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

by

Scott L. Vandenberg
David J. DeWitt

Computer Sciences Technical Report #987
December 1990

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

Scott L. Vandenberg

David J. DeWitt

Computer Sciences Department
University of Wisconsin-Madison
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.

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.

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. These features make algebraic specification a desirable thing for a data model/retrieval language. In recent years it has become apparent that the relational model is not always the right choice for a particular application [Care86,Schw86], and many new data models have been proposed [Abit88b,Lecl87,Fish87,Maie86a,Bane87,Mano86,Roth88,Sche86,Kupe85,Abit88a]. Thus an important open question is this: Exactly how far can the algebraic approach be taken? Is it a reasonable way to implement and optimize all data models, or is there some level of complexity of the data model at which the approach is no longer justified?

1.1. Contributions and Relation to Previous Work

This paper provides a partial answer to these questions by describing a viable algebraic query processor/optimizer for a data model that is considerably more advanced than the relational 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 relational algebra or in any of the more recent advanced 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; 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 nature of 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.

These novel features enable the algebra to successfully model the structures of EXTRA (the DML) 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 all other database algebras that we are aware of, 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 (e.g., a table is not a set of tables). It simply allows slightly more natural algebraic representations of some entities. This many-sortedness allows us to easily model the arbitrary structure of EXCESS types and entities. 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.

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, as well as other operators that have appeared elsewhere. The transformation rules involving these new operators are, of course, also new. Most of the other multiset transformations we provide have not been presented elsewhere either. 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 was inspired by LISP's *mapcar* function. It resembles other algebraic operators [Abit88a,Guti89,Daya87], 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 (as is needed with the ρ operator of [Abit88a], e.g.).

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. Each of these operators is new. The notion of sequences supported in the NST algebra [Guti89] is similar, but not identical, to our notion of arrays. NST does 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, REF and DEREF, as well as 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 in order to ensure that OID domains contain "legal" values (especially in the presence of multiple inheritance). 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 enforcing object identity and one not supporting it at all, but we mix the two semantics in a single algebraic language, and give references the status of a type constructor with the same privileges as the multiset, array, and tuple constructors. The LDM algebra [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 the capabilities provided by inheritance). We explore tradeoffs between this method and a more "obvious" approach.

To put this algebra in perspective, we trace the basic development of database algebras and transformation rules: The relational algebra (as presented in [Ullm89]) has five operators: select (σ), project (π), Cartesian product ($\times$), union ($\cup$), and difference ($-$). The relational algebra was followed by algebras for nested (non-first normal form) relations [Abit86, Roth88,Sche86,Aris83,Colb89,Desh87,Fisc83,Jaes82]. This model is an extension of the relational model which allows both scalar- and relation-valued attributes. Algebras for integrating set-valued fields [Ozso87] and aggregates [Ozso87,Klug82] into the relational model have also been designed. All of these algebras are direct extensions of the relational algebra. Complex object models [Abit88a,Hull87a] generalize nested relations by removing the restriction that every entity must be a set of tuples containing either scalars or other sets of tuples as attribute values, recursively. A "complex object" can use the set and tuple type constructors arbitrarily. An algebra that operates on ordered nested relations (plus tuple-valued attributes and ordered sets of scalars) was proposed in [Guti89]. The ALGRES algebra [Ceri87] is similar but it also supports a least fixpoint operation and unordered relations. Object identity was added to complex structures in the LDM algebra [Kupe85] and in IQL [Abit89]. Finally, the algebraic approach has also been extended to cover temporal databases [Tans89,Clif85].

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. Finally, [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. Its rules are templates that are intended to be "instantiated" in actual systems according to certain parameters of the specific data model being implemented. It also does not support several of the constructs of EXTRA.

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 [Grae87]. 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. [Fish87,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 (structures, inheritance, and methods). 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 is not intended to 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 some of the more interesting new transformation rules of the EXCESS algebra by giving some examples that make use of them. 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 the Appendix.

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 [Hull87b, 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 [Rich87] and then registered with the system, and through support for user-defined functions and procedures that are written in the EXCESS query language 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. (Note that the "complex object" of [Abit88a] has no notion of identity. There seems to be no standard terminology here. Our "complex object" is similar to the notion of an object in object-oriented systems [Bane87,Fish87,Lecl87,Maie86a,Mano86].) 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

The EXTRA data model 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

```
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: A simple EXTRA database.

the EXTRA data model. 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).

Briefly, Figure 1 defines four types, all of which happen to be tuple types. The Student and Employee types are subtypes of Person. 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 some method examples). 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. It then creates a university 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 (references to Employee structures).

2.2. The EXCESS Query Language

The EXCESS query language 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 is based on QUEL [Ston76], GEM [Zani83], POSTQUEL [Rowe87], and SQL extensions for nested relations [Dada86, Sche86]. It is designed to provide a uniform query interface to multisets, arrays, tuples, and single objects, all of which can be composed and nested arbitrarily. The language 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 few examples should suffice to convey the basic flavor of the language.

As a first example, 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
```

The second example illustrates the use of an aggregate function over a nested set of objects. The following query retrieves the name of each employee, and for each employee it retrieves the age of the youngest child among the children of all employees working in a department on the same floor as the employee's department.

```
range of EMP is Employees
retrieve (EMP.name, min(E.kids.age
        from E in Employees
        where E.dept.floor = EMP.dept.floor))
```

In this example, the variable E ranges over Employees within the scope of the min aggregate, and within the aggregate it is connected to the variable EMP through a join on Employee.dept.floor. The query aggregates over Employee.kids, which is a set-valued attribute. Here, age is assumed to be defined by a function that computes the age of a Person from the current date and their birthday, so it is a virtual field (or *method*) of the Person type.

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 precise and formal definitions of the algebra can be found in [Vand90].

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. That is, an edge from A to B signifies 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. Each node 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 type name with each node. Some of these names will correspond to named types defined by the user and others will not. Every node has a unique name. 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. This is equivalent to the statement that multisets, arrays, and references are homogeneous (contain or point to elements of the same 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 also captures the intuition that, when references are viewed as simple scalar values and not followed, every type is represented by a tree. Note that this condition implies that every schema cycle contains at least one node of type "ref".

Figure 2 shows an example 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 $order(v) = n = 0$. This is the standard notion of tuple.

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.

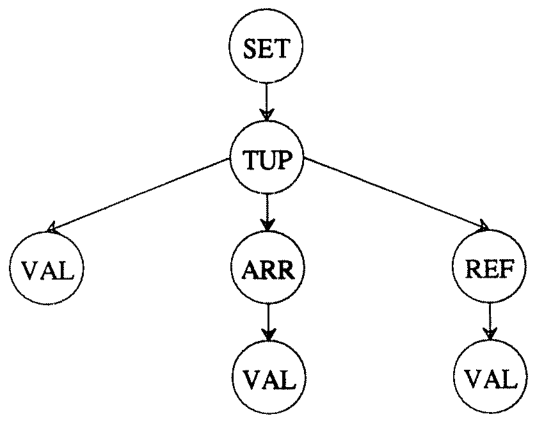


Figure 2: A Schema

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 array with no elements (it is legal for a variable-length array to be empty). Since arrays in the algebra are of varying length, the domain of an array node should contain all possible arrays of all possible lengths, including empty. So for each length (index variable j) we construct all possible arrays using the set-theoretic × operator.

v) ref: *dom(S)* = R(*S1*), where *S1* is the component of *S*. R(n), for any type n, is an infinite subset of **R**, the infinite set of all OIDs. The function R partitions **R** so that if m ≠ n, R(m) ∩ R(n) = ∅. Thus any type has an infinite set of OIDs that can only be used on objects of that type. To see that such a construction is possible, consider the set **P** of all positive integers and the (countably infinite) set **T** of all possible type names, and let f:**T**→**P** be a 1-1 function. Then simply let R(n) for the type named n be the set of all integers whose decimal representation begins with f(n) 1's followed by a 0.

An instance I of a structure with schema S is an element of *dom*(S). An example instance of the schema in Figure 2 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 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), defined as follows:

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 $1 \le i \le n$.

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 $A \rightarrow B$, then this definition assures us that arrays of A can also have B's in them, but does not provide for references to B's appearing where references to A's are expected. To obtain those semantics with the current definition we would need "**ref** $A \rightarrow$ **ref** B", which is different than $A \rightarrow B$.)

Note also that the semantics of OID domains in the presence of multiple inheritance are slightly different than those of value-based domains. We present (intuitively and formally) five rules that specify the semantics, where *Odom*(A) will indicate the domain of all OIDs for the type named A. These rules have not been presented elsewhere.

1) All domains must be infinite. Let $\Rightarrow$ represent logical implication:

$$(\forall t)\ (\, t \in \mathbf{T} \ \Rightarrow\ |\,Odom(t)\,| = \infty\,)$$

2) The portion of a domain that remains after subtracting the domains of all its subtypes must be infinite. 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 the OIDs available for S are also OIDs available for R (every object of type S is also an object of type R). In our formalism, this becomes:

$$R \rightarrow S \ \Rightarrow\ Odom(S) \subset Odom(R).$$

4) If R and S share no descendants in the type hierarchy then they have no OIDs in common. Let $\rightarrow^*$ 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 the OIDs of all types in B are also OIDs for all types in A. Formally, let $A = A_1, ..., A_n$ and $B = B_1, ..., B_m$ 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 the rule is as follows:

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

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

v') ref: $dom(S) = R(S1) \cup (\bigcup_{i=1}^{n} R\,(S_i))$, where we have $S1 \rightarrow S_i$ 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 EXCESS (and those 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 Appendix 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. [Vand90] contains many queries exemplifying the utility of specific operators; we omit most of them here for brevity and present a few examples in Section 3.3. 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

A multiset consists of a number of distinct *elements*, each of which has a certain number of *occurrences* (a *cardinality*) in the multiset. Two multisets are equal iff every element appearing in either multiset has the same cardinality in both. The first four operators below are of most interest to this discussion.

There are 8 fundamental multiset operators: 1) The **additive union** operator ($\uplus$) forms the union of two multisets by summing the cardinalities of an element in the inputs to obtain its cardinality in the result. 2) **Set creation** (SET) is used to create a singleton multiset out of any algebraic structure (it thus does not require a multiset input, like the others in this section). It returns a multiset containing its input. 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 the SET_APPLY operator. SET_APPLY 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. As an 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. The set { 1 } is subtracted from each such occurrence to obtain the result. 4) To facilitate aggregate and other computations, **grouping** (GRP) is introduced as an algebraic primitive. 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. 5) **Duplicate elimination** (DE) converts a multiset into a set; the cardinality of each element in the multiset becomes 1. 6) The **difference** operation (–), 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 produces) 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 [Abit88a].

3.2.2. Tuple Operators

There are four primitive operators on tuples. (Actually, π is expressible in terms of the other three, so it is not "primitive" in the sense of "indispensable".) 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) We use TUP_CAT to **concatenate** two tuples. One important use of this operator is to help simulate the relational cross product operator. 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 the π operator, which removes some fields from the tuple but still produces a tuple. 4) The TUP operator 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 listed below are the most relevant to our purposes.

1) To **create an array** we use the ARR operator. ARR takes a single structure (of any type, not necessarily another array) and makes a 1-element array out of it. Its utility is similar to that of the SET operator. 2) The ARR_EXTRACT operator is used to **extract an occurrence** from an array. It takes an array and returns a single

element of it as a structure. 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 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 $[E(a_1), ..., E(a_m)]$. 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. The bounds are integers ≥ 1 or the special token "last", indicating the current last element of the array. 5) The ARR_CAT operation is used for **concatenation**. ARR_CAT takes two arrays and produces an array whose elements are all elements of the first array (in order) followed by all elements of the second array (in order).

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 (except for their owners; see [Care88a]). Two operators are available on references: dereferencing and creation of a reference from an existing object. 1) The **dereference** operator (DEREF) 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) 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 operation is defined for all input structures R. This might be useful, for example, if we wish to create and 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,Abit88a]). That is, a predicate is an operation (called COMP) that returns its (unmodified) input exactly when the predicate is satisfied (true). Otherwise COMP returns the null value *dne* [Gott88]. (The model also supports an *unk* null, which we do not discuss here. *Dne* nulls are discarded whenever possible during query processing — for example, a relational selection is easily simulated because *dne* nulls appearing in a multiset are ignored.) The null constants are given a semantic interpretation similar to that of [Gott88]. COMP 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 [Ullm89], but it operates on a single structure rather than a set of tuples. If the predicate (P) evaluated with the input structure S is true, then $COMP_P(S) = S$. 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*.

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. For 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 the result of the comparison is COMP_E(A) = (1 4 6 4 1) = A. 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.

3.3. Algebraic Query Examples

This subsection is intended to give the reader the flavor of the algebraic queries and to illustrate the utility of some of the operators. We present two EXCESS queries and an algebraic (but not necessarily optimal) 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), meaning of course that the input of B is the result of A. More examples can be found in [Vand90].

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. 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 EXCESS algebra was designed to implement the EXCESS query language, not to reflect a database-style calculus such as those of [Banc86,Abit88a]. Thus the interesting question of expressive equivalence for this algebra

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

$$\begin{tikzcd}
\mathrm{SET\_APPLY}_{\pi_{name}(\mathrm{INPUT})} \\
\mathrm{SET\_APPLY}_{\mathrm{DEREF}(\mathrm{TUP\_EXTRACT}_{dept}(\mathrm{INPUT}))} \arrow[u] \\
\mathrm{SET\_APPLY}_{\mathrm{COMP}_{\mathrm{TUP\_EXTRACT}_{city}(\mathrm{INPUT})=\text{"Madison"}}(\mathrm{INPUT})} \arrow[u] \\
\mathrm{SET\_APPLY}_{\mathrm{DEREF}(\mathrm{INPUT})} \arrow[u] \\
\text{Employees} \arrow[u]
\end{tikzcd}$$

Figure 4: Query Example 2

is not whether it can express exactly the queries of some formal calculus but whether it can express exactly the queries of EXCESS. It is, of course, crucial that any EXCESS query be expressible in the algebra. The other direction of the equipollence is also 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 be applied to EXCESS as well. Due to lack of space, we only sketch the proof here.

Theorem:

The EXCESS query language and algebra are equipollent.

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). It is interesting to note, however, that the algorithm works basically like one of the methods for translating a QUEL-like relational query into relational algebra [Kort86]: everything in the retrieval list is combined using either joins or cross-products, then the criteria of the "where" clause are applied, then the actual information desired is "projected" to form the final result. Some of the complications in EXCESS arise because elements of the retrieval list are not merely attributes of a range variable — they now correspond to queries constructed using SET_APPLY, TUP_EXTRACT, DEREF, and ARR_EXTRACT (among other operators). These queries and queries for "where" clause comparands are built up using queries that represent the appropriate range variables.

ii) Reduction of algebra to EXCESS: The other direction of the proof is a traditional case-based inductive proof like those found in [Ullm89,Abit88a,Roth88]. We omit most of the cases of the inductive step as our goal here is simply to give a flavor of the proof and to demonstrate that the proof is very straightforward, due mainly to the simplicity of the algebraic operators and their resemblance to constructs in EXCESS. 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 of the same names (this is possible via the induction assumption and the "`retrieve ... into`" statement). 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 set 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 because of the induction assumption; <E1> indicates the EXCESS statement(s) corresponding to algebra expression E1). Finally, this function is invoked on the elements of E2 (any type may have a function defined on it, and the function is invoked using the dot notation, whether the type is a tuple type or not).

```
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 X2 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 attempted a proof of this yet. Such a result would provide an important upper bound for 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 at least some algebras) it allows for the formulation of least fixpoint queries in the algebra [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]. We emphasize that such loops are fundamentally different from the style of loop exemplified by 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 (multiple) inheritance hierarchy for tuple types. Both attributes and methods are inherited by a subtype. A method, in EXTRA/EXCESS, is simply an EXCESS statement (or sequence of them) defined to operate on structures of a certain EXTRA type and returning a structure of some EXTRA type. (In some proposals [Grae88,Kort88], 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 an EXCESS method is defined, it is translated into an algebraic query tree that will execute the method. 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 [Maie86b,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; some examples of this can be found in [Beer90]. Thus we want to be able to optimize the algebraic query tree for the method while taking such query-specific information into account.

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 each of the types Student and Employee. For such overriding we require that the type signatures of all the methods be identical (although it may be possible to relax this slightly). We add the following statements:

```
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) Student and Employee structures 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" given a particular Person, we check (at run time) the actual type of the Person and then invoke the appropriate query tree (possibly with some additional run time optimization). The necessary information could be specified to the appropriate algebraic operator (either SET_APPLY or ARR_APPLY) by providing a "switch table" that, given a type, returns a pointer to the appropriate query tree to invoke. This switch table could be implicitly associated with the set P, eliminating the need to have such a parameter present in the algebra.

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 the results using $\uplus$. In the above example, the initial query tree would look like Figure 5 ($\uplus$ is binary). 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 3.3). This query can now be optimized at compile time, and can take advantage of any transformations involving <Pbody>, <Ebody>, and <Sbody>. The query plan will become invalid if the type hierarchy of Person changes. An easy initial improvement to this strategy would be to do only as many SET_APPLYs as there are distinct method implementations. E.g., if Student did not redefine f(), only 2 SET_APPLY's would be needed (one for Employee, one for Person/Student).

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 certainly be preferable to scanning P three times, as would be required in the second approach.

However, if "f" is extremely complicated, the ability to optimize the entire query will be beneficial. In particular, if an overridden method involves scanning a component set or array that is much larger than the containing set or array, the cost of scanning the containing set or array several times becomes negligible. For example, if P is very

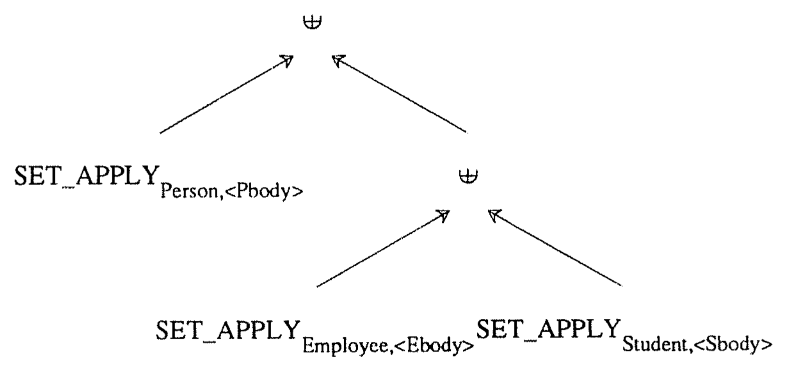


Figure 5: A $\uplus$-based approach to overridden method invocation

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 of P, and an index on the Persons of P, the need to scan P three times when using the $\uplus$-based approach disappears.

5. Algebraic Transformations

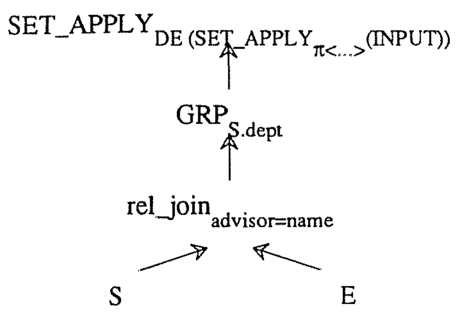
This section describes a few of the new transformation rules that can be used to optimize EXCESS queries and illustrates their use via example queries. A more complete (but still partial) list of the new rules is in the Appendix, which also lists a few rules that are familiar from the relational model to facilitate comparisons. The algebra is capable of simulating nearly all the transformations found in the literature (see Section 1), but here we emphasize the 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 of them represents an alternative execution strategy to be examined by the optimizer. Some of the trees are obtained using heuristics that are always beneficial, but the value of other transformations may depend heavily on the nature of the data. In these examples we take some liberty with the details of the algebra in order to clarify the presentation, but we lose none of the essence of the queries.

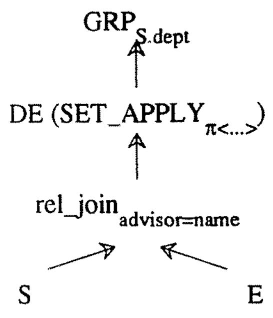
Example 1:

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 part 1 of the Appendix), then groups the result (some details are omitted form this node), performs the final projection (details omitted here also), and eliminates duplicates. Figure 7 shows the application of a rule that pushes DE ahead of grouping (rule 8 in the Appendix); 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 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 (variants of rule 7 from the Appendix and a relational rule). This results in DE operating on $|S| + |E|$





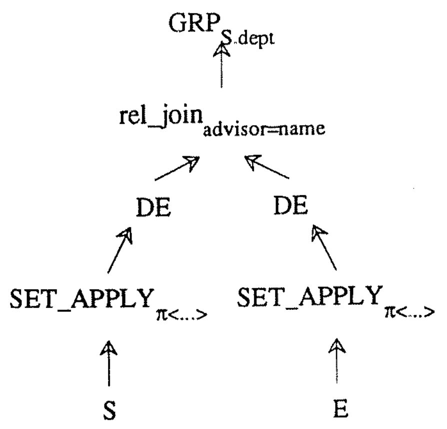


Figure 6: Ex. 1, Initial **Figure 7**: Ex. 1, 1st Transformation **Figure 8**: Ex. 1, 2nd Transformation

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
```

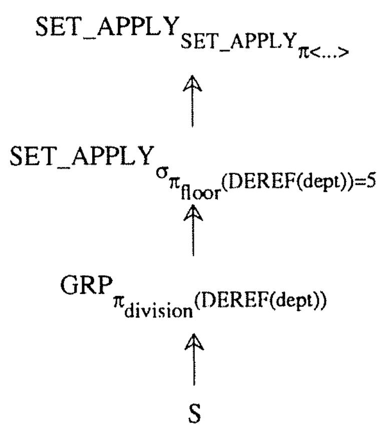

SET_APPLY SET_APPLY π<...>
SET_APPLY σ π floor (DEREF(dept))=5
GRP π division (DEREF(dept))
S

Figure 9: Ex. 2, Initial

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 project the name field. The top three nodes in Figure 9 omit some non-essential details. Figure 10 shows one way of optimizing the query: successive SET_APPLYs are collapsed, twice, to get this query (see rule 15 of Appendix). 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, which is simply $\text{SET\_APPLY}_{\text{COMP}}$. 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 (rules 10 and 26 of the Appendix). 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 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 (it does not always help) 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

$$\begin{tikzcd}
\text{SET\_APPLY}_{\text{SET\_APPLY}_{\pi_{\text{name}}(\text{COMP}_{\pi_{\text{floor}}(\text{DEREF(dept))=5}}(\text{INPUT}))}} \\
\text{GRP}_{\pi_{\text{division}}(\text{DEREF(dept))}} \arrow[u] \\
\text{S} \arrow[u]
\end{tikzcd}$$

Figure 10: Ex. 2, 1st Transformation

$$\begin{tikzcd}
\text{SET\_APPLY}_{\text{SET\_APPLY}_{\pi<\ldots>}} \\
\text{GRP}_{\pi_{\text{division}}(\text{dept})} \arrow[u] \\
\text{SET\_APPLY}_{\text{COMP}_{\pi_{\text{floor}}(\text{dept})=5}(\pi_{\text{name,DEREF(dept)}}(\text{INPUT}))} \arrow[u] \\
\text{S} \arrow[u]
\end{tikzcd}$$

Figure 11: Ex. 2, Alternative 1st Transformation

dept attribute is 5, the comparison holds. Notice that if the DEREF in this query were instead a more complicated subquery, the advantage of this particular optimization 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 (as well as arrays, multisets, and tuples) 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 full system is expected to be running shortly. The algebraic rules discussed in Section 5 and the Appendix are amenable to specification in the input language of the optimizer generator.

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, using results presented in [Hull89,Chan88].

Acknowledgements

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

[Aho79] A. Aho and J. Ullman, "Universality of Data Retrieval Languages", *Proc. Conf. on Principles of Programming Languages*, 1979.

[Aris83] H. Arisawa, K. Moriya, and T. Miura, "Operations and the Properties on Non-First-Normal-Form Relational Databases," *Proc. VLDB Conf.*, Florence, Italy, October, 1983.

[Banc86] F. Bancilhon and S. Khoshafian, "A Calculus for Complex Objects," *Proc. PODS Conf.*, Cambridge, MA, March 1986.

[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.

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

[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.

[Ceri87] S. Ceri, S. Crespi-Reghizzi, L. Lavazza, and R. Zicari, "ALGRES: A System for the Specification and Prototyping of Complex Databases," Tech. Report 87-018, Dipartimento di Elettronica, Politecnico di Milano, 1987.

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

[Clif85] J. Clifford and A. Tansel, "On an Algebra for Historical Relational Databases: Two Views", *Proc. ACM SIGMOD Conf.*, Austin, TX, 1985.

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

[Colb89] L. Colby, "A Recursive Algebra and Query Optimization for Nested Relations", *Proc. ACM SIGMOD Conf.*, Portland, Oregon, 1989.

[Dada86] Dadam, P., et al, "A DBMS Prototype to Support Extended NF^2 Relations: An Integrated View of Flat Tables," *Proc. of the 1986 SIGMOD Conf.*, Washington, DC, May 1986.

[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.

[Desh87] V. Deshpande and P.-A. Larson, "An Algebra for Nested Relations," Research Report CS-87-65, University of Waterloo, Dec. 1987.

[Fisc83] P. C. Fischer and S. J. Thomas, "Operators for Non-First-Normal-Form Relations," *Proc. IEEE COMPSAC*, 1983.

[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.

[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, June 1988.

[Hull87a] R. Hull, "A Survey of Theoretical Research on Typed Complex Database Objects", in *Databases*, ed. J. Paredaens, Academic Press, London, 1987.

[Hull87b] Hull, R., and King, R., "Semantic Database Modeling: Survey, Applications, and Research Issues", *ACM Comp. Surveys* 19, 3, Sept. 1987.

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

[Jaes82] G. Jaeschke and H.-J. Schek, "Remarks on the Algebra of Non First Normal Form Relations," *Proc. ACM PODS Conf.*, Los Angeles, CA, 1982.

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

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

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

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

[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.

[Maie86a] D. Maier, J. Stein, A. Otis, and A. Purdy, "Development of an Object-Oriented DBMS," *Proc. 1st OOPSLA Conf.*, Portland, OR, 1986.

[Maie86b] D. Maier and J. Stein, "Indexing in an Object-Oriented DBMS," Tech. Report CS/E-86-006, Oregon Grad. Center, Beaverton, Oregon, May 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.

[Ozso87] G. Ozsoyoglu, Z. Ozsoyoglu, and V. Matos, "Extending Relational Algebra and Relational Calculus with Set-Valued Attributes and Aggregate Functions," *ACM Trans. Database Sys.* 12(4), Dec. 1987.

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

[Rich87] Richardson, J., and Carey, M., "Programming Constructs for Database System Implementation in EXODUS," *Proc. of the 1987 SIGMOD Conf.*, San Francisco, CA, May 1987.

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

[Rowe87] Rowe, L., and Stonebraker, M., "The POSTGRES Data Model," *Proc. of the 13th VLDB Conf.*, Brighton, England, Aug. 1987.

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

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

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

[Ston76] Stonebraker, M., Wong, E., and Kreps, P., "The Design and Implementation of INGRES," *ACM Trans. on Database Sys.* 1, 3, Sept. 1976.

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

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

[Vand90] S. Vandenberg and D. DeWitt, "An Algebra for Complex Objects with Arrays and Identity", Tech. Report #918, Computer Sciences Dept., University of Wisconsin, Madison, Wisconsin, March 1990.

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

APPENDIX

This appendix lists a subset of the algebraic transformation rules that can be used to optimize EXCESS queries. We omit the validity proofs for these transformations as most of them are straightforward algebraic and multiset-theoretic manipulations.

1. Some Derived Operators

The multiset union and intersection operators assign cardinalities for result elements by taking the max and min, respectively, of their input cardinalities. (Recall that the $\uplus$ and $-$ operators use the sum and difference of the input cardinalities.) Note that the simplicity of the operator definitions leads to simple transformation rules. More complicated rules are consequences of those presented below (see Section 4 of Appendix).

Multiset union: $A \cup B = (A - B) \uplus B$

Multiset intersection: $A \cap B = A - (A - B)$

Relational-like Θ join: $\text{rel\_join}_{\Theta}(A, B) =$

$$\text{SET\_APPLY}_{\text{COMP}_{\Theta}(\text{INPUT})}\,(\text{SET\_APPLY}_{\text{TUP\_CAT(field1, field2)}}(A \times B)),$$

where Θ is f_1 <op> $f_2 \wedge \ldots \wedge f_{n-1}$ <op> f_n, the <op>s are of the appropriate forms, and the f_i are arbitrary algebra expressions. In addition, any conjunct may be negated. The TUP_CAT is necessary because the result of $\times$ is a set or ordered pairs. Here we use "field1" as a shorthand for $\text{TUP\_EXTRACT}_{\text{field1}}(\text{INPUT})$, etc. This kind of join follows the definitions of [Kort86]. An equi-join-like operator is obtained by restricting Θ to be $f_1 = f_2 \wedge \ldots \wedge f_{n-1} = f_n$, where the f_i are of the form $\text{TUP\_EXTRACT}_j(\text{INPUT})$.

Multiset selection: $\sigma_E\,(A) = \text{SET\_APPLY}_{\text{COMP}_E}\,(A)$. This and the next operator are similar to relational selection.

Array selection: $\alpha_E\,(A) = \text{ARR\_APPLY}_{\text{COMP}_E}\,(A)$

Relational-like $\times$: $\text{rel\_}\times = \text{SET\_APPLY}_{\text{TUP\_CAT(TUP\_EXTRACT}_1\text{(INPUT),TUP\_EXTRACT}_2\text{(INPUT))}}(A \times B)$

2. Some Rules for Multiset Operators

Rules 1-5 are familiar from the relational model. Rule 15 also appears in [Beer90], and rule 13 is a generalization of another rule from [Beer90] which only accounted for filters with the requisite properties, not for arbitrary algebraic expressions with the requisite properties. This list is not exhaustive.

1) Binary operator associativities:

   A <op> $(B$ <op> $C) = (A$ <op> $B)$ <op> C; op $\in \{ \uplus, \cap, \cup \}$

2) Distribution of $\times$ over $\uplus$:

   $A \times (B \uplus C) = (A \times B) \uplus (A \times C)$

3) Cross product commutativity:

$\text{rel\_}\times(A, B) = \text{rel\_}\times(B, A)$

4) Breaking down a disjunctive selection:

$\sigma_{P1 \vee P2}(A) = \sigma_{P1}(A) \cup \sigma_{P2}(A)$

5) Eliminating cross product:

$DE(\text{SET\_APPLY}_E(A \times B)) = DE(\text{SET\_APPLY}_E(A))$; E applies only to A

6) The result of grouping is a set without duplicates:

$DE(GRP_E(A)) = GRP_E(A)$

7) Distribute DE across $\times$:

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

8) Duplicates can be removed either before or after a set is grouped:

$GRP_E(DE(A)) = \text{SET\_APPLY}_{DE}(GRP_E(A))$

9) If a Cartesian product is being grouped and the grouping expression only applies to one of the inputs (A), that input can be grouped then each group can be combined with the other input (B) using $\times$:

$GRP_E(A \times B) = \text{SET\_APPLY}_{INPUT \times B}(GRP_E(A))$; E applies only to A

10) Push grouping ahead of a selection:

$GRP_{E1}(\sigma_{E2}(A)) = \text{SET\_APPLY}_{\sigma_{E2}(INPUT)}(GRP_{E1}(A))$

11) Distribute SET_COLLAPSE over $\uplus$:

$\text{SET\_COLLAPSE}(A \uplus B) = \text{SET\_COLLAPSE}(A) \uplus \text{SET\_COLLAPSE}(B)$

12) Distribute SET_APPLY over $\uplus$:

$\text{SET\_APPLY}_E(A \uplus B) = \text{SET\_APPLY}_E(A) \uplus \text{SET\_APPLY}_E(B)$

13) Distribute SET_APPLY over $\times$:

$\text{SET\_APPLY}_E(A \times B) = \text{SET\_APPLY}_{E1}(A) \times \text{SET\_APPLY}_{E2}(B)$; $E = E1(E2)$, E1 applies only to A, E2 applies only to B

14) Push SET_APPLY inside a SET_COLLAPSE:

$\text{SET\_APPLY}_E(\text{SET\_COLLAPSE}(A)) = \text{SET\_COLLAPSE}(\text{SET\_APPLY}_{\text{SET\_APPLY}_E}(A))$

15) Combine successive SET_APPLYs:

$\text{SET\_APPLY}_{E1}(\text{SET\_APPLY}_{E2}(A)) = \text{SET\_APPLY}_{E1(E2)}(A)$

3. Transformations for Array Operators

This list is not exhaustive either. Many of the multiset rules carry over to arrays; we do not list those here.

16) Concatenation associativity:

ARR_CAT (A, ARR_CAT (B, C)) = ARR_CAT (ARR_CAT (A, B), C)

17) Extracting an element from a concatenation:

$\text{ARR\_EXTRACT}_n(\text{ARR\_CAT (A, B)}) = \text{ARR\_EXTRACT}_n(\text{A}); n \leq |A|$

$\text{ARR\_EXTRACT}_n(\text{ARR\_CAT (A, B)}) = \text{ARR\_EXTRACT}_{n-|A|}(\text{B}); n > |A|$

18) Extracting from a subarray:

$\text{ARR\_EXTRACT}_p(\text{SUBARR}_{m,n}(\text{A})) = \text{ARR\_EXTRACT}_{m+p}(\text{A})$

19) Extracting from ARR_APPLY:

$\text{ARR\_EXTRACT}_n(\text{ARR\_APPLY}_E(\text{A})) = \text{E (ARR\_EXTRACT}_n(\text{A})); \text{E is not COMP}_P \text{ for some P}$

20) Combining successive SUBARRs:

$\text{SUBARR}_{m,n}(\text{SUBARR}_{j,k}(\text{A})) = \text{SUBARR}_{j+m,j+n}(\text{A})$

21) Taking a subarray from a concatenation:

$\text{SUBARR}_{m,n}(\text{ARR\_CAT (A, B)}) = \text{ARR\_CAT (SUBARR}_{m,|A|}(\text{A}), \text{SUBARR}_{1,n-|A|}(\text{B})); \text{if } m \leq |A|$

$\text{SUBARR}_{m,n}(\text{ARR\_CAT (A, B)}) = \text{SUBARR}_{m-|A|,n-|A|}(\text{B}); \text{if } m > |A|$

22) Taking a subarray from an ARR_APPLY:

$\text{SUBARR}_{m,n}(\text{ARR\_APPLY}_E(\text{A})) = \text{ARR\_APPLY}_E(\text{SUBARR}_{m,n}(\text{A})); \text{E is not COMP}_P \text{ for some P}$

4. Some Rules for Tuples, References, and Predicates

Some of the tuple and predicate rules will look familiar from the relational model, where they appeared as portions of rules for sets of tuples [Ullm89,Kort86]. Several rules are omitted.

23) Commutativity of TUP_CAT:

TUP_CAT (A, B) = TUP_CAT (B, A)

24) Distribute π over TUP_CAT:

$\pi_L(\text{TUP\_CAT (A, B)}) = \text{TUP\_CAT } (\pi_{L1}(\text{A}), \pi_{L2}(\text{B}))$; L = L1L2, L1 applies only to A, L2 applies only to B

25) Extracting a field from a TUP_CAT:

$\text{TUP\_EXTRACT}_f(\text{TUP\_CAT (A, B)}) = \text{TUP\_EXTRACT}_f(\text{A})$; f is a field of A

26) Push any expression inside COMP; this is a powerful generalization of commuting selections/projections in relational algebra:

$E(COMP_{P1}(A)) = COMP_{P2}(E(A)); P1(INPUT) = P2(E(INPUT))$

27) Combine successive COMPs into a conjunction:

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

Note that the rules for pushing relational selection and projection ahead of a relational join [Ullm89,Kort86] are consequences of rules 13, 24, and 27.

28) Invertibility of REF and DEREF:

DEREF (REF (A)) = REF (DEREF (A)) = A