Algebraic Support for Complex Objects with Arrays, Identity, and Inheritance

Scott L. Vandenberg
David J. DeWitt

Computer Sciences Department
University of Wisconsin
Madison, WI 53706

ABSTRACT — Algebraic query processing and optimization for relational databases is a proven and reasonably well-understood technology. Recently the algebraic approach has been extended to more advanced data models (nested relations, complex objects, object-oriented systems). Here we continue this evolution by presenting novel algebraic operators and transformations supporting grouping, arrays, references, and multisets. We also propose a new approach to processing and optimizing overridden methods in the presence of multiple inheritance. The utility of both the algebraic operators and the transformation rules is demonstrated with examples. Object identity is incorporated into the algebraic domains, giving an original, intuitive set-theoretic semantics for the domains of object identifiers in the presence of multiple inheritance. We prove that the algebra is equipollent to the QUEL-like user-level query language and discuss some other expressiveness issues.

1. Introduction

The relational model of data [Codd70] has been very successful both commercially and in terms of the research opportunities it has provided. One of the major reasons for this is that the model lends itself to an execution paradigm that can be expressed as an algebra [Codd70, Ullm89]. An algebraic execution engine is used to process queries and to optimize them by rewriting algebraic expressions into different algebraic expressions that produce the same answer in a (hopefully) more efficient manner. Algebraic implementation/optimization techniques are well-understood and algebraic specifications of data retrieval languages lend themselves to theoretical examination in terms of expressiveness and other issues. In recent years it has become apparent that the relational model is not always the right choice for a particular application [Care86], and many new data models have been proposed [Bane87, Daya89, Peck88, Sche86, Kupe85, Abit88]. Thus an important open question is this: Exactly how far can the algebraic approach take us?

1.1. Contributions and Relation to Previous Work

This paper provides a partial answer to this question by describing a viable algebraic query processor/optimizer for an advanced data model. The EXTRA/EXCESS system [Care88a] supports the following advanced features that are of interest here: types and top-level database objects of completely arbitrary structure, multisets, arrays, methods (written in the EXCESS query language) defined on any type, multiple inheritance of tuple attributes and methods, and a form of object identity that allows (but does not force) any part of any structure to have identity separate from its value. None of these issues is satisfactorily addressed in other algebras. This paper demonstrates that these constructs can be managed algebraically. The algebra contains the following original features, which we discuss in more detail below: a type constructor-based approach to defining the operators; operators and transformations for multisets and arrays; support for object identity via a new type constructor with associated operators; support for grouping; domain definitions that give a clear semantics to domains involving object identifiers (OIDs) and arrays; and several alternatives for processing queries involving overridden methods. Another interesting contribution is the proof that EXCESS and the algebra are equipollent (equivalent in expressive power). Most such proofs are between algebras and calculi; we omit a calculus and prove direct correspondence with a user-level query language. There are literally dozens of database algebras, so the following paragraphs are only a brief sketch of the EXCESS algebra's relation to them; [Vand91] has a more complete treatment.

The algebra's novel features enable it to successfully model the structures of EXTRA (the DDL) and process the queries of EXCESS (the DML). One such feature is the "many-sorted" nature of the algebra. This means that the algebraic structures need not all have the same type (or "sort"), as is the case with other database algebras, which require all database objects, and thus all query inputs and outputs, to be sets. Our relaxation of set-orientation means that we do not need to model a real-world entity as a set if it is not really a set. This allows more natural algebraic representations of some entities. The many-sortedness allows us to easily model the arbitrary structure of EXCESS types. The algebra of [Guti89] is many-sorted in the sense that arithmetic is part of the database algebra, but the portion of the algebra corresponding to the usual notion of database algebras is not many-sorted, giving it a much different flavor than the EXCESS algebra. [Daya89] presents a version of relational algebra in which every operation can apply to a single tuple as well as to a set of tuples, but there is no notion of separating operators into groups appropriate for a particular type constructor. The constructs of [Beer90] provide a many-sorted flavor in that they will not always work for all type constructors, but there is still no concept of different operators for different type constructors.

This research was partially supported by the Defense Advanced Research Projects Agency under contract N00014-85-K-0788, by an IBM Graduate Fellowship, and by a donation from Texas Instruments.

Permission to copy without fee all or part of this material is granted provided that the copies are not made or distributed for direct commercial advantage, the ACM copyright notice and the title of the publication and its date appear, and notice is given that copying is by permission of the Association for Computing Machinery. To copy otherwise, or to republish, requires a fee and/or specific permission.
© 1991 ACM 0-89791-425-2/91/0005/0158...$1.50

Another unique feature of our algebra is its complete algebraic treatment of multisets (sets that allow duplicates). We provide original operators for additive union, grouping, set creation, and looping, among others. The transformation rules involving these new operators are, of course, also new, as are most of the other multiset transformations. ENCORE [Shaw89] and ALGRES [Ceri87] provide both sets and multisets, but the semantics of their multiset operations are not specified. Some of our multiset operators are similar to those of [Daya82], but [Daya82] restricts itself to the relational model and includes the redundant intersection operator. The SET_APPLY looping operator described below resembles other algebraic operators [Abit88, Guti89, Daya89], but is unique in that it allows the application of any algebraic expression to the elements of the multiset and needs no special syntax to apply the expression.

Several of the array operators are new and none of the array transformation rules have been presented elsewhere. There is an array looping operator similar to SET_APPLY. The ARR_EXTRACT operator extracts a single element from an array, and the result is not an array containing the element but simply the element itself. There is also an array creation operator and an operator to collapse an array of arrays. The notion of sequences supported in [Guti89] is similar, but not identical, to our notion of arrays. They do not support unordered sets or fixed-length arrays. Also, our operators can be used in such a way that the ordering properties of the arrays can either be preserved or not, depending on the requirements of the query.

Object identity is supported by introducing a new type constructor, called "ref". This constructor-based approach allows the algebra to mix object- and value-based semantics at will. We introduce two new operators and transformations involving them. This allows us to treat OIDs as values in the domain of the algebra, enabling a simple, intuitive, and original set-theoretic semantics to be imposed on them. These domain definitions are new. This treatment also enables the algebra to be defined using only one form of equality, instead of one form for OIDs and one for values, as is done in [Shaw89, Osbo88]. [Abit89] defines two separate languages, one forcing object identity on all database objects and one not supporting it at all, but we mix the two semantics in a single algebraic language, and give references (i.e., object identifiers that support sharing, etc.) the status of a type constructor with the same privileges as the multiset, array, and tuple constructors. This language also supports least fixpoint operations; EXCESS does not. LDM [Kupe85] forces object identity on everything in the database.

A new approach for processing queries involving overridden methods is proposed. The problem being addressed is that of determining the appropriate method to invoke when we do not know the exact type of an entity until run time (due to inheritance). We explore tradeoffs between this method and a more "obvious" approach.

Algebraic transformation rules for the relational algebra can be found in [Ullm89]. [Scho86] presents such rules for the nested relational model. Some rules for complex object models with identity are proposed in [Osbo88, Shaw89]; these rules are mainly straightforward extensions of relational or nested relational transformation rules. [Beer90] proposes a meta-level algebra for collections of complex objects with identity and includes some transformation rules that go beyond what is done in the relational model. This algebra, however, does not correspond to a specific data model but rather to a higher-level notion of collections of objects. It also does not support several of the constructs of EXTRA. We provide dozens of transformation rules designed specifically for this system [Vand90b].

The algebra and transformation rules specified here are the basis for an optimizer for the EXCESS query language. EXTRA/EXCESS is being implemented using the EXODUS extensible DBMS toolkit [Care88b], and the optimizer is being built using the EXODUS optimizer generator [Care88b]. In addition to forming a useful optimizer for this system, the techniques (operators and transformation rules) are applicable to other systems that support similar constructs but do not do so algebraically (e.g. [Bane87, Zani83]). The primitive nature of the algebraic operators allows other operators to be defined in terms of them quite readily. This will result in the ability to test a wide variety of algebraic operators for utility and optimizability. Furthermore, the existence of such an optimizer will enable research into statistics and cost functions for advanced data models.

1.2. Organization

The remainder of the paper is organized as follows: Section 2 briefly describes the EXTRA DDL and EXCESS DML, concentrating on the features that are relevant to the algebra as described here. Section 3 defines and motivates the algebra's structures and operators and demonstrates its equipollence to EXCESS, again emphasizing new operators and old operators with new characteristics. This section does not establish or discuss the algebra's equipollence (or lack thereof) to any other algebra, although we make some general comments on how this might be done in the future. Some algebraic alternatives for processing queries involving overridden method names are presented and discussed in Section 4. One of these alternatives is original and widely applicable. In Section 5 we illustrate, by example, some of the more interesting new transformation rules of the EXCESS algebra. Section 6 draws some conclusions, outlines current and future work on the system, and reports on its implementation status. A partial list of the novel algebraic transformation rules for EXCESS can be found in [Vand90b].

2. The EXTRA DDL and EXCESS DML

Two concepts are central to the design of EXTRA/EXCESS: extensibility and support for complex structures with optional identity. In addition, the model incorporates the basic themes common to most semantic data models [Peck88]. Extensibility in EXTRA/EXCESS is provided through both an abstract data type mechanism, where new types can be written in the E programming language [Care88b] and then registered with the system, and through support for user-defined functions and procedures that are written in EXCESS and operate on (user-defined) EXTRA types. Only the latter form of extensibility is of interest here. Complex objects are complex structures in the database (thus every "object" is a "structure"), possibly composed of other structures, that have their own unique identity. Such objects can be referenced by their identity from anywhere in the database. This section presents an overview of the key features of EXTRA and EXCESS; more details can be found in [Care88a].

2.1. The EXTRA Data Model

EXTRA includes support for complex structures with shared subobjects, a novel mix of object- and value-oriented semantics for data, an inheritance hierarchy for top-level tuple types, and support for persistent structures of any type definable in the EXTRA type system. Figure 1 shows a simple database defined using EXTRA. In EXTRA, the tuple, multiset, and array constructors for complex objects are denoted by parentheses, curly braces, and square brackets (combined with the **array** keyword), respectively. Object identity is denoted by the **ref** keyword. In EXTRA, subordinate entities are treated as values (as in nested relational models [Sche86]), not as objects with their own separate identity, unless prefaced by **ref** in a type definition or an object creation statement. The declaration **ref x** indicates that **x** is a reference to an extant object (an OID).

Figure 1 defines four types, all of which happen to be tuple types. The Student and Employee types are subtypes of Person.

```
define type Person:
(
        ssnum:          int4,
        name:           char[ ],
        street:         char[20],
        city:           char[10],
        zip:            int4,
        birthday:       Date
)

define type Employee:
(
        jobtitle:       char[20],
        dept:           ref Department,
        manager:        ref Employee,
        sub_ords:       { ref Employee },
        salary:         int4,
        kids:           { Person }
)
inherits Person

define type Student:
(
        gpa:            float4,
        dept:           ref Department,
        advisor:        ref Employee
)
inherits Person

define type Department:
(
        division:       char[ ],
        name:           char[ ],
        floor:          int4,
        employees:      { ref Employee }
)

create Employees:       { ref Employee }
create Students:        { ref Student }
create Departments:     { ref Department }
create TopTen:          array [1..10] of ref Employee
```

Figure 1: An EXTRA database

The semantics of this inheritance are that all attributes and methods of Person are also attributes and methods of Student and Employee (see Section 4 for method examples), and that Students and Employees are also Persons. Any inherited attribute or method can be overridden with a new type specification or method body, respectively. Multiple inheritance is allowed and is discussed more fully in Section 3.1. Figure 1 creates a database consisting of four named, persistent objects: Students (a set of Student objects), Departments (a set of Department objects), Employees (a set of Employee objects) and TopTen, a fixed-length array of Employee objects.

2.2. The EXCESS Query Language

EXCESS provides facilities for querying and updating complex structures, and as mentioned above it can be extended through the use of ADT functions and operators (written in E) and procedures and functions for manipulating EXTRA schema types (written in EXCESS). EXCESS queries range over structures created using the **create** statement. EXCESS allows for the retrieval, combination, and dismantling of any structure definable in EXTRA. The user-defined functions (written both in E and in EXCESS) and aggregate functions (written in E) are supported in a clean and consistent way.

A simple example should suffice to convey the basic flavor of the language. The following query finds the names of the children of all employees who work for a department on the second floor:

```
range of E is Employees
retrieve (C.name) from C in E.kids
   where E.dept.floor = 2
```

Note the extended "." notation and the ability to range over multiset-valued attributes. More examples appear later in the paper.

Methods (written in EXCESS) may be defined at any time after their types are defined using a **define function** command; see Section 4. New types created during query processing do not participate in inheritance in any way — they do not inherit any attributes or methods (other than those specifically requested) from the types from which they were created nor do they become part of any inheritance hierarchy created during query processing.

3. The EXCESS Algebra

This section describes the algebra used to implement EXTRA/EXCESS. An algebra is formally defined as a pair (S, Θ), where S is a (possibly infinite) set of objects and Θ is a (possibly infinite) set of n-ary operators, each of which is closed with respect to S. The elements of S are called "structures" in this algebra. Section 3.1 describes S and Section 3.2 describes Θ. Some example queries are given in Section 3.3, and Section 3.4 discusses some expressiveness issues concerning the algebra. More detailed and formal definitions of the algebra can be found in [Vand90a].

3.1. The Algebraic Structures

The basic definitions in this section are not new but the treatment of OIDs is new. The full definitions are needed to explain the semantics of OID domains and for completeness. A *database* is defined as a multiset of *structures*. A *structure* is an ordered pair (S, I), where S is a *schema* and I is an *instance*. Schemas are

digraphs (as in LDM [Kupe85]) whose nodes represent type constructors and whose edges represent a "component-of" relationship. An edge from A to B signifies that B is a component of A.

Each node in a schema is labelled with either "set", "tup", "arr", "ref", or "val" (corresponding to the four type constructors plus "val", which indicates a simple scalar value with no associated structure). We also associate a unique type name with each node. Components (fields) of tuples are also named. Every schema has a distinguished root node. We impose the following conditions on a schema *S*:

i) Nodes of type "val" have no components.

ii) A node with no components is either a "val" or "tup" node (the empty tuple type is allowed).

iii) Any node of type "arr", "set", or "ref" has exactly one component: multisets, arrays, and references are homogeneous (contain or point to elements of one type), modulo inheritance.

iv) Let *deref(S)* be *S* with all edges emanating from nodes of type "ref" removed. *deref(S)* must be a forest. This captures the intuition that, when references are not followed, every type is represented by a tree. This also implies that every schema cycle contains at least one node of type "ref".

Figure 2 shows a schema. The root node is at the top and the type names and tuple component names have been omitted. This schema is a multiset of 3-tuples. Each tuple has a scalar field, a field that is an array of scalars, and a field that is a reference (OID) to a scalar.

Next, the domains of values which are defined over the schemas are described. In the definition we will make use of an operation on multisets, the duplicate elimination operator (DE). DE(S) reduces the cardinality of each element of a multiset to 1. The *complex domain* of a schema *S*, written *dom(S)*, is a set (not multiset) defined recursively as follows, based on the type of the root node of *S*:

i) val: $dom(S) = \mathbf{D}$, where **D** is the (infinite) domain of all scalars (excluding OIDs).

ii) tup: $dom(S) = \times_{i=1}^{n} dom(S_i)$, where the S_i are the components of *S*. Note that $dom(S) = \{ () \}$ if n = 0.

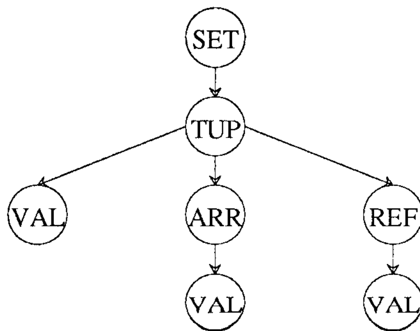

Figure 2: A Schema

iii) set: $dom(S) = \{ x \mid DE(x) \subseteq dom(S1) \wedge |x| < \infty \}$, where *S1* is the component of *S*. The domain is the set of all multisets such that every element of the multiset appears in the domain of the child of the multiset node.

iv) arr: $dom(S) = (\bigcup_{j=1}^{\infty} (\times_{i=1}^{j} (dom(S1)))) \cup \{ [\ ] \}$, where *S1* is the component of *S* and "[ ]" is used to represent an empty array. Since arrays in the algebra are of varying length, the domain of an array node should contain all possible arrays of all possible lengths.

v) ref: $dom(S) = id(S1)$, where *S1* is the component of *S*. *id*(n), for any type n, is an infinite subset of **I**, the infinite set of all OIDs. The function *id* partitions **I** so that if $m \neq n$, $id(m) \cap id(n) = \emptyset$.

Thus any type has an infinite set of OIDs that can only be used on objects of that type. This partitioning of **I** is easily done using the integers [Vand90b]. An instance of a structure with schema S is an element of *dom*(S). An example instance of the schema in Figure 2 is the following: { (26, [1, 2], x), (25, [ ], y) }. Here, "x" and "y" are distinct OIDs whose value is not available to the user.

Note that so far this definition does not take (multiple) inheritance into account. Intuitively, if we have A → B (B inherits from A), we want the "real" domain of A to be *dom*(A) ∪ *dom*(B). More formally, we redefine a domain for schema S to be DOM(S):

$$DOM(S) = dom(S) \cup (\bigcup_{i=1}^{n} DOM(S_i)),$$

where we have $S \rightarrow S_i$ in the type hierarchy for each $1 \leq i \leq n$. The definition is completed by replacing, in parts ii)-iv), *dom* with DOM wherever *dom* appears on the right hand side of a definition.

This is substitutability, the usual semantics for single or multiple inheritance (see e.g. [Bane87]). However, the domains of multisets, arrays, and tuples are actually constructed using the domains of their components, while the domain of a "ref" node is simply a set of OIDs with no relationship to the structure of the component objects. (E.g., if we have A → B, then this definition assures us that arrays of A can also have Bs in them, but does not provide for references to Bs appearing where references to As are expected. To obtain those semantics with the current definition we would need "**ref** A → **ref** B" instead of A → B.)

We present five original rules that specify the desired semantics for OIDs, where *Odom*(A) will indicate the domain of all OIDs for the type named A.

1) The OIDs for a type should never run out. Let ⇒ represent logical implication:
$$(\forall t) (t \in \mathbf{T} \Rightarrow |Odom(t)| = \infty)$$

2) Rule 1 holds even if the type has been specialized with subtypes. This is expressed as follows, where $S = S_1, ..., S_n$ is a list of type names and $Odom(S) = \bigcup_{i=1}^{n} Odom(S_i)$:
$$R \rightarrow S \Rightarrow |Odom(R) - Odom(S)| = \infty$$

3) If S inherits from R then every object of type S is also an object of type R:
$$R \rightarrow S \Rightarrow Odom(S) \subset Odom(R).$$

4) If R and S share no descendants in the type hierarchy then no object has both types R and S. Let →* indicate the

transitive closure of the $\rightarrow$ relation:

$$(\forall t\, (t \in \mathbf{T} \Rightarrow (\neg(S \rightarrow^* t) \wedge \neg(R \rightarrow^* t)))) \Rightarrow Odom(S) \cap Odom(R) = \emptyset.$$

5) This rule specifies the semantics of *multiple* inheritance for OID domains. Intuitively, if each type in a set of types B inherits from each type in a set of types A, then any object of any of the B types is also an object of every A type. Formally, let $A = A_1, \ldots, A_n$ and $B = B_1, \ldots, B_m$ be sets of type names, with $m \geq 1$ and $n > 1$. Let $A \rightarrow B$ signify that all types in B inherit from all types in A. Then

$$A \rightarrow B \Rightarrow \bigcup_{i=1}^{m} Odom(B_i) = \bigcap_{i=1}^{n} Odom(A_i)$$

We modify part (v) of our definition to reflect these semantics:

v') ref: $dom(S) = \mathrm{ID}(S1)$, where $\mathrm{ID}(S) = id(S) \cup (\bigcup_{i=1}^{n} \mathrm{ID}(S_i))$ and we have $S \rightarrow S_i$ in the type hierarchy for $1 \leq i \leq n$.

The domain definitions, including DOM and (v'), now satisfy the semantics for all types. Note that these semantics allow type migration to occur.

3.2. The Algebraic Operators

The orthogonal nature of the type constructors of the algebra has been incorporated into the operator definitions. The algebra is many-sorted, so instead of having all operators defined on "sets of entities", we have some operators for multisets, some for arrays, some for tuples, and some for OIDs (these are the four "sorts" of the algebra). Since EXCESS has the ability to retrieve, combine, and break apart any EXTRA structure(s), the algebra should have this ability as well (otherwise it is not a complete execution engine). This is one motivation for the operator definitions — for each type constructor we introduce a collection of primitive operators that together allow for arbitrary restructurings involving one or two structures of that type.

The many-sorted nature of the algebra gives rise to a large number of operators and thus to a large number of transformation rules (see [Vand90b] for a partial list). At first this may seem to cause an unacceptable increase in the size of the search space the optimizer will need to examine, but this is not really the case. The many-sortedness ensures that only a subset of the operators (and thus of the transformation rules) will be applicable at any point during query optimization. For example, if the optimizer is examining a node of the query tree that operates on a multiset, the rules regarding arrays need not be applied, in general. The following subsections describe multiset, tuple, array, and reference and predicate operators, respectively. [Vand90a,Vand90b] contain complete definitions and numerous examples. For completeness, we list all the algebraic operators in the following subsections. Section 1 indicated the varying degrees of originality associated with these operators, and we give them a corresponding amount of emphasis here.

3.2.1. Multiset Operators

Two multisets are equal iff every element appearing in either multiset has the same cardinality (number of occurrences) in both. The first five operators below are of the most interest to this discussion.

There are eight fundamental multiset operators: 1) **Additive union** ($\uplus$) combines two multisets without doing duplicate elimination. 2) **Set creation** (SET) returns a multiset containing its input, which can be of any type. SET is useful, for example, when one wishes to a add a single element, which does not already occur inside some multiset, to an existing multiset. 3) For **looping/function application** we use SET_APPLY, which applies an algebraic expression E to all occurrences in the input multiset. The result is formed by replacing the occurrences of the input with the structures resulting from applying E to these occurrences. This is an important looping construct, without which we could not even simulate the relational algebra. As an example, let A = { { 1, 1, 2 }, { 2, 3, 4 }, { 1 } }. Then $\mathrm{SET_APPLY}_{\mathrm{INPUT}-\{1\}}(A)$ = { { 1, 2 }, { 2, 3, 4 }, { } }. Here, the symbol "INPUT" refers, in turn, to each occurrence in the input multiset. The set { 1 } is subtracted from each such occurrence to obtain the result. 4) **Grouping** (GRP) facilitates aggregate and other computations. It partitions a multiset into equivalence classes based on the result of an (arbitrary) algebraic expression applied to each occurrence in the multiset. The result will be a multiset of pairwise disjoint multisets, each of which corresponds to a distinct result of the expression applied to the occurrences of the input. 5) **Duplicate elimination** (DE) converts a multiset into a set; the cardinality of each element becomes 1. 6) **Difference** (–), when computing A-B, subtracts the cardinality of an element in B from that in A to obtain the result cardinality of an element. 7) The **Cartesian product** operator (×) is identical to the set-theoretic × except that it allows for (and may produce) duplicates. 8) The **set collapse** operator (SET_COLLAPSE) takes a multiset of multisets and returns the union ($\uplus$) of all the member multisets. This operator also appears in [Abit88].

3.2.2. Tuple Operators

There are four primitive operators on tuples. (Actually, π is expressible in terms of the other three.) 1) **Projection** (π) is similar to the relational projection operator except that it performs its function on a single tuple rather than on a set of tuples. 2) TUP_CAT **concatenates** two tuples. This can help simulate relational cross product. 3) **Field extraction** (TUP_EXTRACT) takes a tuple and returns a single field of the tuple as a structure (the schema of the result is the schema of that field). This differs from π, which always produces a tuple. 4) TUP is used to **create a tuple** — it takes a single structure and makes a unary tuple out of it. TUP could be used, for example, along with TUP_CAT to add a field containing some structure to an existing tuple.

3.2.3. Array Operators

Arrays in the algebra are one-dimensional and variable-length (EXTRA arrays can also be fixed-length, and the algebra operators support those semantics as well). There are nine array operators; we omit the definitions of four operators (ARR_COLLAPSE, ARR_DIFF, ARR_DE, and ARR_CROSS) which are order-preserving analogs of SET_COLLAPSE, –, DE, and ×. The first three operators below are the most relevant for our purposes.

1) To **create an array** we use ARR, which takes a single structure of any type and makes a 1-element array out of it. It is similar to the SET operator. 2) ARR_EXTRACT is used to **extract an occurrence** from an array. It takes an array and returns a single element of it. The distinction between this and

SUBARR is similar to that between the TUP_EXTRACT and π operators. 3) **Apply a function to all occurrences** (ARR_APPLY): This applies an algebraic expression E to every element in the input array. The new instance is simply the array consisting of the result of applying E to each element of the input. This is identical to SET_APPLY except it preserves order. 4) The **subarray** operator (SUBARR) extracts all elements in an array from a given lower bound to a given upper bound and produces an array consisting of these elements in the order found in the input array. 5) ARR_CAT is used for **concatenation**.

3.2.4. Reference Operators and Predicates

References in EXTRA/EXCESS are OIDs that refer to objects which exist in the database independently of objects that reference them. 1) The **dereference** operator (DEREF) collapses a node in the schema graph of a structure. The node corresponding to the part of the structure being dereferenced (materialized) is replaced by its child. The new instance, instead of being an OID, is now an element of the domain of the child node. 2) The **reference** operator (REF) adds a ref-node to the schema graph of a structure and converts its operand into a reference to the operand. This is defined for all structures having an OID. This might be useful, for example, if we wish to manipulate references rather than actual objects during the processing of some queries.

Since algebras are functional languages, we treat **predicates** in a functional manner (similar approaches are taken in [Osbo88, Abit88]). That is, a predicate is an operation (called COMP) that returns its (unmodified) input exactly when the predicate is satisfied (true). Otherwise COMP returns nothing (see [Vand90a] for details). It resembles relational selection [Ullm89], but it operates on a single structure rather than a set of tuples.

Predicates may only appear in a COMP operation; this simplifies operator definitions and implementations. There is only one form of equality, based on the domain definitions — an OID is simply another domain element that can be tested for equality. As an example, let A = (1 4 6 4 1) and let E be the predicate "$\text{TUP_EXTRACT}_{fld2}$(INPUT) = $\text{TUP_EXTRACT}_{fld4}$(INPUT)". Then COMP_E(A) = (1 4 6 4 1) = A. The predicate is satisfied, so the qualifying input structure is returned. Here "INPUT" is merely a shorthand for specifying the entire structure that is the input to the COMP operation; this is different from its function in SET_APPLY and ARR_APPLY.

3.3. Algebraic Query Examples

This subsection is intended to convey the flavor of the algebraic queries and to illustrate the utility of some of the operators. We present two EXCESS queries and an algebraic representation for each. The queries are over the database of Figure 1. In the second example we use a graphical notation to simplify the presentation. An arc from A to B in such a graph is used in place of the linear algebraic expression B(A) — the input to B is the result of A. More examples can be found in Section 5 and [Vand90a].

Example 1:

Figure 3 is a simple query that returns the name and salary of the 5th element of the TopTen array.

```
retrieve (TopTen[5].name, TopTen[5].salary)
```

$$\pi_{name,salary}\ (\text{DEREF}\ (\text{ARR_EXTRACT}_5\ (\text{TopTen})))$$

Figure 3: Query Example 1

Example 2:

This query (see Figure 4) is a functional join [Zani83] that retrieves the names of the departments of all employees who work in Madison. In this and subsequent figures we abbreviate "SET_APPLY" with "S_A". The first expression here converts Employees to a multiset of tuples from a multiset of references (each occurrence in the input multiset is dereferenced). The next node up in the graph selects the tuples such that the employee works in Madison. Then we dereference the "dept" attribute of the qualifying tuples and replace these tuples with the dereferenced "dept" value. The result is a multiset of 1-tuples obtained by projecting the "name" attribute.

3.4. Algebraic Expressiveness

The algebra was designed to implement the EXCESS query language, not to reflect a database-style calculus such as [Abit88]. Thus the interesting question of expressive equivalence for this algebra is not whether it can express the queries of some formal calculus but whether it can express exactly the queries of EXCESS. It is crucial that any EXCESS query be expressible in the algebra. The other direction of the equipollence is interesting in that it restricts the optimization alternatives to the smallest set possible given the power of EXCESS and the structure of the algebra and its rules. It also ensures that intermediate steps in the optimization process are always correct representations of EXCESS queries and that any expressiveness results regarding the algebra can also be applied to EXCESS. We only sketch the proof here.

Theorem:

The EXCESS query language and algebra are equipollent.

```
retrieve (Employees.dept.name)
       where Employees.city = "Madison"
```

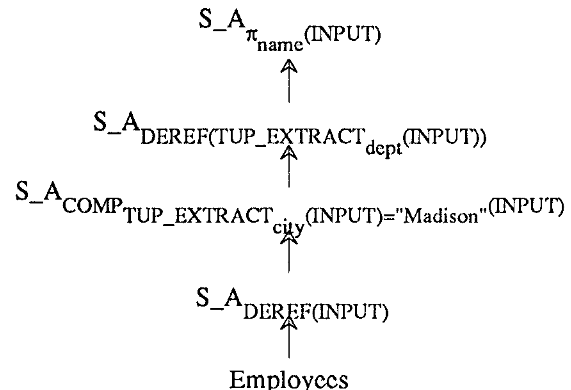

Figure 4: Query Example 2

Proof:

i) Reduction of EXCESS to algebra: The proof that EXCESS is reducible to the algebra is essentially an algorithm that translates any EXCESS query to an algebraic query tree. For brevity, we omit this half of the proof, which is an inductive proof that follows the structure of the algorithm (the induction is on the number of certain EXCESS constructs appearing in a statement).

ii) Reduction of algebra to EXCESS: The other direction of the proof is a case-based inductive proof. We omit most of the cases of the inductive step as our goal is simply to give the flavor of the proof. The proof proceeds by induction on the number of operators in an algebraic expression E. An expression in the algebra consists of one or more named, top-level database objects and 0 or more operators.

Base Case: 0 operators in E

In this case, E = R, a named, top-level database object. The EXCESS query is:

```
retrieve (R) into E
```

Inductive Case: 1 or more operators in E

Assume that all expressions with < n operators ($n \geq 1$) have EXCESS counterparts. There are 23 cases (operators) to consider.

- E = E1 - E2. In this and subsequent cases, assume that E1, E2 have been retrieved into top-level database objects (this is possible via the induction assumption). The EXCESS code for this query is:

```
retrieve (x) from x in (E1 - E2) into E
```

- $E = E1 \times E2$. In EXCESS:

```
retrieve ( E1, E2 ) into E
```

- E = SET (E1). Each type constructor can be used in the target list of a retrieval for output formatting purposes. This translation uses the multiset constructor:

```
retrieve ( { E1 } ) into E
```

- E = ARR_APPLY_{E1} (E2). We first define a type for the elements of the input array (E2). Then a function is defined that applies E1 to structures of this type (this is possible due to the induction assumption; <E1> indicates the EXCESS statement(s) corresponding to algebra expression E1). Finally, this function is invoked on the elements of E2.

```
define type e2_elt : <type(elt(E2))>

define e2_elt function f()
       returns <type(E1(elt(E2)))>
( <E1> )

retrieve (x.f) from x in E2 into E
```

This concludes the inductive case, completing the proof that any query of the algebra can be expressed in EXCESS. Since both directions of the equipollence hold, the theorem is proved. □

A few general remarks about the algebra's power are in order. First, it is capable of simulating most of the algebras mentioned in Section 1 as long as these algebras do not contain the powerset operator (with the obvious exception of [Beer90], which is really a "higher-level" algebra). We conjecture that our algebra is incapable of expressing the powerset, but we have not proved this yet. Such a result would provide an important upper bound on the algebra's expressiveness and computational complexity. This is because the powerset operator, which returns the set of all subsets of its input set, is inherently exponential in nature and that (in many algebras) it allows for the formulation of least fixpoint queries [Gyss88]. Second, it has been observed that the addition of the powerset operator to some algebras has the same effect as adding while-loops with arbitrary conditions [Gyss88]. Such loops are fundamentally different from the style of loop present in the SET_APPLY operator. The latter style of loop executes a statement on each element of a (multi)set in turn. The former kind of loop executes a statement many times, but is not capable of executing the statement on each element of a set (it is not an iteration loop).

4. Algebraic Treatments for Overridden Methods

This section describes method overriding in EXTRA/EXCESS and a new algebraic approach for processing queries that invoke overridden methods on collections of objects that may be of different types due to the inheritance hierarchy for tuple types. Both attributes and methods are inherited by a subtype. A method is simply an EXCESS statement (or sequence of them) defined to operate on structures of a certain EXTRA type and return a structure of some EXTRA type. (In [Grae88], methods are written in a general purpose programming language and database-style optimization is used only if the method is expressible in the algebra.) When a method is defined, it is translated into an algebraic query tree. When the method is invoked, its stored query tree is "plugged in" to the appropriate place in the invoking query tree. The entire query, including the algebraic representation of the method, may now be optimized as a single query. This is clearly better than using a "black box" version of the method, in which the method's query tree can not be optimized along with the invoking query. For example, if (in the database of Figure 1) we define the following method that returns the social security number of an Employee's kid with name "kname":

```
define Employee function get_ssnum
    (kname: char[]) returns int4
( retrieve (this.kids.ssnum)
    where (this.kids.name = kname) )
```

we may be able to take advantage of indices or cached attributes [Maie86, Shek89] if a particular Employee (or set of Employees) has such enhancements. This also allows for transformations that involve nodes in the stored query tree interacting with nodes in the invoking query tree; examples of this can be found in [Beer90]. Thus we want to be able to optimize the algebraic query tree for the method at compile time.

This strategy encounters difficulties when method definitions are allowed to be overridden by subtypes, as is allowed in EXTRA. As an example, suppose the following method is defined on the Person type of Figure 1:

```
define Person function f ( <input_types> )
    returns <output_type>
( <Pbody> )
```

Now we want to override the body of this method for the types Student and Employee. For such overriding we require (as do E and C++) that the type signatures of all the methods be identical (although the implementation could relax this by discarding C++ inheritance and providing its own). We add the following:

```
define Student function f ( <input_types> )
    returns <output_type>
  ( <Sbody> )

define Employee function f ( <input_types> )
    returns <output_type>
  ( <Ebody> )

create P : { Person }
```

The only changes are to the function bodies. The set P can contain Person structures and (because of substitutability) Students and Employees as well. Now suppose the following query is posed:

```
retrieve (P.f(<input_args>))
```

To process this query we must ensure that the proper stored query is invoked for each Person in P. One approach to this is fairly straightforward: the invoking query is optimized without taking the query trees for <Pbody>, <Sbody>, and <Ebody> into account. Whenever the query needs to call "f" on a particular Person, we check (at run time) the actual type of the Person and then invoke the appropriate query tree.

While the previous solution is certainly correct and feasible, it eliminates the important compile time optimization opportunities mentioned above (a more concrete example is given below). There is a second approach that will allow this optimization to take place. We introduce a new parameter to the SET_APPLY operator that is a type name (T). T indicates that only objects that are exactly of type T are to be processed. For example, if T is "Person", then Student and Employee objects are ignored. The solution is to use this version of SET_APPLY for each type in the relevant portion of the hierarchy then union ($\uplus$) the results. In the above example, the initial query tree would look like Figure 5. Each SET_APPLY now has a type name as a parameter as well as the algebraic expression to be applied to each element of the scan (<Pbody>, <Ebody>, and <Sbody> are themselves query trees that can be manipulated by the optimizer, as in the examples of Section 5). This query can now be optimized at compile time, and can take advantage of any transformations involving <Pbody>, <Ebody>, and <Sbody>.

In EXTRA/EXCESS we plan to use both of these solutions to the method overriding problem. In some cases it may be more efficient to use the first approach and in other cases the $\uplus$-based approach may provide some useful optimizations. For example, suppose "f" is a function called "boss" which, given a person "p", returns the name of the person in charge of p's life. For the Person who is not a Student or Employee, this would simply return the Person's name (he is his own boss). For a Student it would return the name of his advisor and for an Employee the name of his manager. Each of these method bodies would be quite simple (at most a DEREF and a TUP_EXTRACT), not allowing for much compile time optimization, and the first technique described above would probably be preferable to scanning P three times, as would be required in the second approach.

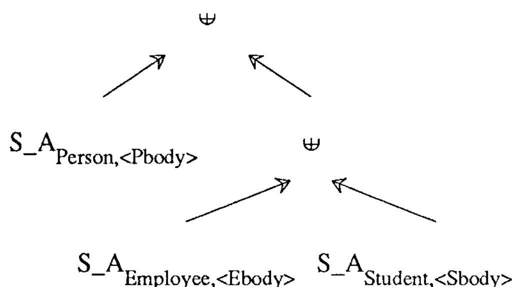

Figure 5: A $\uplus$-based approach to overridden methods

However, if "f" is extremely complicated, the ability to optimize the entire query will be beneficial. For example, if P is very small and the sub_ords attribute of each Employee is very large, then a query invoking an overridden method that scans sub_ords should use the $\uplus$-based approach so that the most expensive part of the query can be optimized at compile time. The $\uplus$-based approach is also advantageous in the presence of certain types of indices. For example, if we have an index on all the Students in P, an index on the Employees in P, and an index on the Persons in P, the need to scan P three times when using the $\uplus$-based approach disappears because these indices can be used to achieve the individual SET_APPLY operations. Finally, multiple SET_APPLYs can sometimes be processed in parallel if the system supports parallelism.

5. Algebraic Transformations

This section describes a few of the new transformation rules that can be used to optimize EXCESS queries and illustrates their use via example queries. A more complete (but still partial) list of the new rules is in [Vand90b]. The algebra is capable of simulating nearly all the transformations found in the literature (see Section 1), but here we emphasize some original rules. Each example presents an EXCESS query over the database of Figure 1 and a series of algebraic representations of that query, in a manner similar to that of Section 3.3. None of these query trees is necessarily intended to be the final plan for the query. Each represents an alternative execution strategy to be examined by the optimizer. In general, their relative merits depend heavily on the nature of the data. In these examples we omit some details of the algebra but we lose none of the essence of the queries.

Example 1:

This query retrieves, without duplicates, the names of all advisors of Students, grouped by their students' departments. For this example, assume that the "advisor" field of Student is a value (the advisor's name) instead of a reference to the advisor. The EXCESS query is:

```
range of S is Students, E is Employees
retrieve unique (S.dept.name, E.name)
   by S.dept where S.advisor = E.name
```

Figure 6 is one way to execute the query — it is similar to what would be produced as an initial query tree by the EXCESS parser. We omit the initial dereferencing of Students and Employees. The query joins the two sets using an operator similar to relational join (defined in [Vand90b]), then groups the result, performs the final projection, and eliminates duplicates. Figure 7 shows the application of a rule that pushes DE ahead of grouping:

$$GRP_E(DE(A)) = SET_APPLY_{DE}(GRP_E(A))$$

This is especially advantageous when the duplication factor is large, as it is likely to be here. We simultaneously take advantage of the ability to move relational π ahead of GRP if the π produces

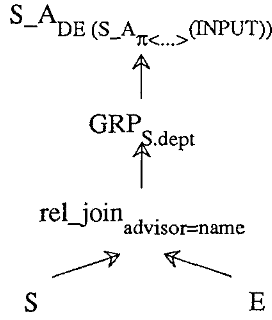

Figure 6: Ex. 1, Initial

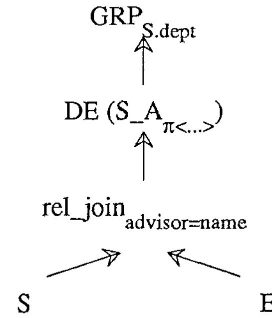

Figure 7: Ex. 1, 1st Transformation

the attributes used by GRP. Here we assume that the new π has been properly adjusted. In Figure 8 we create another alternative by pushing the DE and relational π past the "join" node using this rule:

$$DE\ (A \times B) = DE\ (A) \times DE\ (B)$$

and a relational rule. This results in DE operating on $|S| + |E|$ occurrences rather than $|S| * |E|$ occurrences. The DE and π have been separated into two nodes in Figure 8 to clarify the presentation.

Example 2:

The result of this query is the names of all Students whose major department is located on the 5th floor. The names are grouped by department division (e.g. Engineering, Arts and Sciences, etc.). The EXCESS query is:

```
range of S is Students
retrieve (S.name) by S.dept.division
   where S.dept.floor = 5
```

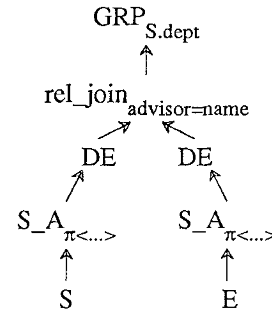

Figure 8: Ex. 1, 2nd Transformation

An algebraic representation appears in Figure 9. Ignoring the initial dereferencing of Students, we group the multiset on the division attribute of its dept attribute, then eliminate the students from departments not on floor 5, then extract the name field. (Here we use σ as a shorthand for SET_APPLY_{COMP} and T_E for TUP_EXTRACT.) Figure 10 shows one way of optimizing the query: successive SET_APPLYs are collapsed, twice, using this rule:

$$SET_APPLY_{E1}(SET_APPLY_{E2}(A)) = SET_APPLY_{E1(E2)}(A)$$

First we collapse the top two nodes of the query in Figure 9 into one node to eliminate one scan of the set, then the query of Figure 10 is obtained by doing the same thing to the *subscript* of the new top node of the query. The σ and π are combined into one SET_APPLY by breaking σ down into its definition. This SET_APPLY first compares the floor attribute of the student's dept attribute to 5, and if the equality holds, the π is applied. The outer SET_APPLY merely invokes the inner one on each group formed by the GRP operation. This ability to optimize within the subscripts of operators in a straightforward manner is extremely useful.

Another way of optimizing this query is presented in Figure 11, which is derived directly from Figure 9. Two rules are used to obtain this version of the query:

$$GRP_{E1}(\sigma_{E2}(A)) = SET_APPLY_{\sigma_{E2}(INPUT)}(GRP_{E1}(A))$$

and (if we have P1(INPUT) = P2(E(INPUT)))

$$E\ (COMP_{P1}(A)) = COMP_{P2}(E\ (A))$$

First, we make use of the fact that selections can be pushed ahead of grouping, with enormous savings if the selectivity factor is low, which it could be here. The other optimization made in Figure 11 is not as obvious. We rewrite the COMP operation using a rule

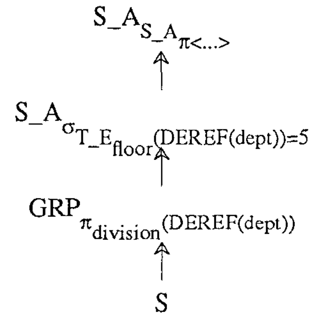

Figure 9: Ex. 2, Initial

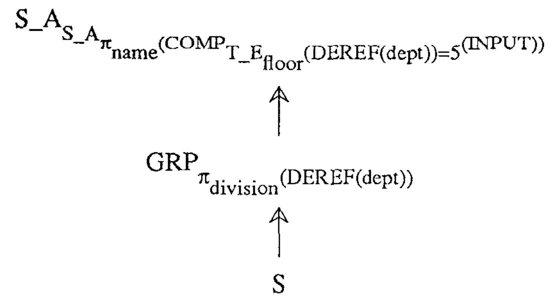

Figure 10: Ex. 2, 1st Transformation

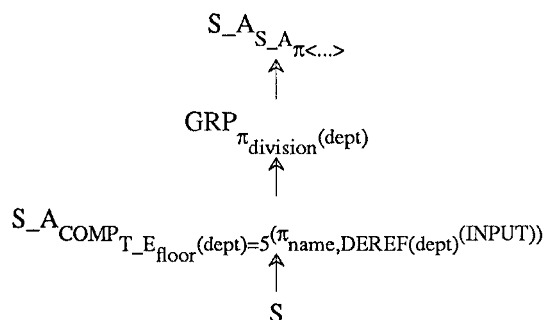

Figure 11: Ex. 2, Alternative 1st Transformation

that allows any expression to be pushed inside of a COMP, as long as operators subsequent to the COMP take into account that the result type of the COMP has now changed. This rule helps here because now the "dept" attribute needs to be DEREF'd only once — before the COMP, which needs to access the fields of "dept". The next time we need to access fields of "dept", in the GRP operation, we need not DEREF it again. The input to the COMP operator is a projection of an element of S. If the floor attribute of this element's dept attribute is 5, the comparison holds. Notice that if the DEREF in this query were instead a more complicated subquery, the savings would be even greater.

6. Conclusions and Future Work

The algebraic approach to database query processing continues to be successful long after its introduction in the relational model. Others have designed algebras for systems with complex structures, enforced object identity, and ordered sets; some research efforts have also included limited sets of transformation rules. Here we extended the algebraic paradigm even farther by providing operators and transformation rules encompassing such issues as array and reference type constructors, multisets, grouping, overridden (inherited) method names, and other issues. This paper also presented set-theoretic semantics for formally specifying the domains of OIDs in the presence of multiple inheritance. The algebra's utility lies in its provable equipollence to EXCESS and in its flexible operators and transformation rules, which can be applied to other systems as well.

EXTRA/EXCESS is being implemented using the EXODUS extensible DBMS toolkit. Much of the system is now operational, including the parser, many of the algebraic operators, the runtime query execution system, the DML support, and support code for the EXODUS optimizer generator, which is being used to build the optimizer. The algebraic rule set (see [Vand90b]) that forms part of the input to the generator is believed to be complete but this has not been proved (a generated optimizer functions with or without a complete rule set). The full system is expected to be running shortly.

Future work includes an investigation of cost functions and useful statistics for complex object data models and testing of various algebraic operators, defined in terms of the primitive ones listed in Section 3, to determine which of these derived operators will be useful for query processing or amenable to optimization. Issues of indexing, data caching, type extents, and other advanced access methods will also be studied in the optimizer. Further examination of the algebra's expressiveness will be made.

Acknowledgements

The authors are grateful to Raghu Ramakrishnan, David Maier, Catriel Beeri, Michael Carey, Joseph Albert, and both referees.

REFERENCES:

[Abit88] S. Abiteboul and C. Beeri, "On the Power of Languages for the Manipulation of Complex Objects," TR No. 846, INRIA, May 1988.

[Abit89] S. Abiteboul and P. Kanellakis, "Object Identity as a Query Language Primitive", *Proc. SIGMOD Conf.*, Portland, Oregon, 1989.

[Bane87] J. Banerjee, et al., "Data Model Issues for Object-Oriented Applications," *ACM Trans. Office Info. Sys.* 5(1), Jan. 1987.

[Beer90] C. Beeri and Y. Kornatzky, "Algebraic Optimization of Object-Oriented Query Languages", *Proc. Int. Conf. Database Theory*, Paris, France, Dec. 1990.

[Care86] M. Carey and D. DeWitt, "Extensible Database Systems", *Proc. Islamorada Workshop on Large Scale Knowledge Base and Reasoning Systems*, Feb. 1986.

[Care88a] M. Carey et al., "A Data Model and Query Language for EXODUS," *Proc. SIGMOD Conf.*, Chicago, Illinois, 1988.

[Care88b] M. Carey et al., "The EXODUS Extensible DBMS Project: An Overview", Comp. Sci. TR #808, Univ. of Wisconsin, Madison, Wisconsin, Nov. 1988.

[Ceri87] S. Ceri et al., "ALGRES: A System for the Specification and Prototyping of Complex Databases," TR 87-018, Dipartimento di Elettronica, Politecnico di Milano, 1987.

[Codd70] E. Codd, "A Relational Model of Data for Large Shared Data Banks," *Comm. ACM* 13(6), June 1970.

[Daya82] U. Dayal et al., "An Extended Relational Algebra with Control Over Duplicate Elimination", *Proc. PODS Conf.*, 1982.

[Daya89] U. Dayal, "Queries and Views in an Object-Oriented Data Model", *Proc. 2nd Intl. Workshop on DBPL*, Gleneden Beach OR, 1989.

[Grae88] G. Graefe and D. Maier, "Query Optimization in Object-Oriented Database Systems: The REVELATION Project", TR CS/E 88-025, Dept. of Comp. Sci. and Eng., Oregon Graduate Center, 1988.

[Guti89] R. Guting et al., "An Algebra for Structured Office Documents", *ACM Trans. Office Info. Sys.* 7(2), April 1989.

[Gyss88] M. Gyssens and D. Van Gucht, "The Powerset Algebra as a Result of Adding Programming Constructs to the Nested Relational Algebra," *Proc. SIGMOD Conf.*, Chicago, Illinois, 1988.

[Kort86] H. Korth and A. Silberschatz, *Database System Concepts*, McGraw-Hill, New York, 1986.

[Kupe85] G. Kuper, "The Logical Data Model: A New Approach to Database Logic," PhD. Thesis, Dept. of Comp. Sci., Stanford University, Stanford, CA, Sept. 1985.

[Maie86] D. Maier and J. Stein, "Indexing in an Object-Oriented DBMS," TR CS/E-86-006, Oregon Grad. Center, Beaverton, Oregon, May 1986.

[Osbo88] S. Osborn, "Identity, Equality, and Query Optimization", in *Advances in Object-Oriented Database Systems*, ed. K. Dittrich, Lecture Notes in Comp. Sci. no. 334, Springer-Verlag, Berlin, Germany, 1988.

[Peck88] J. Peckham and F. Maryanski, "Semantic Data Models," *ACM Comp. Surveys* 20, 3, Sept. 1988.

[Sche86] H.-J. Schek and M. Scholl, "The Relational Model with Relation-Valued Attributes," *Information Sys.* 11(2), 1986.

[Scho86] M. Scholl, "Theoretical Foundations of Algebraic Optimization Utilizing Unnormalized Relations," *Proc Intl. Conf. Database Theory*, Rome, 1986.

[Shaw89] G. Shaw and S. Zdonik, "A Query Algebra for Object-Oriented Databases", TR CS-89-19, Dept. of Comp. Sci., Brown University, Providence, RI, March 1989.

[Shek89] E. Shekita and M. Carey, "Performance Enhancement Through Replication in an Object-Oriented DBMS", *Proc. ACM SIGMOD Conf.*, Portland, Oregon, 1989.

[Tans89] A. Tansel and L. Garnett, "Nested Historical Relations", *Proc. ACM SIGMOD Conf.*, Portland, Oregon, 1989.

[Ullm89] J. Ullman, *Principles of Database and Knowledge-Base Systems*, 2 vols., Computer Science Press, Rockville, Maryland, 1989.

[Vand90a] S. Vandenberg and D. DeWitt, "An Algebra for Complex Objects with Arrays and Identity", TR #918, Comp. Sci. Dept., University of Wisconsin, Madison, Wisconsin, March 1990.

[Vand90b] S. Vandenberg and D. DeWitt, "Algebraic Support for Complex Objects with Arrays, Identity, and Inheritance", TR #987, Comp. Sci. Dept., University of Wisconsin, Madison, Wisconsin, Dec. 1990.

[Vand91] S. Vandenberg, "A Survey of Database Algebras", manuscript in preparation, 1991.

[Zani83] C. Zaniolo, "The Database Language GEM," *Proc. ACM SIGMOD Conf.*, San Jose, CA, 1983.