
closest [center]

The query consists of two steps. In the first step the intersection point of river Rhine and highway A1 is computed and made available as a POINT object named "crosspoint". In the second step the **closest** operator is applied to the cities with more than 100000 inhabitants, and the point "crosspoint".

(8) Sort cities by distance from München (Munich).
(cities **extract** [sname="Muenchen"; center])
{Munich};
cities **extend** [**dist** (center, Munich) {MunDist}]
ord [MunDist +]

By a *data model* we mean a mathematical model for data and operations on data. In this sense, the Gral model frame is a data model; its structured objects (sequences of tuples) and operations have been formally defined in [Gü88b]. The essential part of the semantics definition is the treatment of nested algebra expressions (since an operator, for example **select**, **join**, **extend** may have an algebra expression as a parameter); this and most of the operator definitions can also be found in [GüZC88]. For the application-specific extension *geo-relational algebra* semantics have also been formally defined in [Gü88b], that is, a model for LINE and PGON objects is given and the operations shown above (as well as the remaining operators) have been defined in terms of these object models. Geo-relational algebra as such is discussed in more detail and illustrated by examples in [Gü88a].

The Gral model frame is extensible in that new sorts for atomic data objects as well as new operators involving old or new atomic data types and relations can be added. To perform such a data model extension an application developer should provide a mathematical model for a new data type. For operators the functionality and allowed parameters must be specified and the mapping associated with the operator be defined. This corresponds to a formal specification of a program to be implemented. In many cases application developers will omit the specification of extensions at the level of the formal data model and just implement new types and operators. However, in case of more complex data types, and to be able to check the correctness of the implementation, it is advisable to also extend the formal model.

We have seen that the algebra is used rather directly as a query language. To extend the query language, besides introducing new types and operators, one needs to specify syntactical aspects such as a pattern for using the new operator, e.g. "(_ # _)", and the structure of the parameter list, if the operator has parameters. To make the query language somewhat homogeneous and to allow the Gral system parser to analyze expressions, certain rules have to be followed:

(A) Use postfix notation whenever an operator takes at least one relation as an operand.

(B) Put parentheses if and only if the resulting object is atomic.

There is a lot of flexibility in the allowed syntactical patterns:

(1) Prefix outside parentheses # (_), # (_ , _), ...
(2) Prefix or infix in parentheses (# _), (# _ _), ...
(_ # _)
(_ # _ _), (_ _ # _), ...
(3) Postfix in parentheses (_ #), (_ _ #), ...
(4) Postfix without parentheses _ #, _ _ #, ...

Parameter lists may also have a fairly general structure, basically a parameter list may consist of several sublists, separated by semicolons, where each list may have a fixed or variable number of items, separated by commas. How syntactical patterns and the structure of parameter lists are made known to the parser is discussed in Section 4.3.

3. Descriptive and Executable Algebra

Before describing the Gral system architecture it is important to introduce the distinction between *descriptive* and *executable* algebra. In any Gral application a query is formulated as an expression of the application-defined many-sorted algebra. It should be clear that such a query is to be understood as the user's specification of some data object (relation or atomic object) that the system should return; this data object is described by a *hypothetical* series of applications of operators to the stored objects. It is the task of the optimizer to translate this query to an executable query plan and, in fact, to select among many available query plans the most efficient one.

In an extensible database system it is necessary that the application developer understands this "language of query plans" since obviously it must also be extensible. A central idea of the Gral system is that the language of query plans should also be an algebra, with the same syntactical structure, although with different sorts and operators, as the query language algebra. This simplifies the task of the application developer since he/she needs to understand only a single formalism; it is surely preferable to introducing a quite different way of describing query plans. The optimization process can then be understood (and partially be formulated) by the application developer as a translation from one algebra expression to another one. To distinguish, we call the query language algebra the *descriptive* and the "language of query plans" the *executable* algebra.

Whereas for the descriptive algebra the foremost design goals are expressive power and simplicity, and efficiency plays no role at all, the major issue in the design of an executable algebra is efficiency. On the other hand, because of the requirement of extensibility the executable algebra should still be as simple as possible, so a reasonable balance between efficiency and simplicity must be found.

Some important ideas to achieve efficiency at the level of the executable algebra are the following:

- Use index structures (access paths).
- Support processing in "tuple mode", that is, as far as possible perform a series of operations on a single tuple instead of copying the relation for each operation.
- Use complex operators. That means, whenever an operator needs to process a relation, it should do as much as possible during this access. For example, a sorting operator should include a selection condition to eliminate tuples before sorting, and allow to process each tuple further after sorting.

These are well-known strategies for the implementation of relational database systems. We show how they can be incorporated into the design of an executable algebra.

In the descriptive algebra a sort is associated with a set of objects and an operator with a function; neither is a representation specified for the objects nor an algorithm for computing the function. In contrast, in the executable algebra a sort corresponds to a specific object representation within the system and each operator has associated a fixed algorithm or procedure in the system. Generally for a Gral application system the set of sorts will consist of the following subsets and specific sorts:

- REL - each sort in REL describes one kind of physical representation (primary index) of a relation.
- INDEX - each sort describes one specific secondary index structure.
- TUPLE - a sort describing a tuple in memory.
- TIDSEQ - a sort describing a collection of tuple identifiers (through which tuples represented in one of the REL representations can be accessed).
- ATOM - these are the sorts for atomic objects that are also present in the descriptive algebra.

In the currently running Gral prototype there is only a single sort SREL in REL corresponding to a simple sequential representation of a relation. Primary and secondary index structures will be added soon. To illustrate the general approach we describe an executable algebra already containing index structures. In this design we have

REL = {SREL, STIDREL, BTREE}

SREL is a sequential representation used, for example, for temporary relations during the processing of a query. STIDREL is a permanent representation providing tuple identifiers; a relation represented in this way can be accessed sequentially or through a secondary index. BTREE is a B*-tree representation ordered by one of the attributes. It can be accessed through the primary index, through a secondary index, or sequentially. Let TIDREL = {STIDREL, BTREE} be the structures accessible through a secondary index.

INDEX = {SBTREE, SGRID, SMLGRID}

All of these structures allow to obtain a collection of tuple identifiers. SBTREE is a B*-tree used as a secondary index, SGRID a grid file [NiHS84], supporting geometric searches on a set of points in the plane, SMLGRID a multi-layer grid file [SiW88] supporting searches on a set of rectangles in the plane. The stored rectangles in an SMLGRID index are the bounding boxes of extended objects such as lines or polygons.

Sorts TUPLE and TIDSEQ are also present and we have

ATOM = {BOOL, INT, REAL, STR, POINT, LINE, PGON, RECT}

The executable algebra has an additional sort for rectangles which are needed as bounding boxes of LINE and PGON objects.

The set of atomic types with ordered domains is ORD = {BOOL, INT, REAL, STR}.

A subset of the operators of the current Gral executable algebra extended by some operators for index structures is shown in Figure 3-1.

REL	$\rightarrow$ SREL	***scan***, ***sort***	_ #
REL $\times$ REL	$\rightarrow$ SREL	***product***, ***smjoin***, ***inside_join***	_ _ #
REL	$\rightarrow$ ATOM	***extract***	(_ #)
TUPLE	$\rightarrow$ TUPLE	***sel***, ***proj***, ***extend***	_ #
BTREE(ORD_i) $\times$ ORD_i	$\rightarrow$ SREL	***exactmatch***	_ _ #
BTREE(ORD_i) $\times$ ORD_i $\times$ ORD_i	$\rightarrow$ SREL	***rangescan***	_ _ _ #
SGRID(POINT) $\times$ RECT	$\rightarrow$ TIDSEQ	***geo_range_search***	_ _ #
SMLGRID(RECT) $\times$ RECT	$\rightarrow$ TIDSEQ	***geo_intersection_search***	_ _ #
TIDSEQ $\times$ TIDREL	$\rightarrow$ SREL	***tidscan***	_ _ #
EXT	$\rightarrow$ RECT	***bbox***	# (_)
POINT $\times$ PGON	$\rightarrow$ BOOL	***pr_inside***	(_ # _)

Figure 3-1: Operators of the Executable Algebra

The executable algebra contains also atomic operations on standard types; these are as in the descriptive algebra. Also operators for SBTREEs have been omitted; they are similar to those on primary BTREEs but return sets of tuple identifiers.

The ***scan*** operator has a parameter list of the form [E_1; E_2] where E_1 is an algebra expression of result type BOOL and E_2 an expression of result type TUPLE. This operator scans a relation given in any of the REL representations. E_1 is a *filter* expression which is evaluated for each tuple of the operand relation; those tuples passing the filter are then made available to the second expression E_2 by the keyword *tuple* to which tuple operators can be applied. So E_2 allows processing in "tuple mode". For example, the query "List the names of big cities" looks in descriptive (**D**) and executable (**E**) algebra as follows:

D cities **select** [cpop > 500000] **project** [cname]
E cities ***scan*** [cpop > 500000; tuple ***proj*** [cname]]

The ***sort*** operator realizes the **ord** operator of the descriptive algebra but has also an input filter expression and a tuple expression applicable to tuples after sorting. Similarly the binary relation operators have two input filter expressions and a tuple expression to be applied to the concatenated tuples resulting from the operation. For example, the ***product*** operator computes the Cartesian product of the operand relations and has a parameter list of the form [E_1, E_2; E_3] where E_1 and E_2 are the filter expressions and E_3 is the tuple expression. It can be used to answer query (5) from Section 2 "Associate with each city the state containing it":

D cities states **join** [center **inside** region] **project** [cname, sname]
E cities states ***product*** [true, true; tuple ***sel*** [center ***pr_inside*** region] ***proj*** [cname, sname]]

The selection operator on tuples ***sel*** either returns the operand tuple or a special object called **notuple**; any tuple operator receiving **notuple** as an operand also returns

notuple. The same query could also have been realized using the ***inside_join*** operator. This operator can be applied to (a relation with) a set of points and (a relation with) a set of regions; it performs a plane-sweep [PrS85] to join tuples of the two operand relations where the point is contained in the region. We consider a slightly more complicated example query with this operator "List for big cities their population number as a percentage of their state's population":

D cities **select** [cpop > 500000] states
join [center **inside** region]
extend [(cpop/spop)*100 {percent}]
project [cname, percent]

E cities states ***inside_join*** [cpop > 500000, true;
center, region;
tuple ***extend*** [(cpop/spop)*100 {percent}]
proj [cname, percent]]

The ***smjoin*** operator performs a sort/merge join. Its parameter list has the form [E_1, E_2; E_3 cop E_4; E_5]. E_1, E_2 and E_5 are filter and tuple expressions, as before. E_3 and E_4 are expressions involving only attributes of the first and second operand relation, respectively, and *cop* is one of the comparison operators {<, ≤, =, >, ≥}. The two operand relations are first sorted by the value of their respective expression E_3 or E_4 and then merged according to the comparison operator. - The ***extract*** operator of the executable algebra has no parameters; it can be applied to a relation with a single tuple and a single attribute and returns the attribute value as an atomic object of the attribute's type (see the example below).

Access to primary or secondary index structures is also realized by executable operators. A BTREE is organized by attribute values of one of the ordered data types and can be accessed with a single value of the same type or a range of such values through the ***exactmatch*** and ***rangescan*** operators. These operators also have filter and tuple expressions like the ***scan*** operator to impose additional restrictions on the returned tuples and to process these tuples further. For the following examples we assume that relations and index structures are organized as follows:

cities:	BTREE(STR) ordered by the cname attribute
cpop_index:	SBTREE(INT) on the cpop attribute
center_index:	SGRID(POINT) on the center attribute
states:	BTREE(STR) ordered by the sname attribute
region_index:	SMLGRID(RECT) on the region attribute

The query "Retrieve the tuple representing the city Essen" can then be formulated as:

D cities **select** [cname = "Essen"]
E cities "Essen" ***exactmatch*** [true; tuple]

An SGRID index stores a set of points; the ***geo_range_search*** operator returns the tuple identifiers belonging to points inside a search rectangle given as the second operand. Similarly an SMLGRID index stores a set of rectangles; the ***geo_intersection_search*** operator returns the tuple identifiers belonging to rectangles intersecting a search rectangle. The returned tuple identifiers can be processed further through the ***tidscan*** operator which accesses the original relation and allows to filter and process tuples. As an example we consider the query "Find cities in Bayern!":

D (states **extract** [sname = "Bayern"; region])
{Bayern};
cities **select** [center **inside** Bayern]

E (states "Bayern" ***exactmatch*** [true;
tuple ***proj*** [region]] ***extract***) {Bayern};
center_index ***bbox*** (Bayern) ***geo_range_search***
states ***tidscan*** [center ***pr_inside*** Bayern; tuple]

Let us reconsider the executable algebra from the point of view of extensibility. The Gral kernel system will contain generally useful relation representations such as those given above in REL and, say, SBTREEs as a secondary index structure, as well as sorts TUPLE, TIDSEQ, and BOOL. It will have executable operators needed to implement the descriptive operators of the model frame, for example ***scan***, ***sort,*** ***smjoin***, ***product***, ***extract***, ***exactmatch***, ***tidscan***, etc. Standard types such as INT, REAL, STR together with their operations will be available as a module. A Gral application system may then add its own special types such as POINT, LINE, PGON, and provide operators for these types as well as special index structures such as SGRID, SMLGRID with suitable operators for searching these structures.

This distinction between Gral kernel system and Gral application system arises from a practical point of view. Note, however, that the Gral architecture to be described next is *absolutely uniform* with respect to *all* sorts and operators. For example, ***scan*** and ***smjoin*** are brought into the system through the same extension mechanism as any other operator and they might as well be application-defined. Parser, query evaluator, and optimizer handle these operators in the same way as any application-defined operator. We believe that only in this way a clean architecture can be achieved.

4. An Extensible System Architecture

4.1. Survey

The Gral system prototype has been implemented during the last two years in a project at the University of Dortmund. The implementation language is Modula-2; the system runs on Sun workstations under UNIX. The interactive interface uses SunWindows and SunCGI for window management and graphic representation of geometric objects; this interface is mostly written in C. The system currently offers the following functionality:

- Database operations: create, destroy, open, close, save a database, that is, a collection of relations.
- Relation operations: create, destroy a relation; insert a tuple into a relation.
- Load/unload: move a relation out of the system into a UNIX text file and load it from such a text file. This is currently the only way to change the contents of a relation (through a standard text editor).
- Show: display the contents of a relation. Attribute types that are textually representable are shown in a text window; types that are graphically representable (such as geometric types) are shown in a graphics window. The text window shows tuples and their attributes sequentially; one can scroll forward or backward or access directly any position within this textual representation of the relation. The graphics window shows a section of the real cartesian plane which can be selected by the user; geometric objects occurring as attribute values of the displayed relation are visible as far as they intersect this

section. It is possible to overlay the graphical representations of several relations.

- Query: enter a query either in descriptive or in executable algebra. The resulting relation or atomic object is then displayed. The descriptive algebra implemented so far is a subset of the geo-relational algebra described in [Gü88a].

Note that the interactive interface offering this functionality is entirely *data type independent*. For example, the program that displays a relation does not know how to display any attribute value. Instead, it calls through the *type manager* a type-specific procedure to display a value of this type which was provided by the implementor of this type. So this program is prepared to handle any attribute type that will be added to the system in the future. According to the philosophy of the Gral model frame the only object type built into the system is that of relations.

In this section we focus on the *layered architecture for query evaluation* and emphasize and explain extensibility at the various layers. This architecture is shown in Figure 4-1.

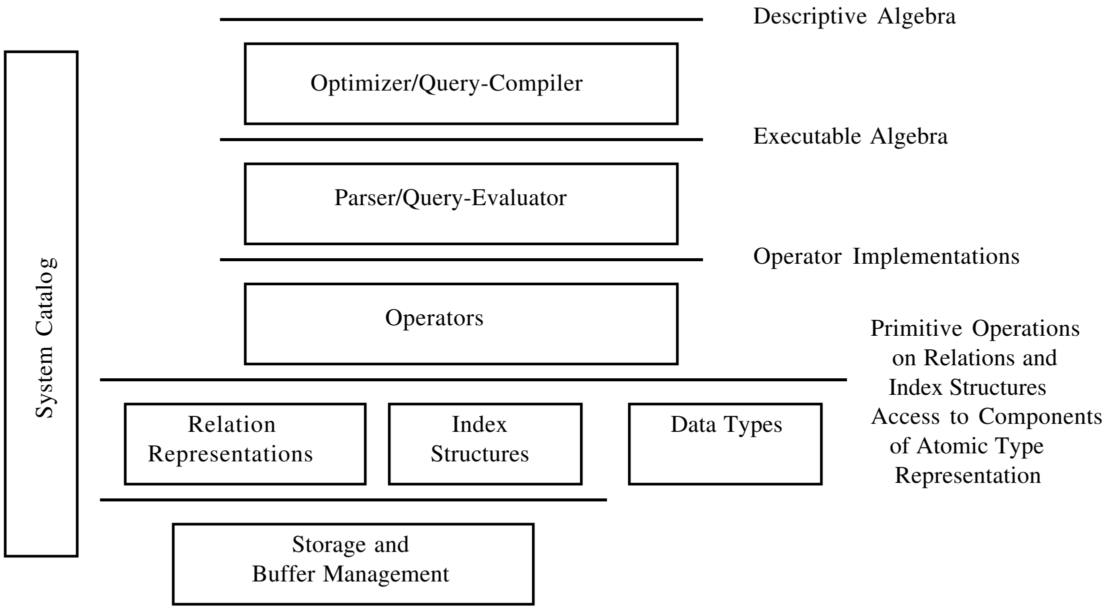

Figure 4-1: Architecture for Query Evaluation

Although query processing proceeds top-down, the architecture of the system is best explained bottom-up. The bottom layer provides rather traditional *storage and buffer management* except that page sequences are available to hold data objects occurring as attribute values (it is generally assumed that "atomic" data objects may have arbitrary size) and facilities for fixing page sequences in the buffer. Currently we see no need for this layer to be extensible. The second layer provides *representations* for the sorts of the executable algebra; the implementation of relation representations and index structures uses the functions offered by the storage management layer. The second layer provides also a higher level representation of a tuple in memory with suitable operations and a scan component offering uniform scanning operations for all relation representations. This layer is extensible by atomic data types, index structures, and relation representations.

The third layer consists of *implementations of executable operators*. These are based on the primitive access operations offered by relations or index structures and the representation of atomic types realized by the layer below. For example, the ***scan*** operator of the executable algebra uses the scan component of the representation layer. A "point inside polygon" test operator (***pr_inside***) accesses the representations of point and polygon objects implemented in that layer. Each executable operator is implemented by two procedures. One of these describes the *schema transformation* (ST) performed by the operator (for example, the ***product*** operator creates a new relation schema by concatenating the two operand schemas). The second procedure implements the *occurrence transformation* (OT), that is, the mapping of the relation instances or atomic data objects. There are two separate procedures because the ST procedure is already called during the parsing of an executable algebra expression whereas the OT procedure is called during query evaluation, after parsing is completed. This layer is, of course, extensible by operator implementations.

The next layer contains the *parser* and *query evaluation* components. Given an executable algebra expression, the parser produces an operator tree representation in memory. For this purpose it accesses information in the system catalog, in particular the "Operators" table which contains for each operator its name (or symbol), the functionality, and a description of the parameter list structure, if there is one. As mentioned before, the parser also calls an operator's ST procedure to apply the operator to an operand schema (which may in turn be the result of parsing a subexpression). The parser has no built-in knowledge of any operator. The same is true for the query evaluator. This component "understands", however, the structure of the operator trees which the parser builds. The query evaluator traverses the operator tree, computes bindings for the leaves, representing objects, and calls for each operator represented by an internal node the corresponding OT procedure. The query evaluator takes (references to) the operand objects from the tree and passes them to the procedure as parameters; it receives the resulting object from the procedure and links it into the tree. This layer is extensible by operator descriptions being entered into the system catalog.

Up to this layer we have a complete system capable to evaluate queries in executable algebra. As mentioned before, in the Gral system this capability is present at the user interface, that is, a user can enter queries in executable algebra. It is necessary to expose this interface to an application developer so that he/she can test new operator implementations in queries for correctness and performance. The knowledge gained in performance tests can subsequently be built into optimization rules.

The top layer consists of the *optimizer/query compiler* component. The optimizer calls the parser of the layer below to analyze a given descriptive algebra expression and to produce an internal tree representation; the parser again receives the necessary information from the "Operators" table in the system catalog. The same parser can be used because descriptive and executable algebra have the same syntactical structure. The information needed for optimization and query translation itself comes from a text file "OptimizationRules". This file contains various classes of optimization/translation rules in a specific syntax and order. When the Gral system is started the contents of the file are analyzed and represented by internal data structures so that patterns occurring in rules can be matched efficiently against parts of the tree representing the query. How the optimizer works is described in some more detail in Section 4.4. The result is an expression in executable algebra which can be shown in text form to the user, if desired. For further processing of the query, however, it is passed to the query evaluator directly in internal form, so it need not be parsed again. The top layer is extensible by descriptive operators and optimization/translation rules.

In the following three subsections extensibility by data types, executable operators, and descriptive operators is discussed in a little more detail.

4.2. Extensibility by Data Types

The implementation of a data type in Gral consists of:

(1) A Modula-2 type, usually opaque, together with operations for building an object accossiated with a variable of this type, and operations to access the components of an object. This part constitutes the Modula-2 implementation of an abstract data type.

(2) Three classes of procedures:

I Procedures for storing and loading an object. These convert the object representation in a variable to a bytestring and store the object as an attribute value in a relation and vice-versa. Whenever possible the bytestring representation produced by the Modula-2 compiler is used so that no conversion is necessary. Optionally there may be procedures to load an object only partially.

II Procedures for interactive input/output. For example, a procedure to read a string value from the keyboard or a procedure to display a polygon on the graphics screen.

III Procedures to convert from/to textual form.

There exist two different kinds of programs using a data type: programs written *before* the type was implemented and that do not "know" the type, and programs written *afterwards* with knowledge about the type. An example of the first class is the program displaying a relation at the screen. An example of the second class is a program testing two polygons for intersection. Programs of the first class access a type through the *type manager*. This module provides a generic procedure for each task within classes I - III above; the generic procedure just switches to the type-specific procedure provided by the type implementation. The generic procedure is implemented in Modula-2 by an **array of procedure** indexed by the type.

Hence, to add a data type to the Gral system, an AD implements the type as described under (1) and (2) above, usually in a module for the type. Then the type manager module is extended: the procedures in the type module are imported and for each generic procedure in the type manager the corresponding component of its array of procedure is assigned the type's procedure. Finally a procedure "InsertType" is called to enter the name of the type and some other information (e.g. whether the type is sortable, can be displayed textually, graphically, etc.) into the system catalog and the system relation about types.

Similar ideas for providing type extensions have been reported in [StRG83, Ong84, OsH86] although the details differ. For example, in Gral types are not defined within the system's data definition language to avoid inconsistencies with the source code which must be present in the system anyway. What is perhaps new about our treatment of types is the possibility of *partial loading*. Since we generally assume that objects may be arbitrarily large it is useful to allow to load only some part of an object into memory. For example, a polygon is represented by a record with fields for the bounding box, the area, the perimeter, and the vertices (given as an array). An operator testing two polygons for intersection may initially only load the bounding boxes and only load the vertices later if the bounding boxes intersect.

4.3. Extensibility by Executable Operators

Adding an executable operator to the system consists of three steps. First, one implements its schema and occurrence transformation procedures, using the primitives of the representation layer. (That layer contains also a module “SchemaTransformations” providing a data structure to represent a schema and operations on schemas such as concatenating schemas, adding an attrribute, etc.). The new operator then needs to be linked into the “OpManager” module. The OpManager has separate arrays of procedure for ST and OT procedures, one array for each arity of an operator. So the second step is to assign the new procedures to appropriate array components. Through the OpManager these procedures can then be called by parser and query evaluator.

The third step in adding an operator is to make its name, functionality, syntax, and parameter list structure known in the system catalog where this information is then available for the parser. Similar as for types this is achieved by calling a procedure “InsertOp” with text parameters containing corresponding descriptions. The system catalog (and the system relation about operators) has four entries for each operator:

(1) Operator name or symbol.

(2) A description of functionality and syntax in the form *pattern* $\rightarrow$ *result type*. For example, for the ***scan*** and ***pr_inside*** operators the descriptions are:

```
REL scan -> REL
(POINT pr_inside PGON) -> BOOL
```

(3) A description of the parameter list if there is one, otherwise the word “None”. Parameter lists may be either in *standard form* or *irregular*. To be in standard form means that a parameter list may be structured into a fixed number of *sublists*, separated by “;”, each sublist may have a fixed or variable number of *items*, separated by “,”. An item may have a fixed number of parts, separated by blanks or user-defined delimiters. Parts may be recognizable for the system parser or not. Recognizable parts are constants of some atomic type, attribute names, (re)namings, and algebra expressions of specified result type. In Gral these rules are built into a grammar which defines the allowed parameter list descriptions and thus the allowed parameter lists. Parameter list descriptions for ***scan*** and ***proj***, for example, are as follows:

scan: `[Expr(BOOL); Expr(TUPLE)]`
proj: `[Attrname *]`

If a parameter list is irregular, a procedure for parsing the parameter list must be provided by the AD with the operator implementation. In the current Gral system this was never necessary.

(4) The last entry is either “Executable” or “Descriptive” since descriptive operators are described in exactly the same way and appear also in the operator table.

The parser transforms a parameter list encountered in an algebra expression according to the description in (3) to a nested list data structure and connects this representation to the operator's node in the operator tree. When a parameter list contains an algebra expression, the parser recursively parses this expression and builds an operator tree which it links into the nested list structure. In this way nested algebra expressions are represented in the operator tree. During query evaluation, the nested list representation of a parameter list is passed to the operator's OT procedure. The implementor of this procedure (the AD) relies on the parser having built the right structure; thus the description (3) serves as an interface definition between parser and OT procedure. When an OT procedure has to process an algebra expression occurring as a parameter it simply calls the query evaluator on the tree representation of this expression. This is an important point: A user-defined OT procedure calls the system's query evaluator to handle nested algebra expressions, as they occur, for example, in ***scan*** or ***extend*** operators.

4.4. Extensibility by Descriptive Operators and Optimization/Translation Rules

In this section we briefly sketch how the optimizer/query compiler works and how rules are described and arranged in the file “OptimizationRules”; a detailed report is to follow [BeG89]. The translation of a given descriptive algebra expression proceeds in the following stages:

---*Descriptive multistep query*---

1. *Normalize:* In a multistep query names of objects resulting from expressions can be replaced by these expressions without changing the semantics. This is done to obtain a single expression which can be optimized further as a whole.

---*Descriptive expression*---

2. *Decompose:* Certain operators, such as selection, join, extend, are decomposable. For example, a selection with several and-connected conditions can be transformed into several selections with simple conditions. Decomposable operators are taken apart, so that their parts (e.g. simple selections) can move freely.
3. *Improve:* Change the order of operators according to heuristics (classical algebraic optimization).
4. *Compose:* Put subsequent operators together (for example, a series of selections).
5. *Eliminate* constant and common subexpressions.

---*Final descriptive multistep query*---

For each expression in the multistep query:

---*Descriptive expression*---

6. *Translate* descriptive subexpressions to executable subexpressions. (During this stage, the optimizer works on a mixed descriptive and executable expression.)

---*Executable expression*---

7. *Combine* sequences of executable operators into a single operator.

---*Final executable expression (and multistep query)*---

All stages except 1 and 5 are supported by rules; these are given in the file OptimizationRules which is structured roughly according to the stages into sections. In the first stage a query formulated by a user in several steps is transformed (by embedding expressions for names) into a single expression (*normal form*) so that optimization of the whole query is possible. The steps selected by the user might otherwise prevent certain optimizations. Stage 2 (Decompose) is supported by a corresponding section in OptimizationRules:

```
DECOMPOSING RULES
…
RULE
  REL_1 select [BOOL_1 and BOOL_2]
  -> REL_1 select [BOOL_1] select [BOOL_2].

RULE
  REL_1 REL_2 join [BOOL_1]
  -> REL_1 select [BOOL_1] REL_2 product
     if BOOL_1.attr c REL_1.attrnames,
  -> REL_1 REL_2 select [BOOL_1] product
     if BOOL_1.attr c REL_2.attrnames.
…
END
```

A rule consists in general of a pattern followed by one or more replacement patterns each of which may have an associated condition. The condition may involve properties of the expression, such as the attribute names occurring, and properties of the involved objects, such as the number of tuples in an operand relation or the physical sizes of attributes or atomic objects, etc. A rule may be preceded by a specification part to simplify notation within the rule. Each rule described in this way can be viewed as a collection of rules of a production system [Ni80]. For most stages an irrevocable control strategy is used and rules are applied until no further rule is applicable. In these stages the first replacement pattern whose condition is fulfilled is selected to replace the original pattern in the expression. Different control strategies which explore alternatives are used in stages 3 and 6.

The goal of stage 3 is to arrange operators in an order which is "good" according to heuristics. For this purpose it is possible to specify a partial order on descriptive operators to indicate a *desired* order of operators in the expression - whether that order can be achieved is another matter. This is a general facility to formulate such familiar heuristics as "selection and projection should be done as early as possible". The partial order is given by a section "OPERATOR ORDER". How operators may actually be exchanged to achieve a good order is described in the section "IMPROVING RULES", one example is shown:

```
SPEC
  op_1 in {select, project, ord};
RULE
  REL_1 op_1 [Param_1] select [BOOL_1]
  -> REL_1 select [BOOL_1] op_1 [Param_1].
```

The control strategy for stage 3 attempts to arrange operators according to the partial order in such a way that the number of "runs" (ordered subsequences) is minimized. By a suitable definition of operator order and rules, one can let a run correspond to a single operator of the executable algebra and so minimize the number of executable operators needed. The section "COMPOSING RULES" contains rules for stage 4 (no example is shown). Together, stages 2 - 4 realize the classical algebraic optimization.

No rules are necessary to support stage 5 (Eliminate). A constant subexpression is one that is evaluated once for each tuple of a relation but does not depend on the tuple's attribute values. In query (2) of Section 2 "(cities **max** [cpop])" is a constant subexpression. Stage 5 translates this to a two-step query where the constant subexpression is evaluated only once initially:

(cities **max** [cpop]) {Constant_1};
cities **extend** [cpop/Constant_1 {percent}]

The next section contains rules for the translation of descriptive to executable subexpressions. This is the point where descriptive operators are associated with their executable counterparts.

```
TRANSLATING RULES
…
RULE
  REL_1 select [Attrname_1 = Const_1]
  -> REL_1 Const_1 exactmatch [true; tuple]
     if ReprIs (REL_1, Attrname_1, BTREE),
  -> Index (REL_1, Attrname_1) Const_1
     indexexactmatch REL_1 tidscan [true; tuple]
     if ExistsIndex (REL_1, Attrname_1, SBTREE),
  -> REL_1 scan [Attrname_1 = Const_1; tuple].

RULE
  REL_1 REL_2 join[Attrname_1 inside Attrname_2]
  -> REL_1 REL_2 inside_join [true, true;
         Attrname_1, Attrname_2; tuple]
     if Attrname_1 in REL_1.attrnames and
        Attrname_2 in REL_2.attrnames and
        REL_1.size > 100,
  -> REL_1 REL_2 product [true, true; tuple sel
          [Attrname_1 pr_inside Attrname_2]].
…
END
```

The first rule describes how a simple selection can be implemented either by access through primary or secondary index structures, if present, or otherwise by a relation scan. The second rule shows how the **inside** operator of the descriptive algebra, defined on atomic objects, can actually be implemented by a plane-sweep algorithm which is (under certain conditions) the most efficient way to realize this kind of geometric join. This example shows how the system architecture does indeed support the efficient processing of geometric queries, which was one of the initial goals of the Gral development. The rule is also a simple example of how the optimizer can switch between various implementations: for small sets of points (here up to 100 points) forming the cartesian product with subsequent selection is preferable.

The last section "COMBINING RULES" of the file OptimizationRules contains rules for combining executable operators. The previous translation stage which replaces descriptive operators independently of each other by executable operators may, for example, have produced series of ***scan*** operators, or ***scan*** followed by ***product***, etc., which can now be combined into one:

```
RULE
  REL_1 REL_2 product [BOOL_1, BOOL_2; TUPLE_1]
     scan [BOOL_3; tuple]
  -> REL_1 REL_2 product [BOOL_1, BOOL_2;
         TUPLE_1 sel [BOOL_3]].
```

We have given here only a rough survey of the Gral extensible optimizer; most details and the more sophisticated techniques have been left out. For example, the optimizer can be advised by rules to search systematically all permutations of n and-connected simple conditions, within a selection or join, say, and for the translation stage (6) each replacement pattern in a rule can be assigned a numeric expression. The replacement pattern can then be selected according to the value of that expression. These facilities allow one to search the possibilities for implementing a single operator, such as a complex selection, exhaustively, and to make a good choice among these. For example, a simple condition supported by an index structure will be evaluated first. These techniques are described in more detail in [BeG89].

It is now obvious how a descriptive operator is added to the system: Its name, functionality and syntax, and parameter list structure are entered into the system catalog as for executable operators. The descriptive operator is then "implemented" by a collection of rules in the file OptimizationRules. Of course, rules can be added to or removed from the file at any time independently from adding operators.

5. Related Work

We have described the query processing architecture of an extensible relational database system based on many-sorted algebra and shown some of the specific extension mechanisms used in the Gral system. Obviously much of the previous work on extensible database systems is related in some way; some of it can be associated with specific layers of our architecture. Extensibility by relation representations and index structures and in particular the interaction between these two in update processing have been studied in [LiMP87]; their work fits nicely into our representation layer. Methods for adding atomic types, functions, and predicates have been studied in [StRG83, OsH86]. The RAD system [OsH86] uses also a query language based on relational algebra and in principle allows relation mappings (called "transformations"). However, these have not been implemented and no extension mechanisms are described. Techniques for adding operator implementations have also been studied in [St86, GrD87]. In contrast to our approach, in the INGRES [St86] or POSTGRES [StR86] project the architecture of the database system is not affected as a whole by extensibility. That is, there is a collection of built-in types which are treated differently from user-defined types; also all the standard parts of the query language are implemented in a traditional way and user-defined operations are handled separately. These are essentially restricted to simple functions on atomic types; relation mappings are not foreseen. The approach of adding interfaces to a traditional architecture shows in the way how user-defined operations need to specify whether certain methods of the fixed part of the system are applicable, e.g. whether hash-join is feasible for this operator. In Gral, hash-join could in principle as well be a user-defined operation.

Rule-based optimization has also been studied by Freytag [Fr87] and in the EXODUS project [GrD87]. Our techniques have some similarity to Freytag's proposal. Differences are that he describes only the translation stage, so there is no algebraic optimization before or after translation, and that his rules describe how to generate all possible query execution plans but no strategy for selecting a good one. Gral contains a complete extensible optimizer which actually produces a good query plan. Generally one could say that we have filled in many details to his proposal and implemented a relatively complete query language including aggregate functions, sorting, and geometric operators. Freytag considers essentially only selection and join. Graefe and DeWitt [GrD87] focus on the aspect of heuristically searching the space of query plans to find a good one quickly. In their relational test implementation they also consider only selection and join. Another approach to extensible query optimization using the rules of a grammar to construct query plans is described in [Lo88]. - We shall compare the methods for extensible optimization in more detail in [BeG89].

Acknowledgment

Past and present members of the Gral development team are Ludger Becker, Thaddäus Behnke, Anja Huesmann, Dirk Markert, Rainer Matenia, Stefan Pölt, Markus Schneider, Norbert Suchhart, Jacqueline Teschner, Reinhilde Uphaus, Bernd Vielhauer and Leonhard Wawrzinek. Their contribution to the design and implementation of the Gral prototype is gratefully acknowledged.

References

[Bato86] Batory, D., J. Barnett, J. Garza, K. Smith, K. Tsukuda, C. Twichell, and T. Wise, GENESIS: A Reconfigurable Database Management System. University of Texas at Austin, Dept. of Computer Science, Report TR-86-07, 1986.

[BeG89] Becker, L., and R.H. Güting, An Optimizer for an Extensible Geometric Database System. Universität Dortmund, Fachbereich Informatik, Manuscript, 1989.

[Care86] Carey, J.M., D.J. DeWitt, D. Frank, G. Graefe, M. Muralikrishna, J.E. Richardson, and E.J. Shekita, The Architecture of the EXODUS Extensible DBMS. In [Di86], 52-65.

[CaDV88] Carey, J.M., D.J. DeWitt, and S.L. Vandenberg, A Data Model and Query Language for EXODUS. Proc. ACM SIGMOD 1988, 413-423.

[Co79] Codd, E.F., Extending the Database Relational Model to Capture More Meaning. *ACM Transactions on Database Systems 4 (1979)*, 397-434.

[Daya87] Dayal, U., F. Manola, A. Buchman, U. Chakravarthy, D. Goldhirsch, S. Heiler, J. Orenstein, and A. Rosenthal, Simplifying Complex Objects: The PROBE Approach to Modelling and Querying Them. In: H.J. Schek and G. Schlageter (eds.), Proc. BTW 87, Springer 1987, 17-37.

[Di86] Dittrich, K.R., Object-Oriented Database Systems: the Notion and the Issues. Proc. of the IEEE/ACM International Workshop on Object-Oriented Database Systems, Pacific Grove, California (September 1986), 2-4.

[Fr87] Freytag, J.C., A Rule-Based View of Query Optimization. Proc. ACM SIGMOD 1987, 173-180.

[GoTW78] Goguen, J.A., J.W. Thatcher, and E.G. Wagner, An Initial Algebra Approach to the Specification, Correctness, and Implementation of Abstract Data Types. In: R. Yeh (Ed.), Current Trends in Programming Methodology, Vol. IV, Prentice-Hall 1978, 80-149.

[GrD87] Graefe, G., and D.J. DeWitt, The EXODUS Optimizer Generator. Proc. ACM SIGMOD 1987, 160-172.

[Gr84] Gray, P.M.D., Logic, Algebra, and Databases. Ellis Horwood Ltd., Chichester, 1984.

[Gü88a] Güting, R.H., Geo-Relational Algebra: A Model and Query Language for Geometric Database Systems. In: J.W. Schmidt, S. Ceri, M. Missikoff (eds.), Proc. EDBT 1988, 506-527.

[Gü88b] Güting, R.H., Modeling Non-Standard Database Systems by Many-Sorted Algebras. Universität Dortmund, Fachbereich Informatik, Report 255, 1988.

[GüZC88] Güting, R.H., R. Zicari, and D.M. Choy, An Algebra for Structured Office Documents. Universität Dortmund, Fachbereich Informatik, Report 254, 1988, to appear in *ACM Transactions on Office Information Systems (Jan. 1989).*

[HaL82] Haskin, R.L., and R.A. Lorie, On Extending the Functions of a Relational Database System. Proc. ACM SIGMOD 1982, 207-212.

[LiMP87] Lindsay, B., J. McPherson, and H. Pirahesh, A Data Management Extension Architecture. Proc. ACM SIGMOD 1987, 220-226.

[Lo88] Lohman, G.M., Grammar-like Functional Rules for Representing Query Optimization Alternatives. Proc. ACM SIGMOD 1988, 18-27.

[MaD86] Manola, F., and U. Dayal, PDM: An Object-Oriented Data Model. In [Di86], 18-25.

[NiHS84] Nievergelt, J., H. Hinterberger, and K.C. Sevcik, The Grid File: An Adaptable, Symmetric Multikey File Structure. *ACM Transactions on Database Systems 9 (1984)*, 38-71.

[Ni80] Nilsson, N.J., Principles of Artificial Intelligence. Tioga Publ. Company, Palo Alto, CA, 1980.

[Ong84] Ong, J., D. Fogg, and M. Stonebraker, Implementation of Data Abstraction in the Relational System INGRES. ACM SIGMOD Record 14, No. 1 (March 1984), 1-14.

[OsH86] Osborn, S.L., and T.E. Heaven, The Design of a Relational Database System with Abstract Data Types for Domains. *ACM Transactions on Database Systems 11 (1986)*, 357-373.

[PiT86] Pistor, P., and R. Traunmüller, A Database Language for Sets, Lists, and Tables. *Information Systems 11 (1986)*, 323-336.

[PrS85] Preparata, F.P., and M.I. Shamos, Computational Geometry: An Introduction. Springer 1985.

[ScS86] Schek, H.J., and M.H. Scholl, The Relational Model with Relation-Valued Attributes. *Information Systems 11 (1986)*, 137-147.

[SiW88] Six, H.W., and P. Widmayer, Spatial Searching in Geometric Databases. Proc. IEEE Data Engineering Conf. 1988, 496-503.

[St86] Stonebraker, M., Inclusion of New Types in Relational Data Base Systems. Proc. 2nd Intl. Conf. on Data Engineering (Los Angeles, CA, February 1986), 262-269.

[StR86] Stonebraker, M., and L.A. Rowe, The Design of POSTGRES. Proc. of the 1986 SIGMOD Conf. (Washington, DC, May 1986), 340-355.

[StRG83] Stonebraker, M., B. Rubenstein, and A. Guttmann, Application of Abstract Data Types and Abstract Indices to CAD Databases. Proc. 1983 ACM Engineering Design Applications, 107-114.

[Schw86] Schwarz, P., W. Chang, J.C. Freytag, G. Lohman, J. McPherson, C. Mohan, and H. Pirahesh, Extensibility in the Starburst Database System. In [Di86], 85-92.