An Algebra for Complex Objects with Arrays and Identity

by
Scott L. Vandenberg
David J. DeWitt

Computer Sciences Technical Report #918
March 1990

An Algebra for Complex Objects with Arrays and Identity

Scott L. Vandenberg
David J. DeWitt

Computer Sciences Department
University of Wisconsin-Madison
Madison, WI 53706

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.

An Algebra for Complex Objects with Arrays and Identity

Scott L. Vandenberg
David J. DeWitt

Computer Sciences Department
University of Wisconsin-Madison
Madison, WI 53706

ABSTRACT

In this paper we present the design of an algebra that will be used to optimize queries involving complex objects, multisets, arrays, and optional object identity. The target system of the algebra also provides top-level objects that are not sets and a notion of generic functions (functions which can be applied to objects of different types). The algebra supports these features via four orthogonal type constructors (multiset, array, tuple, and reference) and a fixed set of simple operators defined on them. This algebra is being used directly in a query optimizer for the EXTRA/EXCESS system, which is under development at the University of Wisconsin-Madison using the EXODUS extensible DBMS toolkit.

1. Introduction

In recent years it has become apparent that the relational model of data [Codd70], which is extremely useful in many scenarios, can not adequately model all application domains [Care86, Schw86]. Many new models have been proposed to fill the gap. These normally fall into one of the following categories: semantic models [Hamm81, Abit88b], object-oriented models [Lecl87, Bane87, Fish87, Maie86, Mano86], nested relational models [Roth88, Sche86], and complex object models [Kupe85, Abit88a]. In this paper we examine issues of algebras and query optimization for complex objects supporting the additional concepts of arrays and object identity.

A data model supporting these features (as well as several others), called the EXTRA/EXCESS data model [Care88a], is under development at the University of Wisconsin-Madison. The system is being implemented using the EXODUS extensible DBMS toolkit [Care88b]. Briefly, the EXTRA data model allows one to define types with arbitrary structure using 0 or more of the set, tuple, and array type constructors in an orthogonal manner. Arrays are 1-dimensional[1] and can be either fixed- or variable-length. Database entities are then defined to be of one of these types--there is a clear type/instance dichotomy. In addition, any component of one of these structural type definitions can be declared as a reference, or unique object identifier, that refers to an object of a particular type. The notion of references provides sharing in the model. Multiple inheritance among tuple types is provided as well, and sets and arrays are homogeneous modulo certain rules involving this inheritance hierarchy. The model also

[1] Multi-dimensional arrays can be constructed using arrays of arrays, as in the C programming language [Kern78].

supports user-defined ADTs, derived attributes, automatic maintenance of inverse relationships, and several varieties of object ownership semantics. EXCESS, the query language for the model, is essentially an extension of GEM [Zani83]. It provides for the retrieval and update of objects of any EXTRA type and supports user-defined functions and procedures written in EXCESS. Procedures differ from functions in that functions return a value and have no side-effects, while procedures usually have side-effects and return no value.

To relate this system to other systems proposed in the literature, it is useful to conceive of the relational model as the model from which many others are derived. The nested relational models are a direct extension that adds the ability to have relation-valued fields, and complex objects extend these by allowing the tuple and set type constructors to be applied in any order. Finally, object-oriented systems can in some sense be seen as extending complex objects with the concepts of object identity, classes, methods, and inheritance (and possibly other concepts--there is no standard object-oriented data model). EXTRA/EXCESS, then, can be viewed as a complex object model which also has an array type constructor and the ability to use object identifiers (but doesn't require their use, as object-oriented systems do). The notion of methods is only partially modeled by our EXCESS functions and procedures, and our notion of inheritance is slightly less robust as well (only tuple types may be involved).

The remainder of the paper is organized as follows: Section 2 will motivate the algebra and its overall design and Section 3 will present a brief overview of the algebra's structures and operators. Sections 4 and 5 contain more complete descriptions of the structures and operators, and Section 6 presents some examples of queries expressed in the algebra. Some related work is described in Section 7. The last section contains a summary and an outline of current and future work.

2. Motivation

Queries posed by a DBMS user are usually formulated in a language which describes what data to get but not how to get it; this is a calculus-based language. But a procedural form of the query, consisting of a sequence of operations on stored data, is much more useful for query processing, as it can frequently be rearranged to produce different plans for executing the query. Some of these will be faster than others. Algebraic query optimizers perform these transformations on the original query until a suitable query execution plan is obtained.

The EXTRA/EXCESS system contains a number of features that have not appeared in an algebra. The following six features combine to make the data model unique enough so that no existing query algebra applies: arrays as a type constructor, optional object identity, the presence of top-level objects of any type, generic functions (defined

below), a limited form of (multiple) inheritance (specialization among tuple types, to be exact), and the use of multisets rather than sets. A multiset is a set that allows the same element to appear more than once; e.g., { 1, 1, 2 } is a multiset but not a set. Each element has an associated cardinality, or number of *occurrences*, in the multiset.

Generic functions are user-defined functions (e.g. aggregates) that operate on multisets or arrays. Such a function is defined to take as input a multiset or array containing elements of a type, call it T. T can either be a named, existing EXTRA type or a set of operations. In the former case, the input multiset or array must contain elements of type T. In the latter case, the input multiset or array can contain elements of any type for which the specified operations are defined. The output of a generic function, like the input, is either a specific, named EXTRA type or is defined to be the same type as the type of the input elements (i.e., the type is bound at compile- or run-time to the type of the input elements, which will not be known until then if T is not a specific type).

Another motivation is the lack of a standard algebra for complex object systems. Relational and nested relational systems seem to have settled on standard sets of operators (with minor variations). But the proliferation of data models more complex than these, and the many differences among them, make it seem unlikely that a standard will arise in the near future, if at all. The algebra presented here is not meant to be such a standard, but is meant to facilitate research into the properties of algebras for complex object and object-oriented systems. A major difficulty with extending the relational and/or nested relational algebras directly to complex objects is the essentially infinite variety of structural combinations available due to the arbitrary application of the type constructors in these more advanced models. But the relational algebra was so successful that it seems something should be learned from it. Therefore we adopt the relational approach of first defining the structures available during query processing and then defining the operators which operate upon these structures. A key advantage of the relational approach is the small number of relatively simple concepts involved and the simplicity of the operators. We hope to achieve this not by extending the relational algebra but by using a similar approach to defining the algebra.

3. An Overview of the Algebra

The orthogonal nature of the type and entity declarations of EXTRA has been carried over into the algebra. Any database entity (we use the term "entity" to distinguish it from "object", which has many connotations) can be either a multiset, a tuple, an array, a reference to an entity, or a simple value. An entity can be composed of any number of other entities of any type. Thus the algebra has type constructors corresponding to each of these sorts of entities: "set", "arr", "tup", and "ref". (There is no type constructor corresponding to scalar values.) Arrays are

sufficiently different from multisets and tuples to warrant a separate type constructor. We treat all arrays in the algebra as having variable length. "Ref" is a type constructor because there are operations that can be performed on references but not on the things they reference and vice-versa. Multisets were chosen over sets since in reality most systems never implement true sets, they implement multisets while using an algebra which was designed to manipulate only sets. Multisets also facilitate aggregate computations in the algebra. In the algebra, then, an entity is built using scalar values and these four type constructors applied in any fashion. In addition to these type constructors, the algebra needs to take into account the concepts of inheritance, generic functions, and predicates in order to become a relatively complete query processing paradigm. We will not discuss the details of the algebra's inheritance mechanisms in this paper. This leaves us with four type constructors plus predicates and generic functions.

Briefly, schemas in the algebra are modeled as directed graphs in which each interior node represents a type constructor and each leaf node represents the domain of scalar values. There is an edge from v1 to v2 if the type represented by v2 is a component of the type represented by v1. There is no restriction on the structure (type) of v2, which means that there is no restriction on the operations one might need to perform on the components of v1. Thus we do not know, for example, that every multiset will contain tuples, as in the relational model. We know only that a multiset will contain elements of some specific type. This holds for all the type constructors. For this reason we organize the operators around the type constructors--each constructor has an associated set of operators, each of which operates only on structures whose outermost type constructor is the one in question. There are nine operators on multisets, four on tuples, ten on arrays, and two on references. Every operator takes one or two algebraic objects as input and returns one algebraic object as its output, so the definitions automatically preserve closure of the algebra.

Note that we have left out predicates and generic functions. Predicates exist in the algebra only in one operator, a general comparison operator which compares an arbitrary algebra expression against an arbitrary predicate and is defined to return an object of the algebra rather than "true" or "false". This preserves the functional nature and closure property of the algebra and isolates predicates in a single operator, keeping the other operator definitions quite simple. Finally, the definition of a generic function is viewed as creating an entirely new operator in the algebra, and since the user must define it as consuming and producing EXTRA types, it will not violate the closure property of the algebra. Thus the total number of operators in the algebra is unbounded (but at least 26).

All of the operators have been carefully defined to enable all possible manipulations of the data definable in EXTRA. It should also be pointed out that the relational, nested relational, and complex object [Abit88a] algebras

(without the powerset operator) are all subsumed by this algebra. The next sections contain more complete definitions of the algebraic structures and operators.

4. The Algebraic Structures

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. This section describes S and the next section describes Θ. The elements of S are called "structures" in this algebra.

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 whose nodes represent type constructors and whose edges represent a "component-of" relationship. That is, the edge A→B says that B is a component of A.

More formally, a *schema* is a digraph S = (**V**, **E**), where **V** is the set of labeled vertices and **E** is the set of edges. Let *order*(v) indicate the number of outgoing edges from vertex v. The labeling function for vertices is *type*(v), with *type*(v) ∈ {set, tup, arr, ref, val}. (These constants correspond to the four type constructors plus "val", which indicates a simple scalar value with no associated structure.) We impose the following conditions on S:

i) *type*(v) = val => *order*(v) = 0.

ii) *order*(v) = 0 => *type*(v) ∈ {val, tup}.

iii) *type*(v) ∈ {set, arr, ref} => *order*(v) = 1.

iv) If there is a path $(v_1,...,v_n)$, and $v_1 = v_n$ (i.e., there is a cycle), for any n > 1, then ∃ i (*type*(v_i) = ref).

v) Let *deref(S)* be S with all edges emanating from nodes of type "ref" removed. *deref(S)* must be a forest.

The first condition ensures that any node representing a simple value has no components. The next condition states that only value and tuple nodes may have no children (the empty tuple type is a legal type). Condition (iii) states that all nodes representing multisets, arrays, and references have only one component. This is equivalent to the statement that multisets, arrays, and references are homogeneous (contain or point to elements of the same type), modulo certain inheritance rules which we describe later. Condition (iv) requires any cycle in the schema to have at least one node of type "ref". This is to ensure that certain undesirable (infinite) data structures do not arise. The fifth condition captures the intuition that, when references are viewed as simple scalar values and not followed, every type is represented by a tree. Note that condition (v) implies condition (iv). We also introduce a distinguished node in S, denoted *root*(S), which indicates the node representing the top-level constructor of the type associated with the

schema *S*. Figure 1 shows an example schema.

We now describe the domains of values which are defined over the schemas. In the definition we will make use of two standard operations on multisets: the cross-product, which is analogous to the set-theoretic cross-product but allows (and produces) duplicates, and 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 defined recursively as follows (where "{" and "}" are multiset, not set, constructors):

i) If *type*(*root*(S)) ∈ {val, ref} then *dom(S)* = **D**, where **D** is the (infinite) domain of all scalars.

ii) If *type*(*root*(S)) = tup then $dom(S) = \times_{i=1}^{n} dom(S_i)$, where × indicates the multiset cross-product operator and the S_i are the children of *S*. Note that *dom(S)* = ∅ if *order*(v) = n = 0.

iii) If *type*(*root*(S)) = set then *dom(S)* = { x | DE(x) ⊆ *dom(S1)* }, where *S1* is the child of *S*.

iv) If *type*(*root*(S)) = arr then $dom(S) = (\cup_{j=1}^{\infty} (\times_{i=1}^{j} (dom(S1)))) \cup \{ [\,] \}$, where × denotes the cross-product operator, *S1* is the child of *S*, and "[ ]" is used to represent an array with no elements (it is legal for a variable-length array to be empty).

Part (i) of the definition reflects the nature of object identifiers--they are merely scalars, and gain meaning as object identifiers by being part of a "ref" node in a structure. The domain of a tuple type follows the usual relational definition. For a multiset node (part (iii) of the definition), the domain is the set of all multisets such that every element (but not every occurrence of an element) of the multiset appears in the domain of the child of the multiset node. Since arrays in the algebra are of varying length, the domain of an array node should contain all possible

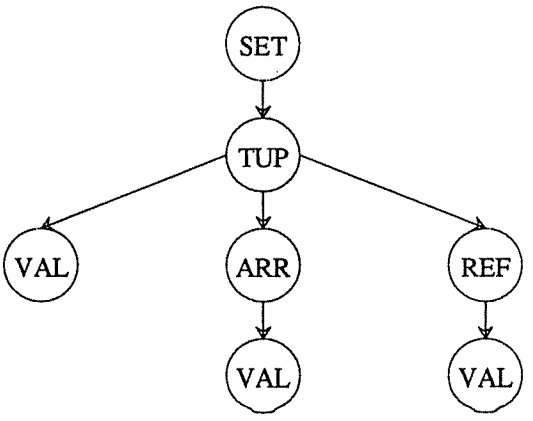

Figure 1: A Schema

arrays of all possible lengths, including empty. So for each length (index variable j) we construct all possible arrays using the set-theoretic × operator.

An instance I of a structure with schema S is an element of *dom*(S) ∪ { *unk*, *dne* }. *Unk* is a null value indicating that the value is unknown, but that there is a value. *Dne* is a null value indicating that the value does not exist. The *unk* null is essentially the same as the "null" of GEM [Zani83].

We will not detail the algebra's structural inheritance mechanisms here, except to mention that only tuple types are involved in the inheritance hierarchy and that entities of type X can always replace entities of type Y if X inherits from Y (substitutability). An example instance of the schema in Figure 1 is the following, where {}, [], and () denote multiset, array, and tuple type constructors, respectively: { (26, [1, 2], x), (25, [ ], y) }. Here, "x" and "y" are distinct object identifiers whose value is not available to the user.

5. The Algebraic Operators

Each operator of the algebra takes one or two structures (as defined in the previous section) as input and produces a single, entirely new structure as output. In this section we describe each operator informally (the formalisms are omitted for brevity and can be found in [Vand89b]). We also omit descriptions of operator behavior in the presence of nulls. The following subsections describe the operators for multisets, tuples, arrays, references, predicates, and generic multiset/array functions.

5.1. Multiset Operators

Recall that a multiset consists of a number of distinct *elements*, each of which has a certain number of *occurrences* in the multiset. Two multisets are equal if every element appearing in either multiset has the same cardinality in both. Multisets in EXTRA/EXCESS are homogeneous modulo tuple type inheritance. That is, a multiset is always defined as containing elements of a single specific type. However, if the type of these elements is a tuple type, and this tuple type has subtypes in the EXTRA type hierarchy, then elements in the multiset may also be of any of these subtypes. We now describe each of the nine operators.

1)-3) **Union** (∪), **difference** (–), and **additive union** (⊎): These resemble the set-theoretic union and difference operators. Union and additive union differ in that ⊎ adds the cardinalities of an element appearing in both multisets to obtain its cardinality in the result. ∪ uses the maximum of these cardinalities for that computation, and thus degenerates to set-theoretic union when no duplicates are present. The difference operator, when computing A-B, subtracts the cardinality of an element in B from that in A to obtain the result cardinality of an element. Note

that the identity $A \cap B = A - (A - B)$ still holds for multisets (the proof of this is trivial).

These three operations are defined on any two multisets whose elements have *compatible* schemas. Briefly, two schemas are compatible if they have the same structure with the possible exception of some of the tuple nodes in the schema graph. Corresponding tuple nodes in the two schema graphs must either be identical or have a common supertype in the EXTRA inheritance hierarchy. The result of one of these binary operations on compatible types will have this supertype in place of the "conflicting" tuple types of the inputs.

Examples: Let $A = \{1, 1, 2, 3\}$ and $B = \{1, 2, 2\}$. Then $A \cup B = \{1, 1, 2, 2, 3\}$, $A - B = \{1, 3\}$, and $A \uplus B = \{1, 1, 1, 2, 2, 2, 3\}$.

4) **Cartesian product** ($\times$): This is identical to the set-theoretic Cartesian product except that it allows and may produce duplicates. The result is a multiset of ordered pairs. Note that this is different than the relational algebra cross product, which results in a set of tuples which have been concatenated from the two input tuple types. This was not possible here since the inputs may not be multisets of tuples.

Example: Let $A = \{1, 2\}$ and $B = \{3, 3\}$. Then $A \times B = \{ (1\ 3), (1\ 3), (2\ 3), (2\ 3) \}$.

5) **Duplicate elimination** (DE): This operator converts a multiset into a set. Any element with cardinality > 1 has its cardinality reduced to 1. Pure set-theoretic processing can be imitated using this operator.

Example: Let $A = \{1, 1, 2\}$. Then $DE(A) = \{1, 2\}$.

6) **Grouping** (GRP): This operator 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. Each of these multisets will contain the original occurrences appearing in the input, not the results of applying the expression to these occurrences. This operator is useful in computing aggregates and other generic functions.

Example: Let $A = \{ (1\ 2), (1\ 3), (2\ 5) \}$. Then $GRP_{\pi_1(INPUT)}(A) = \{ \{ (1\ 2), (1\ 3) \}, \{ (2\ 5) \} \}$, where π is the tuple projection operator (described fully in the next section). This GRP operation partitions the tuples of A into equivalence classes based on the value of the first attribute of the tuple. The symbol "INPUT" refers, in turn, to each occurrence in the input multiset.

7) **Create a set** (SET): This operator takes a single structure and makes a singleton multiset out of it. The SET operation is always defined--that is, any structure may become the only occurrence in a newly created multiset.

It thus differs from the other operators in this section in that its input is not required to be a multiset. SET is useful, for example, when one wishes to add a single element, which doesn't already occur in some multiset, to an existing multiset. For this purpose it would precede one of the union operators.

Example: Let A = [1, 2, 3, 5]. Then SET(A) = { [1, 2, 3, 5] }.

8) **Destroy a set** (SET_COLLAPSE): The SET_COLLAPSE operator takes a multiset of multisets and returns the union ($\cup$) of all the member multisets. This is useful, for example, when extracting a multiset-valued attribute from each tuple in a multiset of tuples.

Example: Let A = { { 1, 2, 3 }, { 2, 3, 3 } }. Then SET_COLLAPSE(A) = { 1, 2, 3, 3 }.

9) **Apply a function to all occurrences** (SET_APPLY): This operator applies an algebraic expression E to all occurrences in the input structure, which must be a multiset. The new structure 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.

Example: Let A = { { 1, 1, 2 }, { 2, 3, 4 }, { 1 } }. Then $\text{SET\_APPLY}_{\text{INPUT-}\{1\}}(A)$ = { { 1, 2}, { 2, 3, 4}, { } }. Here, the symbol "INPUT" refers, in turn, to each occurrence in the input multiset.

5.2. Tuple Operators

The notion of a tuple in EXTRA/EXCESS is the usual notion of tuple. It consists of a fixed number of (ordered) elements, each of which may be of a different type. The same caveat about subtypes that applied to multisets applies here--if a field of a tuple is defined to be of a certain type, a value appearing in that field must be of that type or one of its subtypes. There are four operators on tuples.

1) **Projection** (π): This is similar to the relational projection operator [Codd70] except that it performs its function on a single tuple rather than on a set of tuples.

Example: Let T = (5 6 7). Then $\pi_{1,3}(T)$ = (5 7).

2) **Concatenate** (TUP_CAT): This operator takes two tuples and produces a tuple whose fields consist of all fields of the first tuple (in order) followed by all fields of the second tuple (in order). One important use of this operator is to help simulate the relational cross product operator.

Example: Let T1 = (5 6 7) and T2 = (9 14). Then TUP_CAT(T1, T2) = (5 6 7 9 14).

3) **Create a tuple** (TUP): This operator takes a single structure and makes a unary tuple out of it. The TUP operation is always defined--that is, any structure may become the only field in a newly created tuple. This distinguishes this operator from the others in this section, which must have tuples as inputs. TUP could be used, for example, to add a field containing any structure to an existing tuple.

Example: Let T = [a, b, c]. Then TUP(T) = ([a, b, c]).

4) **Field extraction** (TUP_EXTRACT): This operator takes a tuple and returns a single field of the tuple as a structure. This differs from the π operator, which removes some fields from the tuple but still produces a tuple.

Example: Let T = (9 18 27). Then $\text{TUP\_EXTRACT}_2(T) = 18$.

5.3. Array Operators

As is the case with multisets, arrays in EXTRA/EXCESS are homogeneous modulo tuple type inheritance. An EXTRA array is one-dimensional, and has either a fixed finite positive integer number of elements or a variable (finite but unbounded) number of elements. In the latter case, the array can grow and shrink at any point. In the algebra, however, we do not distinguish between these two cases. The operators have been designed so that arrays of either type can be manipulated with the proper semantics provided the right operators are used. An array has 0 or more elements (with "[ ]" used to indicate an empty array) and the indices begin with 1. In EXTRA/EXCESS, arrays can be associated with a range variable just like multisets can. This implies that the functionality of multisets should also be provided for arrays. However, it is not clear that we always want to destroy the order of an array during query processing. There are applications where it might be useful, for example, to see the results of a join of two arrays of tuples printed out in an order specified by their ordering in the original arrays.

Thus we allow some of the multiset operators to apply to arrays and define order-preserving analogs for some of the other multiset operators. In addition, there are several operators designed specifically for arrays. The $\cup$, $-$, $\times$, and GRP multiset operators are now extended to allow an array argument or arguments. These operators were described above. The operations that can be performed specifically on arrays are as follows: extract a subarray, concatenate two arrays, create an array, extract an element from an array, and apply an algebraic expression to all elements of an array. These we will describe below. We omit the definitions of five other operators (ARR_COLLAPSE, ARR_UNION, ARR_DIFF, ARR_DE, and ARR_CROSS) which are order-preserving analogs of SET_COLLAPSE, $\cup$, $-$, DE, and $\times$. Finally, note that the ARR_CAT, ARR_UNION, and ARR_DIFF operators operate on compatible structures, just like $\uplus$, $\cup$, and $-$ do.

1) **Subarray** (SUBARR): This operator 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 original (input) array. The bounds are integers ≥ 1 or the special token "last", indicating the current last element of the array.

Example: Let A = [1, 1, 2, 3, 5, 8, 13]. Then $SUBARR_{3,5}(A)$ = [2, 3, 5].

2) **Concatenate** (ARR_CAT): This operator takes two arrays and produces an array whose elements consist of all elements of the first array (in order) followed by all elements of the second array (in order). It is similar, in terms of what the result contains, to the $\uplus$ operator.

Example: Let A1 = [1, 2, 3, 5, 7] and A2 = [11, 13, 17]. Then ARR_CAT(A1, A2) = [1, 2, 3, 5, 7, 11, 13, 17].

3) **Create an array** (ARR): This operator takes a single structure and makes a 1-element array out of it. The ARR operation is always defined--that is, any structure may become the only element in a newly created array. This differs from the other operators in this section in that it is not required to have an array input. Its utility is similar to that of the SET operator.

Example: Let A = { 47, 3 }. Then ARR(A) = [{ 47, 3 }].

4) **Extract an occurrence** (ARR_EXTRACT): This operator takes an array and returns a single element of it as a structure. This differs from the SUBARR operator, which removes some elements from the array but still produces an array. This distinction is similar to that between the π and TUP_EXTRACT operators.

Example: Let A = [{ 1, 2 }, { 9 }, { 3 }]. Then $ARR\_EXTRACT_2(A)$ = { 9 }.

5) **Apply a function to all occurrences** (ARR_APPLY): This operator applies an algebraic expression E to every element in the input structure, which must be an array. The new instance is simply the array consisting of the result of applying E to each element of the input $[a_1, ..., a_m]$. The result is then $[E(a_1), ..., E(a_m)]$.

Example: Let A = [(a b), (c d)]. Then $ARR\_APPLY_{\pi_1(INPUT)}(A)$ = [(a), (c)]. The symbol "INPUT" functions as in SET_APPLY.

5.4. Reference Operators

References in EXTRA/EXCESS can be thought of as object identifiers (OIDs) that refer to objects which exist in the database independently of objects that reference them (except for their owners; see [Care88a]). We again encounter the tuple-type inheritance notion here: a structure declared as a reference to objects of a certain type may also refer to objects of its subtypes in the EXTRA type hierarchy for tuple types. Two operators are avail-

able on references: dereferencing and creation of a reference from an existing object.

1) **Dereference** (DEREF): This operator effectively 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 just an object identifier, is now a complete element of the domain of the child node.

2) **Reference** (REF): This operator adds a ref-node to the schema graph of a structure and converts its operand into a reference to the operand. This operation is defined for all inputs R (i.e., it does not operate only on references, as the DEREF operator does). This might be useful, for example, if we wish to create and manipulate references rather than actual objects during the processing of some queries.

5.5. Predicates

Since algebras are functional languages, we treat predicates in a functional manner. That is, a predicate is an operation (called COMP) that returns its (unmodified) input exactly when the predicate is satisfied (true). Otherwise a null value is returned.

Comparisons (COMP): This operator takes a single structure as input and compares it to an arbitrary algebraic expression using one of a fixed set of comparators. It bears a resemblance to the relational select operator [Ullm82], but it operates on a single structure rather than a set of tuples. There are three truth-values in our predicate system: T (true), F (false), and UNK (unknown). If the predicate (P) evaluated with the input structure S is T, $COMP_P(S) = S$. If the value of the predicate is UNK the COMP operator returns *unk*. If the value of the predicate is F then COMP returns *dne*. Notice that we distinguish between the UNK truth-value and the *unk* null value, which is actually a legal data value anywhere in the database. Also note that there is no need for a truth-value corresponding to the *dne* null value (the results of comparisons involving *dne* are always either T or F; the complete truth-tables can be found in [Vand89b]).

Predicates consist of atomic equality predicates connected by $\wedge$ and $\neg$. In the COMP operator, equality (and thus multiset membership, which is conceptually an equality test against every occurrence in a multiset) is based solely on value equality. The special null constants are given a semantic interpretation, similar to that of [Gott88]. It will also be useful to pose queries that treat null values as a symbolic constant rather than as a semantic constant [Zani83, Gott88]. Thus we allow two auxiliary comparators, == and $\in\in$, which force the null constants *unk* and *dne* to be treated symbolically, like any other constants, rather than semantically.

Examples:

i) Let A = (1 4 6 4 1). Then $COMP_E(A)$ = (1 4 6 4 1) = A, where E is the predicate "$TUP\_EXTRACT_2$(INPUT) = $TUP\_EXTRACT_4$(INPUT)". The predicate is satisfied, so the qualifying input structure is returned. Here the symbol "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 the SET_APPLY and ARR_APPLY operators.

ii) Let A be as above. Then $COMP_E(A)$ = *dne*, where E is the predicate "$TUP\_EXTRACT_3$(INPUT) = 9". The predicate is not satisfied (it evaluates to F), so *dne* is returned.

5.6. Generic Multiset/Array Functions

As mentioned in Section 2, EXTRA/EXCESS provides generic multiset/array functions. Each generic function in effect defines a new operator, but since the behavior of each is necessarily unknown (all we know is its signature), we will treat all such functions similarly in the algebra and in the transformation rules (which are not discussed here).

When a generic multiset function, say G, is invoked in the query language, semantic checking of the constraints on its input type will occur before algebraic query processing begins. Within the algebra, G(E), for some algebra expression E, will be defined iff E is a multiset or array. G will return a structure of some EXTRA type, so there are no problems in composing G with any of our other operators as defined above. Example 5 in the next section shows how generic functions fit into algebraic queries.

6. Examples

In this section we give several EXCESS queries and their algebraic counterparts. Except where noted, these queries range over three top-level multiset objects, each containing references to a tuple type (the Student, Employee, and Department types). Assume that the Employee type has scalar fields *name*, *salary*, and *city*, and a field *dept* that is a reference to a Department tuple. The Department type has scalar fields *name* and *floor* and an *employees* field, which is a set of references to Employee tuples. Further assume that the Student and Employee types inherit their *name* and *city* fields from the supertype Person. In the examples we will often use field names rather than integers to identify the attributes of a tuple; this is merely to make the examples easier to read. Also recall that many of the operators use subscripts, but none use superscripts.

Example 1:

Assume we have a top-level array of 10 references to Employee tuples, called TopTen, and we wish to retrieve the name and salary of the 5th-best employee. The EXCESS and algebra for this are in Figure 2.

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

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

Figure 2: Example 1

The remainder of the examples are presented in a graphical format to make them easier to read. The actual database structures appear at the bottom, and each successive node going upward represents an algebraic expression to be applied to its input structure.

Example 2:

This query retrieves the names of all employees working for a department on floor 3; see Figure 3. The first expression converts Departments to a multiset of tuples from a multiset of references (each input occurrence in the multiset is dereferenced). The next node up in the query selects the tuples from its input for which the "floor" attribute is 3. In this node, the higher-level "INPUT" refers to a single occurrence of the input multiset, and the lower-level "INPUT" happens to refer to the same thing (recall that "INPUT" in the COMP operator signifies the whole input structure, not just a single occurrence from an array or multiset). The next expression up in the tree replaces each department with its employees attribute. The final result is obtained by unioning the multisets in the multiset of multisets represented by this expression then dereferencing each employee object and projecting its name field. The result is a multiset of 1-tuples.

Example 3:

This query is a functional join [Zani83] that retrieves the names of the departments of all employees who work in Madison. It is depicted in Figure 4. 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

```
range of D is Departments
retrieve (E.name) from E in D.employees where D.floor = 3
```

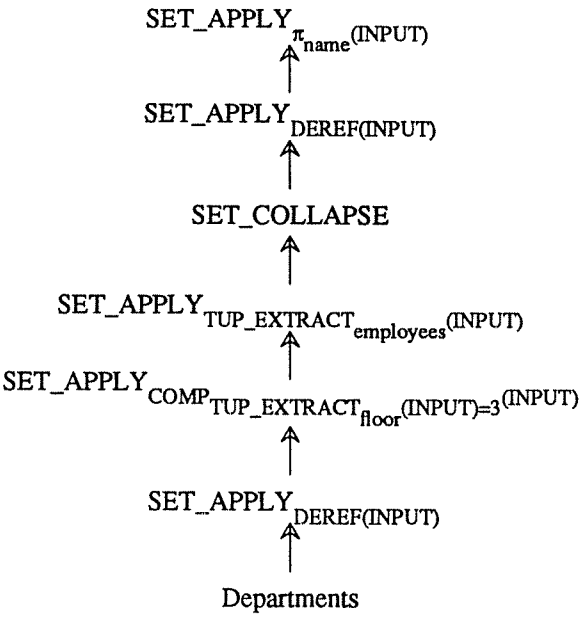

Figure 3: Example 2

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

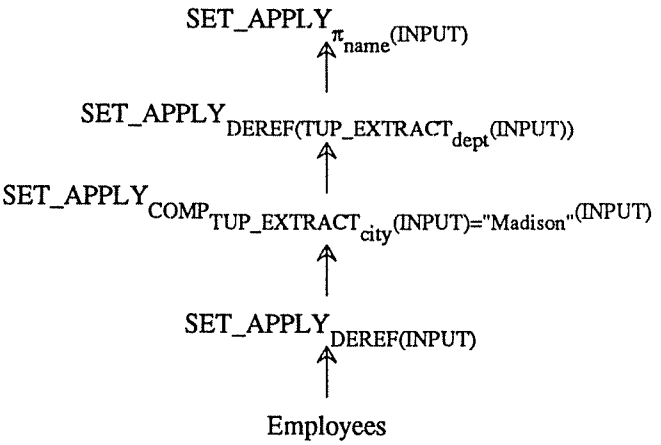

Figure 4: Example 3

multiset of 1-tuples obtained by projecting the "name" attribute.

Example 4:

This query is a value-based join that retrieves the names of all persons living in the same city as Jones. See Figure 5. In this query, we chose not to initially dereference the elements of the top-level multisets as we did in the previous example. Instead, we dereference the objects each time we need to traverse into them. This was done simply to demonstrate that there are decisions to be made about when to dereference objects. In this query, the two union nodes obtain the multiset of all people by unioning the multiset of all employees and the multiset of all students. Then we perform a cross-product of this multiset with itself. The result of this is a multiset of 2-tuples, in which the first and second fields are references to Persons and represent the range variables P1 and P2, respectively. Thus the expression following the Cartesian product compares the "name" attribute of each P2 tuple to "Jones" and performs a selection (in relational terms). The next node (another selection) evaluates the predicate "P1.city = P2.city"; we omit the predicate to simplify the presentation. The

```
retrieve (P1.name) from P1, P2 in (Students + Employees)
where P1.city = P2.city and P2.name = "Jones"
```

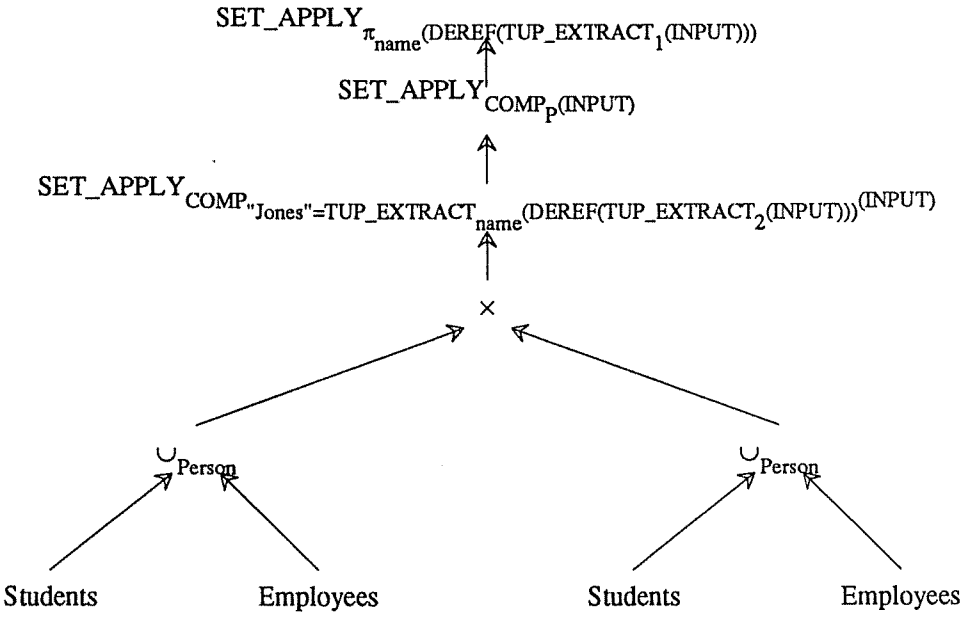

Figure 5: Example 4

result is obtained by projecting the "name" attribute of the P1 tuple from each qualifying ordered pair of references which remains.

Example 5:

This query retrieves, for each department, the average salary of that department's employees, using the generic multiset function "avg" (which presumably was defined at some point by a user to consume and produce certain EXTRA types). The EXCESS and algebraic forms of the query are shown in Figure 6. The first non-trivial node in the query tree converts Employees from a multiset of references to a multiset of tuples. Then we group the tuples according to equal values of the "dept" attribute. Next, for each resulting group, we replace each tuple in the group with the salary attribute of that tuple. Finally, we apply the generic multiset function "AVG" to each element of the grouped multiset to get a multiset of integers (presumably) as the result.

```
retrieve (avg(E.salary by E.dept from E in Employees))
```

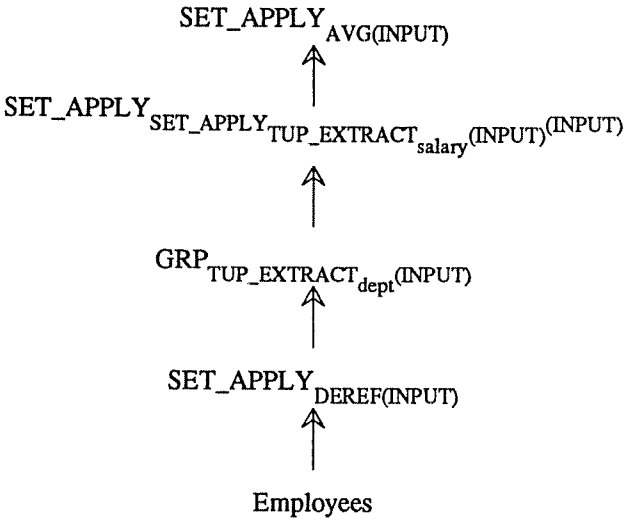

Figure 6: Example 5

7. Related Work

To summarize the relationship of this algebra to previous work, it is useful to divide the relevant algebras into two categories: those designed for object-oriented systems and those designed for manipulating complex objects. This algebra belongs primarily to the second class, but has some support for the first. The first category includes the algebras of [Kort88, Grae88, Shaw89, Osbo88]. Algebras of the second sort include this one, [Abit88a], and (to some extent) LDM [Kupe85]. In this second category, the query languages are not necessarily Turing-complete, and the algebras consist of a fixed set of methods that are used in all queries. In an object-oriented system, implementation methods are associated with types or classes rather than with a certain operation (e.g., SET_APPLY or π), and the algebraic operators are used to aid in optimization when the query is expressible in terms of them. There are also more specific similarities to other algebras appearing in the literature; we mention some of the more important ones here, relating them to the six characteristics of the data model listed in Section 2:

Arrays as a type constructor: The notion of sequences supported in the NST algebra (an algebra for office documents; see [Guti89]) is similar, but not identical, to our notion of arrays, but NST does not support unordered sets in addition to arrays. Our operators can also manage both fixed- and variable-length arrays, and 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.

Optional Identity: No algebra we are aware of has analogs of our DEREF and REF operators. [Abit89] defines two separate languages, one supporting object identity and one not, but we mix the two semantics in a single model, and give references the status of a type constructor with the same privileges as the multiset, array, and tuple constructors.

Top-Level Objects of Any Type: Our algebra allows any structure, not just a set or multiset, to be a top-level object in the system. That is, not every database object needs to be coerced into a set if that is not how the user wants to view it. Our operators reflect this by being oriented toward a specific type constructor. No other algebra attempts to do this in a clean, consistent way. The O_2 data model provides this form of orthogonality in its type definitions (as does EXTRA) but not in an algebra. The model of [Guti89] allows only scalars and sequences of tuples to be top-level objects.

Generic Functions: [Klug82] takes a similar, but less general, approach to incorporating aggregates into the algebra. Our generic functions are defined to accept and return actual *structures* of the algebra rather than being defined to operate only on a particular column of a relation and to return a scalar value.

Inheritance: The VERSO algebra [Abit86], a nested relational algebra, also has a notion of schema compatibility, but it really has nothing to do with an inheritance hierarchy and is much less restrictive than ours. Their enforcement of Partitioned Normal Form (PNF) gives it a different definition and purpose. (PNF is a nested relational normal form which states that every relation at every level of nesting must have a key composed entirely of scalar fields.) We did not discuss the other inheritance-related formalisms of the algebra here; details are in [Vand89b].

Multisets: ENCORE [Shaw89] provides both sets and multisets, but the semantics of their multiset operations are not specified. Our multiset operators are similar to those of [Daya82], except that [Daya82] restricts itself to the relational model, includes the redundant intersection operator, and models (in the algebra) a multiset as a set of entities with associated integer counts. In our algebra the cardinalities of the elements are not explicitly stored.

Finally, our SET_APPLY and ARR_APPLY operators were actually inspired by certain constructs of the LISP programming language, but other algebras have similar operators: The complex object algebra of [Abit88a] has a "replace" operator which restructures every object in a set in a potentially arbitrary way, but most of the power of that language lies in this operator. Our SET_APPLY and ARR_APPLY are merely looping constructs. Probe's PDM algebra [Daya87] provides the "apply_append" operator, which applies a function to each element of a set of tuples and appends the result to each tuple. The NST algebra's λ operator is similar to "apply_append". [Vand89a] contains an extensive bibliography.

8. Summary and Future Work

We have presented an algebraic formulation of the capabilities of the EXTRA data model and EXCESS query language discussed in Sections 1 and 2. We will be able to algebraically optimize queries involving arrays, optional object identity, multisets, top-level objects of any type, generic multiset/array functions, and the EXTRA inheritance hierarchy. This is an entirely new capability, and should provide valuable insight into the nature and utility of algebraic query optimization. The large number of operators will provide a correspondingly large number of transformation rules and thus many opportunities for optimization. This algebra can simulate the relational algebra and the versions of nested relational and complex object algebras that do not provide the powerset operator. This operator was not included here since EXCESS is not capable of describing the queries that require its use (i.e., queries which actually need to construct all subsets of a given set). In one sense this is beneficial, as it eliminates a source of potentially exponential (in terms of execution time) queries. In another sense, it eliminates one way of modeling

recursive queries in the algebra, and if EXCESS is ever extended to support recursion we may wish to re-examine this decision (even though it may be possible to model recursion in the algebra in other ways).

The primitive nature of the algebra operators has several implications for our current and future work. First, an initial implementation of the system should be simple given the above operator definitions. Second, it should be relatively easy to describe, informally, the class of queries expressible in the algebra when an operator or set of operators is removed from it. Third, an algebraic optimizer needs to provide enough, but not too many, opportunities for query transformation. The algebra as is may provide too many (in the sense that all such transformations need to be examined at optimization time, and a slow optimizer is self-defeating in the presence of ad-hoc queries), but the point is that it can be tuned to vary the number and type of alternative plans examined. This can be done by defining composite operators in terms of the primitive ones described here as well as transformation rules based on these composite operators. If such composite operators and rules are used whenever possible in place of a larger number of more primitive operators, the number of alternative plans shrinks. An example of this is the relational join operator. This is often formally described as a combination of the select and cross-product operators [Ullm82]. Collapsing two operators into one like this reduces both the size of the query tree and the number of alternative access plans. Finally, this algebra allows us to experiment with alternative definitions of such higher-level operators. For example, it is unclear exactly how an analog to the relational join operator should be described in terms of more primitive operators. There are probably several alternatives, and we will test each of them.

Finally, a decision needs to be made regarding the distinction between the $\cup$ and $\uplus$ operators; namely, which one should be used for conventional query processing? The $\cup$ operator has the advantage of behaving like set-theoretic union when duplicates are not present, but its method of partial duplicate elimination seems less intuitive than that of the $\uplus$ operator. This decision will also affect the behavior of the SET_COLLAPSE operator, which is currently defined in terms of $\cup$ but could be changed.

Our current and future work includes devising a full set of algebraic transformation rules, methods to implement the operators, functions describing the cost of each operator/method, and functions estimating selectivity information for use by the optimizer. All of these are needed as input to the EXODUS optimizer generator [Grae87], which will be used to produce the executable optimizer. Implementation of EXTRA/EXCESS (using the EXODUS extensible DBMS toolkit) is well under way. The parser is complete, and the implementation of EXTRA (the data definition language) is also basically complete. EXCESS and the optimizer are now being implemented. We will also attempt a determination of the expressive power of the algebra in terms of Chandra's hierarchy [Chan81,

Chan88, Hull89, Abit89] and of any useful sublanguages of the algebra that can be obtained by removing an operator or type constructor from the original algebra.

Acknowledgements

The authors are grateful to Raghu Ramakrishnan, Michael Carey, Yannis Ioannidis, Guy Lohman, and Peter Pistor for comments and insight and to the IBM Corporation, which generously supports the first author through a graduate fellowship.

REFERENCES

[Abit86] S. Abiteboul and N. Bidoit, "Non First Normal Form Relations: An Algebra Allowing Data Restructuring," *J. Computer and System Sciences* 33, 1986.

[Abit88a] S. Abiteboul and C. Beeri, "On the Power of Languages for the Manipulation of Complex Objects," Technical Report No. 846, INRIA, May 1988.

[Abit88b] S. Abiteboul and R. Hull, "Update Propagation in a Formal Semantic Model," *Data Eng.* 11(2), June 1988.

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

[Bane87] J. Banerjee, H.-T. Chou, J. Garza, W. Kim, D. Woelk, N. Ballou, and H.-J. Kim, "Data Model Issues for Object-Oriented Applications," *ACM Trans. Office Info. Sys.* 5(1), Jan. 1987.

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

[Care88a] M. Carey, D. DeWitt, and S. Vandenberg, "A Data Model and Query Language for EXODUS," *Proc. SIGMOD Conf.*, Chicago, Illinois, 1988.

[Care88b] M. Carey, D. DeWitt, G. Graefe, D. Haight, J. Richardson, D. Schuh, E. Shekita, and S. Vandenberg, "The EXODUS Extensible DBMS Project: An Overview", Comp. Sci. Tech. Report #808, Univ. of Wisconsin, Madison, Wisconsin, November, 1988.

[Chan81] A. Chandra, "Programming Primitives for Database Languages", *Proc. Conf. on Principles of Programming Languages*, 1981, pp. 50-62.

[Chan88] A. Chandra, "Theory of Database Queries", *Proc. Conf. on Principles of Database Systems*, 1988, pp. 1-9.

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

[Daya82] U. Dayal, N. Goodman, and R. Katz, "An Extended Relational Algebra with Control Over Duplicate Elimination", *Proc. PODS Conf.*, 1982.

[Daya87] U. Dayal, M. DeWitt, D. Goldhirsch, J. Orenstein, "PROBE Final Report", Tech. Report CCA-87-02, Computer Corporation of America, Cambridge, MA, 1987.

[Fish87] D. Fishman, D. Beech, H. Cate, E. Chow, T. Connors, J. Davis, N. Derrett, C. Hoch, W. Kent, P. Lyngbaek, B. Mahbod, M. Neimat, T. Ryan, and M. Shan, "Iris: An Object-Oriented Database Management System," *ACM Trans. Office Info. Sys.* 5(1), Jan. 1987.

[Gott88] G. Gottlob and R. Zicari, "Closed World Databases Opened Through Null Values", *Proc. 14th VLDB Conf.*, Los Angeles, CA, 1988.

[Grae87] G. Graefe and D. DeWitt, "The EXODUS Optimizer Generator," *Proc. SIGMOD Conf.*, San Francisco, CA, May 1987.

[Grae88] G. Graefe and D. Maier, "Query Optimization in Object-Oriented Database Systems: The REVELATION Project", Tech. Report CS/E 88-025, Dept. of Computer Science and Engineering, Oregon Graduate Center, 1988.

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

[Hamm81] M. Hammer and D. McLeod, "Database Description with SDM: A Semantic Database Model", *ACM TODS* 6(3), September 1981.

[Hull89] R. Hull and J. Su, "On Accessing Object-Oriented Databases: Expressive Power, Complexity, and Restrictions", *Proc. ACM SIGMOD Conf.*, Portland, Oregon, 1989.

[Kern78] B. Kernighan and D. Ritchie, *The C Programming Language*, Prentice-Hall, Englewood Cliffs, New Jersey, 1978.

[Klug82] A. Klug, "Equivalence of Relational Algebra and Relational Calculus Query Languages Having Aggregate Functions," *J. ACM* 29(3), July 1982.

[Kort88] H. Korth, "Optimization of Object-Retrieval Queries (extended abstract)," Dept. of Computer Sciences, Univ. of Texas, Austin, Texas, April 1988.

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

[Lecl87] C. Lecluse, P. Richard, and F. Velez, "O_2, an Object-Oriented Data Model," *Proc. SIGMOD Conf.*, Chicago, IL, 1988.

[Maie86] D. Maier, J. Stein, A. Otis, and A. Purdy, "Development of an Object-Oriented DBMS," *Proc. 1st OOPSLA Conf.*, Portland, OR, 1986.

[Mano86] F. Manola and U. Dayal, "PDM: An Object-Oriented Data Model," *Proc. Int'l. Workshop on Object-Oriented Database Sys.*, Asilomar, CA, Sept. 1986.

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

[Roth88] M. Roth, H. Korth, and A. Silberschatz, "Extended Algebra and Calculus for ¬1NF Relational Databases," *ACM Trans. Database Sys.*, 13(4), December 1988.

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

[Schw86] P. Schwarz, W. Chang, J. Freytag, G. Lohman, J. McPherson, C. Mohan, and H. Pirahesh, "Extensibility in the Starburst Database System", *Proc. 1986 Intl. Workshop on OODB*, Pacific Grove, CA, September 1986.

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

[Ullm82] J. Ullman, *Principles of Database Systems*, Computer Science Press, Rockville, Maryland, 1982.

[Vand89a] S. Vandenberg, "Practical Complex Object Algebras," in *Workshop on Database Query Optimization*, ed. G. Graefe, CSE Tech. Report 89-005, Oregon Graduate Center, Portland, Oregon, May 30, 1989.

[Vand89b] S. Vandenberg, "An Algebra for EXTRA/EXCESS Queries," preliminary thesis proposal, University of Wisconsin-Madison, July 1989.

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