Algebras for Object-Oriented Query Languages

Scott Lee Vandenberg

Technical Report #1161

July 1993

ALGEBRAS FOR OBJECT-ORIENTED QUERY LANGUAGES

by

SCOTT LEE VANDENBERG

A thesis submitted in partial fulfillment

of the requirements for the degree of

Doctor of Philosophy

(Computer Sciences)

at the

UNIVERSITY OF WISCONSIN-MADISON

1993

ABSTRACT

Algebraic query processing and optimization for relational databases is a proven and reasonably well-understood technology. This thesis presents a data model and query language that were developed in part to facilitate the study of algebraic query processing and optimization for advanced data models. It also continues the evolution of the algebraic paradigm by presenting operators and transformation rules that correspond to this data model and thus extend the paradigm in several directions. Additional research into algebraic constructs is also presented; this work was done in the context of a newer system that provides even more features than the previous one. A comprehensive survey of database algebras is included to help place the work in perspective.

Many of the results were obtained in the context of the EXTRA/EXCESS system, which was intended as a test vehicle for the EXODUS extensible database system toolkit. The EXTRA data model includes support for complex objects (via orthogonal type constructors), sharing, and object and value semantics. The EXCESS query language provides facilities for querying and updating EXTRA data, and it can be extended through the addition of procedures and functions for manipulating EXTRA schema types and generic set functions.

The algebraic constructs developed for EXTRA/EXCESS include operators and transformations supporting grouping, arrays, references, and multisets. I also propose a new approach to processing and optimizing overridden methods in the presence of multiple inheritance. The thesis presents an intuitive set-theoretic semantics for the domains of object identifiers in the presence of multiple inheritance. I constructively prove that the algebra is equipollent to EXCESS, giving a complete semantics and a translation algorithm for the EXCESS language.

In the context of a more powerful data model, the thesis will describe techniques for handling abstraction in an algebraic setting; differing notions of equality in an algebra and their relation to non-determinism; and algebraic techniques for processing and optimizing some queries over tree structures.

Taken as a whole, the techniques and results presented in this thesis indicate that the algebraic approach to query processing and optimization is both feasible and beneficial in object-oriented database systems and beyond.

ACKNOWLEDGEMENTS

I owe a great deal to the guidance and encouragement of my advisor, Professor David DeWitt. It has been a privilege to work with him during my years in Madison. Without his clear thinking, enthusiasm, understanding, and faith in me this thesis would never have happened.

Professors Michael Carey and Raghu Ramakrishnan have always made themselves available to answer my questions and to discuss my ideas, and their time is sincerely appreciated.

I would also like to thank the other members of my committee, Jeffrey Naughton, Marvin Solomon, and Michael Byrd, who all made useful comments about my work. Professor Olvi Mangasarian deserves credit for being in the right place at the right time.

I have had the pleasure of working with a large number of excellent fellow graduate students over the years, and I would like to thank the following for technical advice and for making the process of graduate school bearable: Dan Frank, Goetz Graefe, Paul Bober, Robert Netzer, Dan Schuh, Joel Richardson, Eugene Shekita, Mike Zwilling, Dan Lieuwen, and all members of the appropriately-named Slow Moving Vehicles hockey team.

More recently I have been involved in some research outside of Wisconsin, and I would also like to acknowledge the following people for their encouragement: Stan Zdonik, Gail Mitchell, Ted Leung, Bharathi Subramanian, David Maier, and Bennet Vance.

David Stemple's advice has been invaluable, and I would like to thank Arnold Rosenberg for his help in locating some theoretical results.

Finally, I would like to thank my parents and my wife, Kristin Bennett, for sticking with me during some very difficult times and some very difficult decisions. Kristin's patience, understanding, and energy have helped make the completion of this thesis a reality.

TABLE OF CONTENTS

CHAPTER 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 desirable 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 [Kent79, Care88a, Schw86], and many new data models have been proposed [Abit88b, Lecl87, Fish87, Maie86c, 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? This thesis will extend the paradigm to object-oriented models and slightly beyond, demonstrating that the paradigm is viable for advanced data models.

The basic ideas of this thesis, then, are twofold: first, to present a complete object-oriented model and query language and to describe an algebra that models it in such a way as to provide a semantics for the model and language and to allow optimizations to occur; and second, to describe certain portions of a second, more advanced data model and present algebraic formulations for several of its new features, again with the goals of useful semantics and optimizations in mind. The next sections provide some background in the data modeling and algebraic areas, respectively.

1.1. Object-Oriented Data Models

Here we assume that the reader is familiar with the basic concepts of object orientation such as identity, encapsulation, and subtyping. It is the intent of this section to describe the origins of the EXTRA/EXCESS and AQUA systems, not to define the term "object-oriented data model".

While a number of new data models have been proposed in the past few years, there appears to be no consensus on the horizon. A number of database researchers seem to believe that object-oriented database systems are the future [Fish87, Khos87, Banc88, Lecl87, Horn87, Andr87, Bane87, Maie86c], and several flavors of object-oriented models have been identified [Ditt86]. However, there is little consensus as to what an object-oriented database system should be; such systems today range from object-oriented programming languages with persistence to full database systems based on data models that would have been called "semantic data models" in the past. In fact, the former kind of object-oriented DBMS almost seems like a step back to the days of navigational data manipulation languages, as it is not obvious how one will support ad-hoc queries (or optimize accesses effectively) for such systems [Bloo87, Ullm87]. This thesis will address these issues, among others.

Another direction in data model evolution, one which has spawned such efforts as [Codd79, Dada86, Sche86, Schw86, Rowe87], is to extend the relational model in some way. A common goal of these efforts (and of this work as well) is to provide better support for complex objects and new data types than that offered by the relational model, while still retaining such important features as a powerful, user-friendly, data manipulation language. One approach to dealing with complex objects (also known as "structural object-orientation" [Ditt86]) is to provide procedures as a data type [Ston87a]. Another approach is to permit relation-valued attributes [Dada86, Sche86]. A third approach is to take a functional view of data [Ship81, Mano86, Bato87]. In addition, a common theme among many of these efforts is to extend the database system's data definition facilities with support for some form of subtyping.

The EXTRA/EXCESS system was designed partially to test the viability of the EXODUS extensible DBMS toolkit [Care88a], and partially to provide a platform for research into advanced data models. Chapter 4 will present the EXTRA data model and the associated EXCESS query language, which are intended to subsume much of the

modeling and querying functionality of object-oriented and other advanced systems.[1] The result is a synthesis and extension of ideas from other data models and systems, including GEM [Zani83], POSTGRES [Rowe87], NF^2 models [Dada86, Sche86], DAPLEX [Ship81], ORION [Bane87], Trellis/Owl [Scha86], O_2 [Banc88], STDM [Cope84], and STDM's descendant, GemStone [Maie86c]. The most important extensions include: support for complex objects based on a novel mixture of object and value semantics; a user-friendly, high-level query language reflecting this mixture; facilities to allow objects of any type to be made persistent and accessed via the query language; and an integration of user-defined types and operations with the query language at two levels, for both abstract data types and for conceptual schema-level types. Thus EXTRA/EXCESS supports many of the features commonly present in object-oriented systems and supports some additional features.

More recently, a system called AQUA (**A** **QU**ery **A**lgebra) is the result of a joint effort among researchers who have participated in the design of previous object-oriented algebras [Shaw90, Vanc92, Vand91]. AQUA has been designed to address a number of detailed modeling issues that we believe needed further work, but the primary goal of this work has been the design of an algebra that would serve as the input to a broad class of query optimizers. AQUA and the data model on which it is based are strongly typed and are designed to deal correctly and uniformly with abstract types.

Any query language or algebra must be embedded in some data model. AQUA provides a simple model that is general enough to cover the modeling concepts in other object-oriented models. We have adopted a model in which everything is an object in the sense that it has a well-defined interface and can be referenced from other objects, yet we also support value-based semantics.

Another goal of AQUA has been to support many different bulk types in a uniform manner. We have designed AQUA in such a way that it will not preclude additional bulk types like arrays or matrices. The AQUA design also addresses the problems introduced by type-specific equalities by providing special functions that deal with equivalence classes and duplicate elimination in a bulk type object based on some equality specific to the element type.

There are many current applications in which ordered types are required. Scientific applications have a need to store ordered types such as time-series data and genome sequences, and textual databases may store information that

[1] EXTRA stands for **EX**tensible **T**ypes for **R**elations and **A**ttributes; EXCESS is short for **EX**tensible **C**alculus with **E**ntity and **S**et **S**upport.

is structured as a tree. These applications store huge volumes of data and must locate information from these structures efficiently. Trees, for example, are useful for non-traditional database applications such as spatial and geometric data modeling. Chapter 6 introduces some fundamental techniques for processing and optimizing queries on trees.

Query languages and algebras support declarative retrieval from a database. They are based on a set of high-level operations over collections of objects. These operations hide the looping structure that would be present in an algorithm that executes them. By and large, these operations have been confined to manipulations on sets. While there has been some recent work on extending query languages to other bulk types like sequences [Rich92], additional research is needed. Thus AQUA provides support for the ordered data types graph, tree, and list.

The results presented in Chapter 6 are some of my individual contributions to AQUA.

1.2. Algebraic Query Processing and Optimization

As computers become larger (in capacity), smaller (in size), and more powerful, it becomes easier to do more complex things with them than could be done in the past. This is as true of database management systems (DBMSs) as of any other type of software. The central feature of a DBMS is its data model, the formal abstraction which it uses to map real-world entities onto (logical) database entities. Many different DBMSs may implement the same abstract data model (e.g. the relational model [Codd70]). As seen in the previous section, data models are becoming increasingly powerful and complex, and with this power and complexity comes the need to ensure correct and efficient execution of queries on data with increasingly complex structure.

1.2.1. Why an Algebra?

When executing a query in any database system, we wish to minimize the utilization of CPU, memory, I/O, and communications resources. Given current hardware technology (i.e., the availability of cheap memory), what we really want to minimize is a query's response time, which for centralized systems is usually dominated by the I/O transfer time. An orthogonal issue to speed is correctness. Clearly, the fastest query is useless if it returns the wrong answer (or never returns any answer at all). To achieve correctness with respect to a data model, one normally designs a calculus for manipulating the objects in a database. This becomes the definitive standard for lower-level implementation of the system in the sense that any lower-level query languages should be equivalent to the calculus in expressive power. Most of the calculi proposed for this purpose have been first-order. (A first-order

calculus is one in which variables and quantifiers may range only over set elements, not over calculus formulae.) This calculus often becomes the basis of the user-level query language for the system, as it is non-procedural in nature. It has become clear, from experience with the relational model, that it is much easier to pose queries using a non-procedural language than with a procedural language, in which the query must specify not only what data is desired, but how to go about retrieving that data.

Of course, at some point we must decide on a sequence of operations to retrieve the data physically. For this purpose there must be routines to access the data and to perform operations on them (e.g., the relational join). But going directly from a user's query to procedure calls on the actual database will generally result in very inefficient query plans. We need to be able to rearrange the operations to produce an optimal (in the sense described above) sequence of data accesses. The formality of an algebra allows us to do this. Naturally, this algebra must be equipollent (equivalent in expressive power) to the calculus and translatable into actual calls on the database. It is an intermediate language. The relational algebra performs this function in the relational model, and is one reason why the model of query processing in relational systems is so attractive. Other paradigms, such as rule rewriting strategies (for recursive queries), query graphs, tableaux, and other methods [Kim88] have been proposed to fill this role, but algebras remain the favorite, probably due to their simplicity and mathematical rigor. In addition, equipollence proofs between algebra and first-order calculi have well-known proof techniques.

The processing of a query, then, occurs conceptually in four steps: 1) translation from the calculus representation to a functionally equivalent algebra representation; 2) logical transformations of the algebraic query to standardize and simplify it; 3) generation of alternative algebraic strategies combined with alternative access and operator methods to produce plans for retrieving the data; and 4) selection of the cheapest plan based on a designer-specified set of criteria (usually a weighted sum of CPU, I/O, and (in distributed systems) communication costs).

Some systems [Shen87, King81] include another phase in the optimization process. This phase is called semantic query optimization, and consists of generating syntactically different but semantically equivalent versions of the original (calculus) query using user-supplied information such as integrity constraints. Each such semantically equivalent query is then processed as in steps 1-4 above, and the cheapest plan is chosen from among all the plans generated from all the semantically equivalent queries. Other issues regarding integrity constraints include how and when to enforce them and how to specify them; these issues are not discussed here. Other types of semantic information can be used in this phase of optimization, for example knowledge about allowable value ranges of

user-defined types or abstract data types (ADTs).

An excellent survey of query optimization appears in [Jark84]. An overview of some recent research in the area can be found in [Grae89].

1.2.2. A Brief History

The word "algebra" is derived from the Arabic "al-jabr", a word first used for this purpose by the 9th-century Arab mathematician al-Khwarizmi, and brought over into the Latin by Robert of Chester in 1145. The original Arabic word means a restoration or completion, and refers to the now common act of applying equal expressions to both sides of an equality in order to transform it into something more pleasing to the mathematician (or, in our case, the DBMS query processor).

In 1970 Codd proposed the relational data model and algebra [Codd70]. Informally, a relational system models all data as 2-dimensional tables of values. These tables are manipulated using the five operators of the relational algebra.

In 1977, Makinouchi [Maki77] proposed eliminating the flat 2-dimensional restriction, resulting in a body of work on non-first normal form (NF^2) relational models (also called nested relational models). These models allow values in a relation to be either scalars (as in the relational model) or entire relations. The relational algebra was extended with two operators to reflect the new structuring capabilities.

More recently, algebraic techniques have been applied to complex object data models [Abit88a], which extend the NF^2 models by removing the restriction that all data has to be either a relation or a scalar. In these models, data can be represented by sets (of anything), tuples (which are no longer constrained to appear only in sets), or any combination of the "set" and "tuple" type constructors (so called because they are used to construct types). In the relational model the use of these constructors was limited to a single application of the tuple constructor followed by a single application of the set constructor. In NF^2 models, this "tuple-set" sequence of type constructor applications was allowed to be nested. In the complex object models, there is no restriction on the ordering or number of applications of these constructors. There is no standard algebra for such objects, but several algebras have been proposed.

Finally, algebras have been defined for some object-oriented data models [Yu91, Shaw90, Vand91, Vanc92]. These models can be viewed as extending the complex object models with the notions of object identity,

inheritance, and methods (procedures defined to operate on objects of a certain type). There is no accepted standard here either. Most recently, AQUA extends the structures to accommodate ordered types such as trees and graphs.

1.3. Outline of the Thesis

In the following chapter, we present a detailed survey of database algebras, from the relational algebra to object-oriented algebras. Chapter 3 points out the specific algebraic deficiencies to be addressed in subsequent chapters and discusses work related to the development of the EXTRA/EXCESS and AQUA systems.

Chapter 4 presents the design of the EXTRA/EXCESS system. In this chapter we also present examples of the use of EXTRA to model data and the use of EXCESS to compose queries. The extensibility and orthogonality of the model and language are emphasized.

The algebra designed to formally model EXCESS queries is described in Chapter 5. This algebra is proved equipollent to the EXCESS query language. We discuss the operators and some of the more interesting transformations and optimizations that they enable, especially those involving grouping, arrays, references, and multisets. These are also illustrated by example. Formal definitions of the domains from which the data are drawn are also presented. The equipollence proof makes use of some interesting proof techniques and also includes a complete semantics for the EXCESS query language as well as an algorithm for translating EXCESS into the algebra.

Chapter 6 describes work done in the context of the AQUA data model, which is currently under development at Brown University and the University of Massachusetts at Amherst. In this chapter we are concerned specifically with algebraic constructs to model abstraction, varying notions of equality, and queries over trees. We present the fundamental concepts underlying the processing and optimization of selection queries over trees and forests.

In the last chapter we summarize the results of this thesis and draw some conclusions based on those results. We also include suggestions for future work in the area. Appendix A contains a list of the transformation rules of the EXCESS algebra. Appendix B presents the complete proof of equipollence between the algebra and EXCESS.

CHAPTER 2

A SURVEY OF DATABASE ALGEBRAS

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 [Dorn78]. These operations will have certain properties (e.g. commutativity) which make rearrangement of some algebraic expressions possible; this is what makes algebras so desirable for query optimization. Such equivalences are proved by demonstrating an isomorphic mapping between the sets represented by two different expressions. Notice that this definition says nothing about "updates", or changes to the set of objects to which the operators apply. This is the reason that update semantics in most systems are governed by a concept separate from the algebra, namely, integrity rules [Codd80].

A large number of recent systems operate on objects much more complex than those found in relational systems. A few of these have a corresponding algebra which they could use to optimize queries, but most do not. In addition, several "generic" algebras for non-relational systems have been proposed. These are not geared toward any particular system or implementation. (It should be mentioned that several calculi for objects with complex structure have been defined; some notable ones are found in [Banc86, Maie86a, Abit88a].) In the following sections we describe the basic concepts and operations of these algebras, regardless of their connections with actual systems. For each algebra, we will provide the following as appropriate: a short summary of the system and algebra (including goals of the system, structure of the objects operated on, and all of the algebraic operators), descriptions of any normal forms or other restrictions imposed by the model, some of the more important results about algebraic equivalences, whether the operators work on object-based or value-based structures, a rough idea of the expressive power of the algebra, whether or not there is a corresponding calculus, implementation status (where applicable and known), and examples of two algebraic queries (the same queries will be used as often as possible throughout to facilitate comparison). As the discussion moves beyond NF^2 models and systems, the implementation status and examples will largely be omitted.

We will not discuss the relative power of the underlying data models, exhaustive sets of algebraic equivalence rules, or other theoretical language issues. Many of these issues are covered in [Pare88, VanG87, VanG86, Hull87, Hull89a, Chan88]. The purpose of this survey is to convey some idea of the general nature of database algebras in

the hope of gaining some insight into why certain operations exist and of identifying common themes among these algebras.

It is inevitable that some algebras have been either omitted or described only briefly. This is by no means a dismissal of any kind, but reflects the fact that including them in this particular survey would have added little to its value and greatly to its length. Section 2.1 reviews the relational model and algebra. The next section covers nested relational algebras. Section 2.3 discusses algebras for more powerful data models, including the complex object and object-oriented models. In Section 2.4 we summarize the major features of each algebra and draw some conclusions.

2.1. The Relational Algebra

The relational model was first described in [Codd70] and became the model of choice for standard business data processing applications in the 1980s. The model consists of three things: 1) Relations, 2) An algebra to operate on those relations, and 3) Rules and guidelines for database design and maintenance (including integrity constraints, etc.). We do not discuss (3) here.

In the relational model, real-world entities are represented by tuples (or records). A tuple has a fixed integral number of named attributes (or fields). An important restriction is that these fields be filled only with scalar values (character strings are scalars for the purposes of this survey). This is known as the first normal form (or 1NF) restriction. A relation (or table) is a set of tuples with identical layouts. This layout is called the relation's *schema*. As a simple example, suppose we want to store information about books and their authors. We could have a relation consisting of tuples with 2 fields, one for the book's title and one for its author. Both fields could be character strings. See Figure 2.1 for a possible syntax defining such a relation (this syntax defines the tuple format and then creates a set of such tuples). Figure 2.2 contains an example instance of the Books relation.

Relations are manipulated using the relational algebra, which is equipollent to the relational calculus (and hence to first-order predicate calculus). We give brief descriptions of the five standard relational operators as defined in [Kort91] or [Ullm89]:

1) Union ($\cup$): Two relations can be combined into one using a standard set-theoretic union (duplicate tuples are eliminated). The two relations must have the same schema.

```
define type Book:
(
        title:    char [ ],
        author:   char [ ]
)

create Books:  { Book }
```

Figure 2.1: The Books relation definition

title	author
Don Quixote	Miguel de Cervantes
Moby Dick	Herman Melville
Wuthering Heights	Emily Bronte
Marmion	Walter Scott

Figure 2.2: An instance of Books

2) Difference (--): The expression R–S describes a relation consisting of all tuples in R which are not also in S (a standard set-theoretic difference). The two relations must have the same schema.

3) Cartesian Product (×): R×S indicates the relation whose schema consists of all attributes of R followed by all attributes of S. Each tuple of R is "concatenated" with each tuple of S to form the set of tuples in R×S. This is almost (but not quite) identical to the set-theoretic Cartesian product.

4) Projection (π): This operation, when applied to a set of tuples and given a list of attributes, returns the same set of tuples, removing from each tuple all attributes but the ones listed in the π expression. If any duplicates were created during this process, they are removed.

5) Selection (σ): Given a relation R, this operator applies a predicate to each tuple in the relation. If the tuple satisfies the predicate, it becomes part of the result of the selection expression; otherwise, it is dropped.

2.2. Nested Relational Algebras

This section describes several algebras which operate on non-first normal form relations. We will illustrate each algebra via two examples. The example queries range over a database which stores information about a collection of books (in this case, works of fiction with one author, printed by an American publisher, and consisting of one volume). There will be two top-level objects in the database, one for the set of books and one for the set of publishers. The Books and Publishers objects are structured as in Figure 2.3. We have used the notation of the EXTRA data model as it is fairly generic and utilizes reasonably intuitive syntax in its type definitions. No knowledge of EXTRA or EXCESS should be required to understand the type definitions.

```
define type Book:
(
        title:    char [ ],
        author:   char [ ],
        copies:   { ( publ_name:   char [ ],
                      num_pgs:     int4,
                      date:        int2 ) }
)

create Books: { Book }


define type Publisher:
(
        name:          char [ ],
        locations:     { ( city:          char [ ],
                           state:         char [2],
                           phones:        { ( area_code:   int2,
                                              numbers:     { ( num: char [8] ) }
                                            ) }
                         ) }
)

create Publishers: { Publisher }
```

Figure 2.3: Books and Publishers schemas

A book is represented by a tuple with two string-valued attributes, title and author, and a relation-valued attribute (copies). Each element of a copies set represents a particular physical copy of the book, and lists (as a tuple) the publisher's name, number of pages, and date of publication for that particular volume. A publisher is modelled as a tuple containing two attributes: a name and a set of locations. Each location of a particular publisher lists the publisher's city, state, and phone numbers as a tuple of degree 3. The phone numbers, in turn, are represented as a set of (area code, numbers) pairs. A set of actual numbers (represented as tuples of degree 1) is associated with each area code (presumably to avoid redundant storage of the area code).

The example database instance to be used in the following subsections appears in Figures 2.4 and 2.5 (representing the Books and Publishers objects, respectively). The example queries are as follows:

1) Find the names of all publishers of *Moby Dick* who have an office in the state of New York.

2) Retrieve the title and number of pages of every book by Herman Melville.

2.2.1. The DASDBS Algebra

The main feature of the DASDBS (Darmstadt Database System) [Sche85, Sche86, Scho86, Scho87a] is an application-independent kernel based on non-first normal form (NF^2) relations. (An NF^2 relation is one in which attribute values are not restricted to scalars--they can be relations (sets of tuples) as well.) Specific applications will

title	author	copies		
		publ_name	num_pgs	date
Don Quixote	Cervantes	Quick&Dirty Publ. Co.	936	1950
Moby Dick	Melville	Quick&Dirty Publ. Co.	594	1986
		Expensive Publ. Co.	414	1931
Wuthering Heights	Bronte	Pulp Publ. Co.	351	1977
		Expensive Publ. Co.	278	1922
Marmion	Scott	Antique Publ. Co.	377	1808
		Expensive Publ. Co.	501	1898
		Pulp Publ. Co.	400	1972

Figure 2.4: Books instance

name	locations			
	city	state	phones	
			area_code	numbers
				num
Pulp Publ. Co.	Cleveland	OH	216	555-0000
				555-0001
Expensive Publ. Co.	Philadelphia	PA	215	555-0000
	New York	NY	212	555-1234
				555-4444
Antique Publ. Co.	Boston	MA	617	555-0000
				555-9999
	Cincinnati	OH	513	555-1234
				555-4444
Quick&Dirty Publ. Co.	Baton Rouge	LA	504	555-1111
				555-2222
				555-3333
				555-4444

Figure 2.5: Publishers instance

be implemented on top of this kernel (the DASDBS can thus be classified as an extensible system). The kernel itself supports a subset of the NF^2 algebra which will be presented here. The kernel also provides transaction management facilities.

The DASDBS supports NF^2 relations at the storage structure level but not at the user interface level. The algebra operates on NF^2 relations, but the initial algebraic expression is obtained via substitution of NF^2 operators for operations from the application layer (e.g., relational operators, geographic operators, etc.). This is followed by algebraic transformations and access path selection. There is no limit to the level of nesting of the relations. (However, the earliest version of this algebra [Jaes82b] did not allow for general nested relations, but only for nesting of single attributes defined over scalar domains. That is, the domain of a column could be a scalar, a set of scalars, a set of sets of scalars, etc. However, the same collection of operations (with slightly differing semantics, of course) was supported.)

The following seven operators for the NF^2 algebra are proposed:

1) Cross-product (×): This is defined exactly as it is for the relational model; that is, R×S, where R and S are relations, is the set of all tuples t over (attrs(R) ∪ attrs(S)) such that t[attrs(R)] ∈ R and t[attrs(S)] ∈ S. It is assumed that attrs(R) ∩ attrs(S) = ∅.

2) Union (∪): This is also defined as in the relational model; it is a simple set union. The sets must have identical underlying schemas.

3) Difference (−): The difference of two relations is defined as their set-theoretic difference, as in the relational model. The underlying schemas of the two sets must be the same.

4) Selection (σ): Set comparators (=, ≠, ∈, ⊂, ⊆, ⊃, ⊇) and set constants (e.g. ∅ and { 1, 2, 3 }) have been added to the selection clause. Also, the ability to use σ and π within the selection formula has been added. Otherwise, it is the same as the relational σ.

5) Projection (π): This has been extended by allowing arbitrary algebraic expressions to appear in projection lists in order to facilitate retrieval of nested data from a relation. Thus π acts as a sort of navigator through a nested relation--once the proper location has been reached, other operations may be performed. The π operator also includes a renaming facility.

6) Nest (ν): This operator is applied to a single relation and must specify what attributes are to be nested and the name of the single attribute which will replace them. Specifically, $\nu_{A^*=A}$ (R), where A is a set of attributes and A* is a single relation-valued attribute whose tuples have members of A as columns, is a new relation, R*. R* is formed by using the attributes in (attrs(R) − A) as a key. That is, for each value x over (attrs(R) − A) that appears in R, we form one tuple of R*. In the A* attribute of this tuple we place one tuple (defined over A) for each appearance of x in R. An example will be given below.

7) Unnest (μ): This is the inverse of ν. $\mu_{A^*=A}$ (R) is obtained by replacing the single attribute A* with the set of attributes A and for each tuple of R, forming a set of tuples over ((attrs(R) − {A*}) ∪ A) such that for each of these tuples t, t[A] was a tuple in the A* attribute of this tuple of R.

An example should help to illustrate the use of the ν and μ operators. Figure 2.6 shows the result of applying the unnest operator to the "copies" attribute of the Books object; this is specified algebraically as $\mu_{\text{copies = \{publ\_name, num\_pgs, date\}}}$ (Books). The reader should verify that Figure 2.4 represents the result of applying the nest operator to Figure 2.6; i.e., $\nu_{\text{copies = \{publ\_name, num\_pgs, date\}}}$ (Books).

title	author	publ_name	num_pgs	date
Don Quixote	Cervantes	Quick&Dirty Publ. Co.	936	1950
Moby Dick	Melville	Quick&Dirty Publ. Co.	594	1986
Moby Dick	Melville	Expensive Publ. Co.	414	1931
Wuthering Heights	Bronte	Pulp Publ. Co.	351	1977
Wuthering Heights	Bronte	Expensive Publ. Co.	278	1922
Marmion	Scott	Antique Publ. Co.	377	1808
Marmion	Scott	Expensive Publ. Co.	501	1898
Marmion	Scott	Pulp Publ. Co.	400	1972

Figure 2.6: An unnesting of Books

It should be noted that only two new operators, ν and μ, have been introduced. The first five operators are the standard operators needed for relational completeness. In [Scho86], it is proved that some of these seven operators are actually redundant. A truly minimal subset need only consist of σ, $\cup$, μ, and π. The use of σ and π within selection formulae is also redundant.

There are no normal form restrictions imposed on the data in DASDBS; that is, these operators may be applied to any nested relation. This has some consequences in the realm of algebraic equivalences, as will be seen shortly.

The most important property of the μ and ν operators is that, while μ is always the inverse of ν, it is not always the case that $\nu(\mu(R)) = R$. This equality holds iff a certain functional dependency holds on the relation: all non-scalar fields must be functionally dependent on the set of scalar fields. This is known variously as Partitioned Normal Form (PNF) [Roth88, Desh87], Nested Normal Form [Hull88], hierarchical structures, and the ability to provide a set-free denotation for set-valued objects [Hull88]. Here we will refer to it as PNF. (Several other algebras discussed in the following sections do impose the PNF restriction, and it will be seen that these algebras are in reality no more powerful than the relational algebra, while the DASDBS and similar algebras are strictly more powerful than the relational algebra.) Another interesting property of ν is that this operator can be used to perform the grouping phase of aggregate function computation (but not, of course, the actual computation phase) in a fairly straightforward way.

The DASDBS algebra is also interesting in its construction of the join operator from primitive operators. Join processing is handled differently in DASDBS than in the relational model, where it is usually defined in terms of $\times$ and σ;[1] this notion is extended (but not entirely disposed of) in the DASDBS join operator. In DASDBS, joins are performed by using $\times$, $\cup$, σ, and $-$ inside projection lists. This is due to the fundamentally navigational nature of the DASDBS π operator. For example, since $\times$ is defined only for relations at the top level, some way is needed to join a top-level relation with one nested several levels deep. The π operator allows such referencing of arbitrarily deeply nested relations and attributes. An alternative would be to extend the $\times$ operator to include products between relations at different levels and optional join conditions. The join is defined as an outer join to avoid the problem of dangling tuples and to ensure the invertibility of the mapping from the application-level (e.g., relational) operators to NF^2 operators. This necessitates support for null values, which here are given the "unknown" interpretation. The closed-world assumption[2] is made in the DASDBS algebra.

As the DASDBS algebra is a fairly straightforward extension of the relational algebra, it is value-based (as opposed to object-based), just as is the relational algebra. That is, to get a handle on an object, one must use its value(s), not some abstract sort of name or identifier or surrogate for the object. In other words, the system is not object-oriented. No formal calculus has been defined for the DASDBS system, but the algebra presented here would need to be extended with an operator known as the powerset operator (Π; see subsequent sections) in order for it to be equivalent to any of the calculi defined for nested relations [Hull87]. Such a calculus would have to be strictly more powerful than the relational calculus in the sense that it would have to allow the construction of relation-valued as well as tuple-valued objects--this gives it some of the flavor of a second-order system. In particular, this calculus would be able to compute some recursive queries not normally expressible in the algebra without the powerset operator, so we see that Π adds some power to the algebra.

To illustrate the use of the algebra, we give expressions corresponding to the queries (1) and (2) described in the introduction to section 3:

1) Find the names of all publishers of *Moby Dick* who have an office in the state of New York. This is expressed as

---

[1] Codd's original proposal [Codd70] consisted of a different, but equipollent, set of operators: π, column permutation, join, composition, and restriction. Since then, however, the standard definitions have treated σ, π, $\times$, $\cup$, and $-$ as the fundamental operations [Kort91, Ullm89].

[2] The closed-world assumption (CWA) states that anything that can not be inferred from the data in the database is false.

follows.

π[π[publ_name] (σ[name = publ_name] (copies

× (π[name, σ[state = "NY"] (locations)]

(Publishers))))] (σ[title = "Moby Dick"] (Books))

A more understandable portrayal of the query is presented in Figure 2.7 (this figure and others like it are not meant to be a formal notation in any sense of the phrase, merely an intuitive aid). The results of the query can be found in Figure 2.8.

2) Retrieve the title and number of pages of every book by Herman Melville. The result is found in Figure 2.9, and the query is written thus:

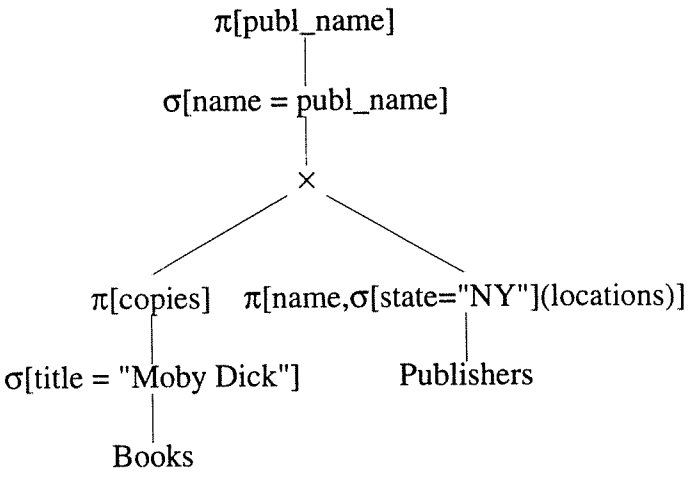

Figure 2.7: Tree representation of Query 1 (DASDBS)

publ_name
Expensive Publ. Co.

Figure 2.8: Result of query in Figure 2.7

π[title, π[num_pgs] (copies): num_pgs] (σ[author = "Herman Melville"] (Books))

In the first query, the join is performed by specifying the cross-product and the selection on name = publ_name over this cross-product. In the second query, notice that the output attribute "num_pgs" is defined using the renaming facility which is a part of the π operator. The use of the renaming facility is necessary in this situation because in general any algebra expression may appear where we have the expression "π[num_pgs]". Thus it is not always possible to use existing attribute names there. Also note that we chose to call the result "num_pgs", but we could have called it anything.

Currently, a prototype implementation of DASDBS is running and several geographical and bibliographic systems have been implemented on top of the kernel.

2.2.2. The AIM Algebra

The AIM project at IBM-Heidelberg [Dada86, Pist86, Jaes85a] is designed to handle an extension of the NF^2 data model, with the goal of supporting non-traditional database applications. The major extension is the ability to handle ordered lists (arrays), multisets, and tuple-valued attributes. Their algebra is similar to the NF^2 algebra presented in the previous section, but also incorporates features from the VERSO and SQL/NF algebras (see elsewhere in this section for descriptions of these). It is an extension of the powerset algebra described in [Jaes82a], which is essentially identical to the earliest version of the DASDBS algebra [Jaes82b]. (Briefly, the extension to 1NF relations in this early algebra consisted of the ability to have the domain of a column in a relation be $P^i(D)$, where D is a scalar domain and P^i represents the powerset operator applied i times to its argument. This algebra had the same seven operators as the DASDBS algebra.) The seven operators described above for the DASDBS system appear essentially unchanged in the AIM algebra, with the exception of selection. Also, the AIM algebra treats

title	num_pgs
Moby Dick	594
Moby Dick	414

Figure 2.9: Result of Query 2

renaming (ρ) as a separate operator, not as part of the π operator. The algebra has been published in two forms, a recursive form [Jaes85a] and a non-recursive form [Jaes85b]; here we discuss the recursive form since it is more natural considering the hierarchical nature of the data and has been proved equivalent to the non-recursive algebra. (The difference between these versions of the algebra is that the operator definitions in the recursive version are sometimes defined in terms of themselves as well as being composable with themselves. See the select and project operations for examples. The corresponding non-recursive versions are merely composable. The non-recursive version resembles some of the algebras to be discussed in subsequent sections.)

The operations are the following:

1)-6) Nest (ν), unnest (μ), projection (π), Cartesian product ($\times$), union ($\cup$), and difference ($-$): These are defined exactly as in the DASDBS algebra.

7) Selection (σ): This is similar to the DASDBS σ, but the AIM selection allows an arbitrary algebraic expression to appear in a σ formula. This mimics the SQL notion of nested queries [Astr76].

8) Keying (χ): This operator is introduced to eliminate the problem of non-invertible unnestings (see previous section). χ appends a key column to a relation before it is unnested then renested, and this ensures that nesting after an unnest will result in the original NF^2 relation [Jaes85a]. This is needed only when there is not already a key column in the relation. Note that if the AIM model supported object identity, and each tuple were regarded as an object, this operator would be superfluous.

9) Renaming (ρ): This operator renames an attribute of a nested relation. The renamed attribute may appear anywhere in the schema. Such an operator is needed only because no ordering is assumed to hold on the columns of a relation. If such an ordering were assumed, renaming would not be necessary and column numbers could be used instead.

Above we stated that the AIM project contains other extensions to the relational model besides the ones mentioned in the previous section. These, and the operators associated with them, are presented below. They were not included above as they do not form a part of the formal AIM algebra, but they do form a part of the query processing portion of the AIM system.

1) Tuple operations: These operate on single tuples. Binary concatenation appends one tuple to the end of another, and projection extracts a set of fields from a single tuple (as opposed to a set of tuples).

2) List operations: There are three operators added for list support. The n-ary concatenation operator forms one list from n lists by appending the second to the end of the first, etc. The sublist extraction operator (which has subscripting as a special case) will return a list which forms a continuous segment of another (usually longer) list. The ends of the sublist are specified by their positions in the original list. The duplicate elimination operator removes repeated elements from a list.

3) Multiset operations: These include the usual set operations $\cup$, $\cap$, and $-$, as well as restriction (selection) on multisets. There is also a duplicate elimination operator which transforms a multiset into a set. All multisets are homogeneous (contain elements of exactly one type) to preserve certain closure properties.

4) Time: Historical (as-of) queries are supported to retrieve the state of the database as of a particular time.

5) Aggregates: The standard aggregate functions (max, min, sum, count, and avg) are supported.

6) Conversions: There are operators to impose an ordering on lists and multisets, to convert between lists and multisets, to convert between 1NF and NF^2 structures, and to materialize references. Explicit and implicit constructors for lists and multisets, and explicit tuple constructors are also provided. Together, these operations allow for the explicit specification of result structures.

Even though many of the AIM extensions do not play a role in the AIM algebra, there are still some interesting points to be made about this algebra. First, null values are not supported in the formal algebra. This entails the possibility of losing information when unnesting. Specifically, if we try to unnest a tuple which has $\varnothing$ as the value of the attribute being unnested, the tuple disappears in the result, and we have lost any information it contained ($\varnothing$ is an allowable field value). Another important issue involves the χ operator, which is used to simulate a PNF requirement on the data. That is, whenever the absence of PNF within a relation can cause a problem, the keying operator may be used to artificially impose PNF on the relation. However, there are no restrictions placed on the actual data in the database. Also, it has been observed [Scho87b] that the μ operation can be modelled, optimally, using joins (we do not present the actual transformation here). Joins in AIM are defined using the $\cap$ operator as follows: if the join attributes are both scalar, two tuples will join with one another if the attribute values are equal (as in the relational natural join). If the join attributes are both relation-valued, the tuples join iff the intersection of the two relation-valued join attributes is not $\varnothing$. This definition of the join operator seems to imply that the AIM system is value-based rather than object-based, but this is only partially true. There is a system-supported facility which provides for surrogate-valued fields; a surrogate can identify any object or subobject in the database. Use of this

facility is optional, however, so that joins can be performed as just described whether surrogate values are used or not. Furthermore, the referential integrity of these surrogates is not guaranteed.

The algebra has some intriguing equivalence properties as well; we will list only a few of the more important ones here. Both the μ and π operators are commutative, for example. The ν operation, though, is commutative only if a certain multi-valued dependency (MVD) holds: $\nu_A \nu_B (R) = \nu_B \nu_A (R)$ if $D \twoheadrightarrow A$, where D = attrs(R) - {A, B}. This becomes an iff relationship if we replace the MVD by a weak MVD[3] [Jaes82b]. (Of course, these relationships also hold for the DASBDS algebra, and for any algebra with the same operator definitions.) As mentioned above, one interesting thing about the χ operator is that it enforces (albeit at a low level) the PNF restriction on the relations in the database. However, χ is only applied when needed to preserve the invertibility of the μ and ν operators. That is, other queries may operate on data which is not in PNF. This is of interest since it has been shown that restricting one's data to PNF, for both user-defined and intermediate data, reduces the algebraic power to that of the relational algebra [Hull88]. Intuitively, this is because in PNF one no longer needs to explicitly describe a set in order to get a handle on that set. That is, a set will always be denoted by a scalar value, and this scalar value will be the key for the particular level of the nested relation at which the set resides. References to sets are in this sense object-based rather than value-based when PNF is enforced. We thus have a set-free denotation of any set. In the AIM algebra, the χ operator is viewed as being invoked automatically as needed to preserve PNF, and this implies a loss of expressive power. Alternatively, [Jaes85a] could have chosen to make the χ operator optional and explicit (rather than something that happens "under the covers") and thus not give up the extra expressivity provided by the NF^2 algebra. There is some disagreement as to how restrictive PNF will actually be in real-world modelling scenarios--it may be largely a matter of taste whether one's data is more naturally modelled using PNF or an unnormalized representation.

As with DASDBS, no formal calculus has been defined for the AIM system, and no Π (powerset) operator exists in the algebra, so the same remarks about calculus equipollence apply here as well.

---

[3] A MVD $x \twoheadrightarrow y$ says that, if there are 2 sets of y values corresponding to a certain x value, these sets are the same. A WMVD says that if there are 2 such sets AND their intersection is nonempty, they are the same. Alternatively, an MVD says that if a relation R contains the tuples (x, y1, z2) and (x, y2, z1) then it also contains the tuples (x, y1, z1) and (x, y2, z2). A WMVD says that if R contains any three of the above tuples, it must also contain the fourth. The term "weak" arises because the definitions imply that all MVDs are WMVDs, but not vice-versa. For example, the relation R consisting of the tuples (x, y1, z2) and (x, y2, z1) satisfies the WMVD $x \twoheadrightarrow y$ but not the MVD $x \twoheadrightarrow y$. More intuitively, a MVD says that there is a unique set of y values associated with each x value; a WMVD says that there can be several sets of y values associated with each x value *as long as these sets are disjoint.*

Our two example queries would be formulated as follows in the AIM algebra; note the almost identical syntax to that of the DASDBS algebra (the only exception being the use of a semi-colon rather than a colon in naming the result attribute "num_pgs"):

1) Find the names of all publishers of *Moby Dick* who have an office in the state of New York (see Figure 2.8 for the results of this query):

π[π[publ_name] (σ[name = publ_name] (copies
× (π[name, σ[state = "NY"] (locations)]
(Publishers))))] (σ[title = "Moby Dick"] (Books))

A more appealing form of this query may be found in Figure 2.7.

2) Retrieve the title and number of pages of every book by Herman Melville (see Figure 2.9 for the results of this query):

π[title, π[num_pgs] (copies); num_pgs] (σ[author = "Herman Melville"] (Books))

The AIM system has been partially implemented and also includes support for user-defined data types and user-defined functions. Neither of these features is reflected in the algebra, however.

2.2.3. The Vanderbilt Algebra

The algebra for NF^2 relations described in [Fisc83] was not defined for use with a particular system, but rather as a general algebra for use with any NF^2 system, just as Codd's relational algebra [Codd70] was intended to apply to any number of relational systems. The objects operated on by the Vanderbilt algebra are exactly the NF^2 relations, as with the DASDBS algebra. That is, a relation can have atomic- and relation-valued attributes.

Schemas are defined using rules, with one rule for each relation, regardless of the level at which the relation occurs in the schema. For instance, our example Books-Publishers database would be defined as follows:

Book = (title, author, copies),

copies = (publ_name, num_pgs, date),

Publisher = (name, locations),

locations = (city, state, phones),

phones = (area_code, numbers),

numbers = (num).

The following operations can be applied to nested relations:

1) Project (π): This is defined exactly as in the relational algebra.

2) Union ($\cup$): This is also defined exactly as in the relational algebra.

3) Difference (−): This is the standard set-theoretic difference, as in relational algebra.

4) Cartesian Product ($\times$): Again this is defined as in the relational model.

5) Select (σ): This is similar to the relational σ operator but has been extended with set comparators and set-valued constants.

6) Nest (NEST): This is defined exactly as is the ν operator in the DASDBS and AIM NF^2 algebras.

7) Unnest (UNNEST): UNNEST is identical to the μ operator of the DASDBS and AIM algebras.

8) Flatten ($UNNEST^*$): This operator simulates a sequence of UNNEST operations on a relation which will transform it into an equivalent flat (1NF) relation. That is, all possible unnestings are performed. Note that the order in which these are performed is irrelevant, since $UNNEST_{A=S}\ (UNNEST_{B=T}(R)) = UNNEST_{B=T}(UNNEST_{A=S}(R))$ [Fisc83].

These operators are all of course composable and are defined to operate on any NF^2 relation(s). In other words, there are no normal form or other restrictions imposed on the data. However, since there is no mention of null values in [Fisc83], we assume that they are not supported.

A large number of algebraic equivalences for these operators are given in [Fisc83], but here we will only mention that several desirable algebraic properties do not hold without certain restrictions on the data. Among these is the commutativity of the NEST operator, as described in a previous section. While Fischer and Thomas point out the usefulness of the NEST and UNNEST operators for things such as view definitions, user-posed queries, and descriptions of logical to physical mappings of the database, they also question the fundamentality of these operators. Indeed, several research efforts have rejected these operators in favor of more "basic" operations [Aris83, Abit88a].

Joins in the Vanderbilt algebra are defined as they are for the AIM algebra; i.e., they are identical to the relational natural join when scalar attributes are involved and are defined using the $\cap$ operator when relation-valued

attributes are being joined. This exemplifies the fact that the algebra is strictly value-based. Also, it should be clear that this algebra has the same expressive power as the DASDBS and AIM algebras described in the previous two sections. The lack of a navigational π operator in the Vanderbilt algebra is made up for by making more extensive use of the ν and μ operators. Also, the UNNEST* operator does not really add any power to the language--for any particular nested relation, it is merely a shorthand for a sequence of UNNEST operations. Finally, it should be noted that no calculus corresponding to the Vanderbilt algebra has been defined.

We conclude by giving the algebraic formulation of our two example queries (the results of which may be found in Figure 2.8 and Figure 2.9, respectively) over the Books-Publishers database of Figures 2.4 and 2.5:

1) Find the names of all publishers of *Moby Dick* who have an office in the state of New York:

$$\pi_{\text{publ\_name}} (\sigma_{\text{name = publ\_name}} ((\sigma_{\text{state = "NY"}} (\text{UNNEST}_{\text{locations = (city, state, phones)}} (\text{Publishers}))) \times (\text{UNNEST}_{\text{copies = (publ\_name, num\_pgs, date)}} (\sigma_{\text{title = "Moby Dick"}} (\text{Books})))))$$

This query is more easily deciphered when presented as in Figure 2.10.

2) Retrieve the title and number of pages of every book by Herman Melville:

$$\pi_{\text{title, num\_pgs}} (\sigma_{\text{author = "Herman Melville"}} (\text{UNNEST}_{\text{copies = (publ\_name, num\_pgs, date)}} (\text{Books})))$$

Notice that the lack of nested selections and projections necessitated the use of the UNNEST operator here. (This does not mean, however, that these operators can be dispensed with in the previous algebras--there exist examples where they are necessary.) In the second example, we could have switched the UNNEST and σ operators and obtained the same result. The same is true for the second UNNEST of the first example but not the first UNNEST of that example.

2.2.4. The SQL/NF Algebra

SQL/NF [Roth87, Roth88] is an extension of the SQL relational query language to handle non-first normal form relations. SQL/NF makes some other improvements and adjustments to SQL as well, but we do not describe these here. As with the previous three algebras, the domain of interest here is the set of NF^2 relations. For the

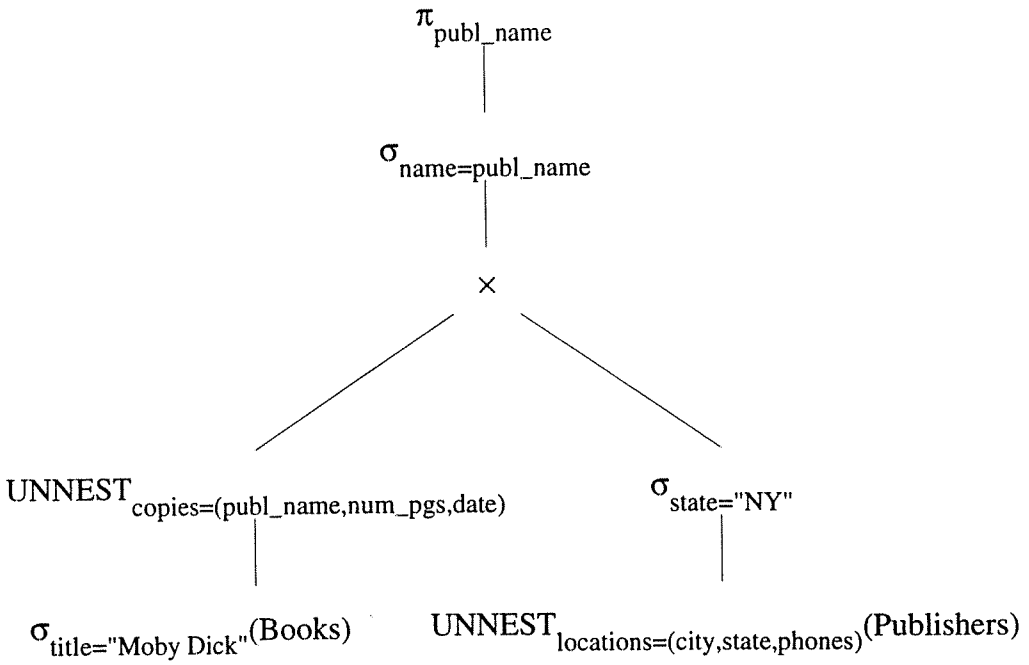

Figure 2.10: Tree representation of Query 1 (Vanderbilt)

purposes of the formal algebra for SQL/NF, database schemas are defined using rules, exactly as with the Vanderbilt algebra. One important restriction made in the SQL/NF algebra is the PNF restriction mentioned previously. This has an effect on how the operators are defined, as will be seen shortly.

The set of operators for the algebra is as follows:

1) Select (σ): [Roth88] claims not to extend the relational σ operator, but their proofs use a σ which is extended with the ability to specify set-valued constants. This is the notion of σ we adopt here. Note that no set-based comparisons have been added; only equality may be tested for.

2) Union ($\cup^e$): Union is defined recursively for pairs of relations with the same schema. For flat schemas, it works exactly as does the relational $\cup$. But when a relation-valued attribute is encountered, the $\cup^e$ operator is applied to this relation, recursively. That is, to preserve PNF, tuples with common values on the scalar fields (i.e., tuples with the same key values) will be combined into a single tuple in which the relation-valued fields will in turn be operated on by $\cup^e$. See Figure 2.11 for an example (Figure 2.11 is taken directly from [Roth88]).

R1

A	X	
	B	Y
		C D
a1	b1	c1 d1
		c1 d2
	b2	c1 d1
		c2 d2
a2	b1	c1 d1
		c2 d3
		c3 d1
	b3	c2 d2
a3	b4	c1 d2

R2

A	X	
	B	Y
		C D
a1	b1	c1 d1
		c3 d2
	b3	c4 d4
a2	b3	c2 d2
a4	b2	c1 d2
		c1 d3

R1 $\cup^e$ R2

A	X	
	B	Y
		C D
a1	b1	c1 d1
		c1 d2
		c3 d2
	b2	c1 d1
		c2 d2
	b3	c4 d4
a2	b1	c1 d1
		c2 d3
		c3 d1
	b3	c2 d2
a3	b4	c1 d2
a4	b2	c1 d2
		c1 d3

R1 $\cap^e$ R2

A	X	
	B	Y
		C D
a1	b1	c1 d1
a2	b3	c2 d2

R1 $-^e$ R2

A	X	
	B	Y
		C D
a1	b1	c1 d2
	b2	c1 d1
		c2 d2
a2	b1	c1 d1
		c2 d3
		c3 d1
a3	b4	c1 d2

Figure 2.11: Extended set operations (taken from [Roth88])

3) Difference ($-^e$): This is defined recursively in a fashion similar to the $\cup^e$ operator. That is, a tuple is kept in R1 $-^e$ R2 if it is in R1 and its scalar values do not match those of any tuple of R2. A tuple of R1 is also kept if it does match the scalar values of some tuple in R2 but the difference between at least one of its nested attributes and the corresponding nested attribute of some tuple of R2 is non-empty. In this second case what we keep is not the original tuple but the original tuple with the $-^e$ operation applied to these disagreeing nested attributes, recursively. See Figure 2.11 for an example.

4) Intersection ($\cap^e$): This is also defined similarly to $\cup^e$. Two tuples intersect if they agree on their scalar (key) attributes and the intersections of all of their nested attributes are non-empty. See Figure 2.11 for an example.

5) Cartesian Product ($\times$): This is defined as in the relational algebra.

6) Projection (π^e): This consists of a normal relational π (i.e., only a top-level attribute may be projected out, but it need not be scalar) followed by a unioning ($\cup^e$) of all the resulting tuples to remove duplicates.

7) Natural Join ($|\times|^e$): The natural join of two relations is defined only if all the common (joining) attributes are top-level attributes of the relation. Two tuples join only if they agree on all common scalar attributes and the intersections ($\cap^e$) of their common set-valued attributes are all non-empty.

8) Nest (ν): This is defined exactly as in the Vanderbilt algebra.

9) Unnest (μ): This is also defined exactly as in the Vanderbilt algebra.

The set operations ($\cap^e$, $\cup^e$, and $-^e$) apply only to sets with identical schemas.

The restriction to PNF clearly has some interesting effects on the definitions of the algebraic operators. In particular, it can lead to recursive definitions of some of them (union, difference, etc.). It also has an interesting consequence regarding the ν operator. We are assured that a ν operation results in another PNF relation iff the set of scalar attributes not being nested functionally determines the set of non-scalar attributes not being nested. It is also pointed out that, if a relation is not in PNF, it is possible to add a key column to it to enforce this property and then remove this column after all operations on the relation have been completed (as is done with the AIM χ operator). Recall that the motivation for PNF was the non-invertibility of certain applications of the μ operator. Thus when such operations arise, we can add a key column. Neither null values nor empty sets are supported in the algebra, which avoids some interesting problems regarding the unnesting of an empty set in a system not supporting nulls. Also, it is clear that, like the rest of the algebras discussed thus far, this algebra is purely value-based.

It should be noted that the extended versions of $\cup$, $\cap$, and $-$ are expressible in terms of the five standard relational operators plus nest and unnest. In other words, for any particular instance of such an operator, it can be replaced by a sequence of these seven operators (but there is no such definition of them in the general case) [Roth88].

A formal calculus corresponding to the SQL/NF algebra has been defined [Roth88] and proved equivalent to the algebra. This calculus has been proved to be strictly more powerful than the given algebra unless one adds the Π operator to the algebra. However, when we are restricted to the domain of PNF relations, as here, the algebra and calculus become equivalent. Basically, this is because not even intermediate results are allowed to violate PNF in

this algebra. In the proof of equivalence to the calculus, this property is also forced on the calculus by adding a key column as described above. The next section gives an intuitive explanation of why PNF reduces expressive power. An implementation of this algebra is in the planning stages.

Before leaving the SQL/NF algebra, we give their versions of our two example queries:

1) Find the names of all publishers of *Moby Dick* who have an office in the state of New York:

$$\pi^{e}_{\text{publ\_name}} (\sigma_{\text{name = publ\_name}} ((\sigma_{\text{state = "NY"}} (\mu_{\text{locations = (city, state, phones)}} (\text{Publishers}))) \times (\mu_{\text{copies = (publ\_name, num\_pgs, date)}} (\sigma_{\text{title = "Moby Dick"}} (\text{Books})))))$$

This query is more easily deciphered when presented as in Figure 2.12.

2) Retrieve the title and number of pages of every book by Herman Melville:

$$\pi^{e}_{\text{title, num\_pgs}} (\sigma_{\text{author = "Herman Melville"}} (\mu_{\text{copies = (publ\_name, num\_pgs, date)}} (\text{Books})))$$

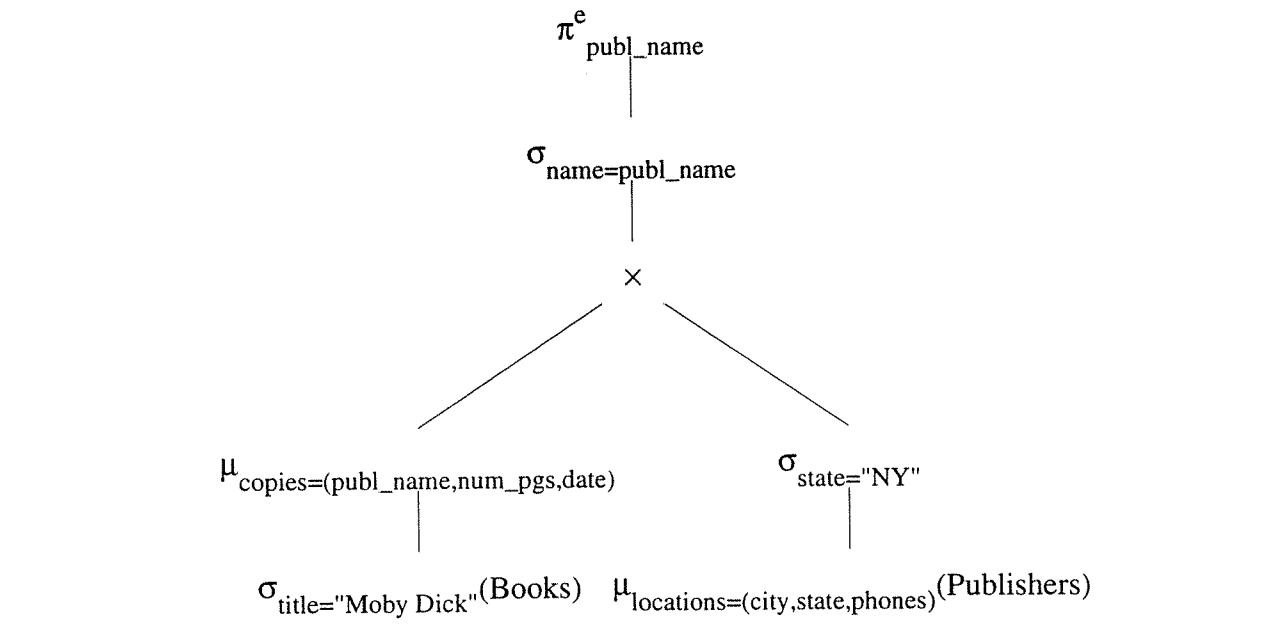

Figure 2.12: Tree representation of Query 1 (SQL/NF)

These queries are posed exactly as in the Vanderbilt algebra. Note that this is so because our schema and relations satisfy the PNF restriction. In the first query, the $|\times|^e$ operator could have been used in place of the σ-$\times$ combination; in this case the result would have been the same since the join attributes are scalar.

2.2.5. The VERSO Algebra

VERSO is a relational database machine developed at INRIA [Abit86]. The intention of the design was that all of the algebraic operations were to be performed by a filter, with the exception of the restructuring operation (see below). Not surprisingly, the VERSO algebra is defined to operate on nested relations. Like SQL/NF, however, it imposes the important PNF restriction. In addition, each relation at each level must have at least one scalar attribute. This is stricter than PNF since PNF allows a relation with no scalar attributes. In this case (no scalar attributes), there can only be one tuple in the relation, otherwise PNF is violated; the only difference in VERSO, then, is that such single-tuple relations are not allowed. There are five unary operations, five binary operations, and a restructuring operator, defined as follows:

1) Extension: This operation takes an instance of a VERSO schema (a V-relation) and transforms it into an instance of a schema which contains, at each level, at least the attributes of the schema of the original relation. The "new" parts of the relation are assigned the value $\varnothing$.

2) Projection (π): This is a recursive extension of the relational π. In VERSO, one can project out an arbitrary subformat (the VERSO term for subschema) of a V-relation. (One format is a subformat of another if the first can be obtained from the second by pruning one or more terminal subtrees of the tree representation of the second format.) Note that in order to perform arbitrary projections of fields in a format (rather than just entire subformats) we need the restructuring operator. Also note that a scalar field may not be projected out (i.e. removed) unless the entire relation (at that level) associated with it is projected out also.

3) Select: Selections are defined over single formats, recursively. At each level of the format, the standard scalar comparisons may be applied to the scalar attributes. In addition to these scalar comparisons, one may test for the existence or non-existence of the result of each nested selection, if desired (i.e., is the result non-empty or empty?). One thing to note is that set equality and membership tests are not permitted; only tests for equality to $\varnothing$ can be performed on sets. There is also an extended version of selection, known as "super-selection". This extends selection by allowing the conjunction of several nested selections to be expressed for

any nested attribute, rather than just one nested selection on this attribute. However, this super-selection can be expressed using the π, join, and regular selection operators.

4) Restriction (restrict): The result of $\text{restrict}_{A=B}(I)$, where I is a VERSO instance, is the set of all tuples of I which agree on the A and B attributes (A and B must refer to scalar attributes). This operation is not recursive, although [Abit86] mentions that a recursive version could easily be defined.

5) Renaming (rename): This operator allows one to give new names to any part of a VERSO format.

6) Union ($\cup$): This is exactly the same as the SQL/NF $\cup^e$ operator. It can also be applied to V-relations having different formats, if these formats are compatible. (Two formats are compatible iff they are both subformats of some other format. This means that the two formats must have the same scalar fields.) To perform the union (or intersection or difference; see below) on compatible formats, the two original instances are padded with empty sets to give them the right schemas, and then a $\cup$ is performed on these extended instances (see the extension operator, above).

7) Intersection ($\cap$): This is almost the same as the SQL/NF $\cap^e$ operator. The difference is that in the VERSO $\cap$, all tuples agreeing on the scalar attributes will be in the intersection, regardless of whether or not their intersections on the relation-valued attributes are empty. Thus the result of $\cap$ may have empty sets as nested attribute values. Since SQL/NF does not support empty sets, this possibility was eliminated. This operator can also be applied to compatible formats, just like the $\cup$ operator.

8) Difference (–): This is exactly the same as the SQL/NF $-^e$ operator. This operator can also be applied to compatible formats.

9) Join: This turns out to be identical to the intersection operator when the schemas of the two relations being joined are identical. But when these schema are not equal (but merely compatible), the join includes more information than an intersection: the join will also include information (attributes) that was in exactly one of the original instances. Such attributes will come through unscathed, as it were, while in the intersection for compatible types, the intersection of these attributes with the artificially extended attributes of the other instance would yield an empty result.

10) Cartesian Product ($\times$): This is defined only if the left operand is a flat relation. The result is a set of tuples, each of which has a tuple of the left input as scalar value and the right input (which may or may not be a flat

relation) as the only nested attribute. That is, for each tuple of the left input, we tack the entire right input onto the end of it. This is one tuple of the result. Note that × is thus not commutative, even in the flat relational case.

11) Restructuring (restruct): This operation takes a V-relation over a certain schema and transforms it into a V-relation over another format. The result of $\text{restruct}_{[f]}(J)$ is the largest relation over the format f which does not add any information to the V-relation J. Here we have omitted a great deal of theory which lies behind the "restruct" operator. For our purposes, it is enough to realize that it subsumes the functionality of the nest and unnest operators and that it can be used for essentially arbitrary restructuring, as long as the result format makes sense relative to the original format (where "makes sense" is given a precise formal meaning in [Abit86]). An example of a VERSO format specification is:

$$\text{title author (publ\_name num\_pgs date)}^*$$

This is our Book schema as defined earlier.

The VERSO algebra does not provide for null values. This leads to the possibility of information loss when using the restruct operator, just as certain combinations of nest and unnest can cause problems in the previously described algebras when ∅ is permitted as a value and null is not (i.e., unnesting ∅ without null values is not a well-defined process). Necessary and sufficient conditions for an information-lossless application of restruct are given which, upon closer inspection, will probably be seen to have PNF as a special case.

Note that here it is not the case that $A \cap B = A - (A - B)$, as is true in the relational algebra. This is because the intersection as defined retains the scalar-valued portion of a tuple even if all the nested intersections are empty, while the difference operator removes a tuple entirely if its nested differences with respect to some tuple in B are ∅ and its scalar parts are equal to those of that tuple.

In terms of expressive power, VERSO falls into the same category as SQL/NF. The reason for this stems from the PNF restriction; however, the reason is not this restriction itself but rather the fact that some of the operator definitions are so heavily dependent on PNF. That is, operators defined to operate on any NF^2 relation(s) will have the same expressive power whether the data happens to be in PNF or not. Intuitively, the real reason is the fact that no intermediate results in a VERSO computation can violate PNF. This makes the VERSO algebra suspiciously similar to the relational algebra, and in fact it has been shown that these two algebras are equipollent, and that PNF relations have a direct correspondence with 4NF (fourth normal form) relations. Again intuitively, this is because

the continual retention of all of the scalar (key) attributes at every level lends itself to an easy simulation using relational algebra operators; we never need to operate explicitly on sets, and can use the key property of the scalar attributes to simulate a 1NF (or 4NF) relation.

The VERSO algebra is value-based, and no calculus has been defined to correspond to it. Implementations were completed both on an SM90 computer and under MULTICS. Our two example queries appear as follows in the VERSO algebra:

1) Find the names of all publishers of *Moby Dick* who have an office in the state of New York:

$$\pi_{name}\,(\text{restruct}_{[name\ (title\ author\ publ\_name\ num\_pgs\ date)^*]}\,(\text{restrict}_{name\ =\ publ\_name}$$
$$(\text{restruct}_{[name\ title\ author\ publ\_name\ num\_pgs\ date]}$$
$$((\pi_{name}\,(name:\,(\exists(location:\,(\exists(state:\,state = \text{"NY"}))))))$$
$$\times\,((title\ author):\,title = \text{"Moby Dick"})))))$$

This query is more easily deciphered when presented as in Figure 2.13.

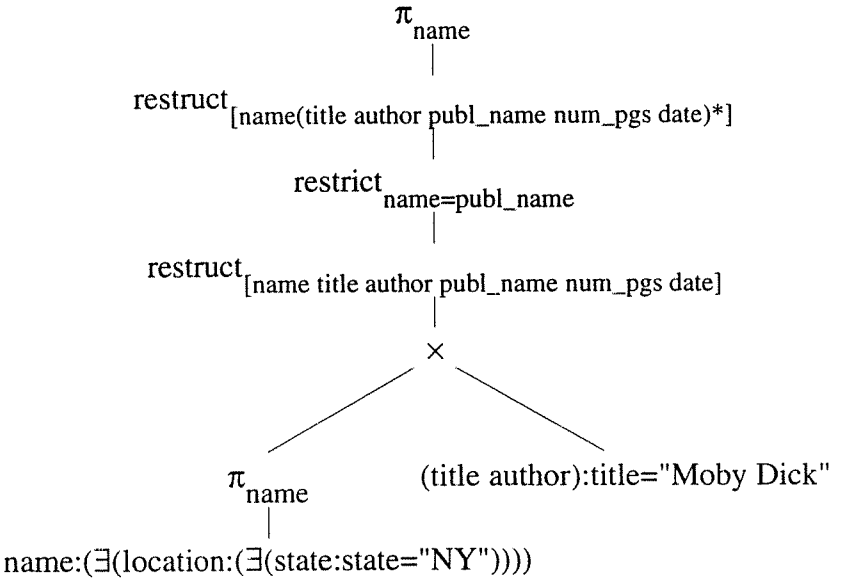

Figure 2.13: Tree representation of Query 1 (VERSO)

2) Retrieve the title and number of pages of every book by Herman Melville:

$\pi_{\text{title, num\_pgs}}$ (restruct$_{\text{[title num_pgs (author publ_name date)*]}}$

(restruct$_{\text{[title author publ_name num_pgs date]}}$

((title author): author = "Herman Melville")))

A few comments are in order here. In example 1, we could not use the join operator since the formats (schemas) being operated over were not compatible, so we used the × followed by a restriction. In order to satisfy the definition of ×, we had to ensure that the first operand was a flat relation, hence the projection on name. Then we brought the publ_name attribute to the top level so that a restriction could be performed. Finally, some restructuring was necessary to allow us to π only on the name field (since scalar attributes cannot be projected out of a result). The second example is a little simpler. It merely requires some restructuring to get the proper attributes to the top level and allow their projection.

2.2.6. The NRDM Algebra

The Nested Relational Data Model (NRDM) is a system being developed and implemented at Indiana University [Desh88]. Once again, the domain of objects is the set of relations with relation-valued attributes, just as in the previous algebras. This algebra also enforces the PNF restriction on its data, which again has an effect on the definition of the algebraic operations. These operators are defined as follows:

1) Union ($\cup^e$): This is defined in the same manner as the VERSO and SQL/NF union operators.

2) Difference ($-^e$): This is also defined as in the VERSO and SQL/NF algebras.

3) Project (π^e): In order to preserve PNF, this is defined as in the SQL/NF algebra.

4) Select (σ^e): This is extended from the relational σ to allow membership testing in sets, set comparators, and selections from any level of the hierarchical structure.

5) Join ($|\times|$): Joins are allowed only when the join attributes are atomic. Joins are identical to the relational natural join, and can occur only when the join attributes are at the top level (presumably--this is not explicitly stated).

6) Nest (ν): This works essentially as in the previous algebras, but is restricted to yield a structure with only one set-valued attribute at the highest level. That is, a ν operation must nest at least all of the relation-valued attributes at the top level of the structure, resulting in just one relation-valued attribute. This can be worked

around by joining two or more relations which have only one set-valued top-level attribute.

7) Unnest (μ): This merely inverts the ν operator.

While this has somewhat less robust join and nest definitions than some of the other algebras, it has the advantage of being implemented in a prototype version. It also enjoys some nice algebraic properties. Specifically, the PNF restriction of course allows invertible unnesting, and the restricted ν definition ensures the preservation of PNF when an arbitrary ν is performed. Null values are not supported. No calculus has been defined to correspond specifically to this algebra, but, like with the other algebras, it is not hard to envisage an adaptation of the SQL/NF calculus [Roth88] to this algebra. It also shares with all of the previously defined algebras the characteristic of being value-based. There are no formal results on the expressive power of this algebra, but it almost certainly has the same expressive power as VERSO and SQL/NF, due to the PNF restriction.

The two example queries, whose results appear in Figure 2.8 and Figure 2.9, respectively, would be expressed as follows in the NRDM algebra:

1) Find the names of all publishers of *Moby Dick* who have an office in the state of New York:

$$\pi^{e}_{\text{publ\_name}}\left((\sigma^{e}_{(\sigma^{e}_{\text{state = "NY"}}(\text{locations}) \neq \emptyset)}(\text{Publishers})\right.$$
$$|\times|_{\text{publ\_name = name}}\left(\mu_{\text{copies = (publ\_name, num\_pgs, date)}}\right.$$
$$\left.\left.(\sigma^{e}_{\text{title = "Moby Dick"}}(\text{Books}))\right)\right)$$

This query is more easily deciphered when presented as in Figure 2.14.

2) Retrieve the title and number of pages of every book by Herman Melville:

$$\pi^{e}_{\text{title, num\_pgs}}\left(\sigma^{e}_{\text{author = "Herman Melville"}}\left(\mu_{\text{copies = (publ\_name, num\_pgs, date)}}(\text{Books})\right)\right)$$

These queries are similar to the corresponding SQL/NF queries, but in the join query notice that the recursive nature of the NRDM σ^{e} operator avoided an unnesting. In the second query, it is still necessary to use the μ before doing the projection, since π is not as "extended" as σ.

2.2.7. The Waterloo Algebra

The algebra for nested relations developed at the University of Waterloo [Desh87], like that designed at Vanderbilt, was not designed with any particular system in mind, but as a general tool for studying and guiding the implementation of nested relational database systems. The Waterloo algebra, like the SQL/NF, VERSO, and

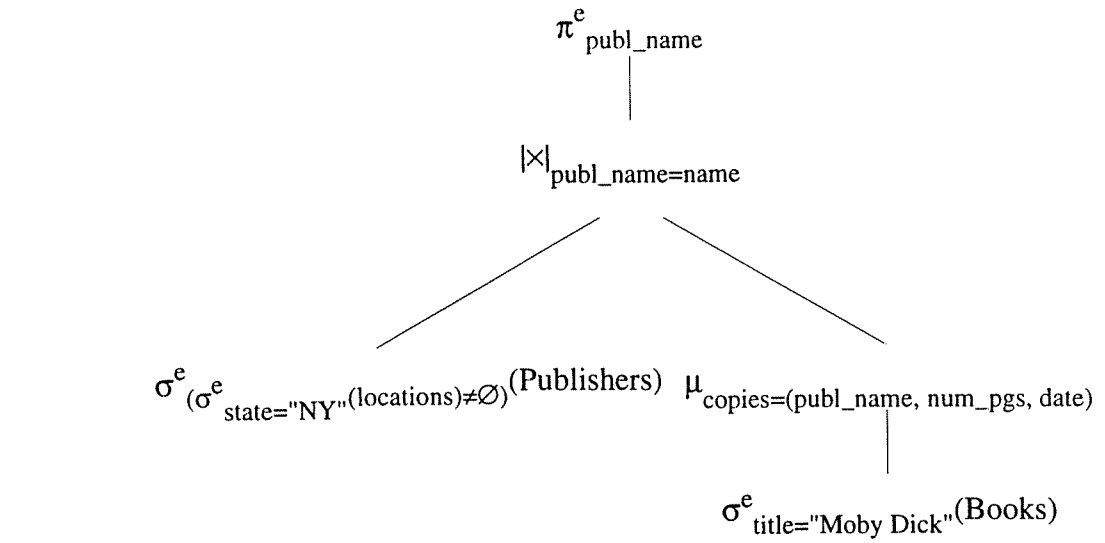

Figure 2.14: Tree representation of Query 1 (NRDM)

NRDM algebras, enforces the PNF restriction on its data. The operators of the algebra are defined as follows:

1) Union ($\cup$): This is defined exactly as is the SQL/NF $\cup^e$ operator.

2) Intersection ($\cap$): This is defined in the same way as the VERSO $\cap$ operator for identical VERSO formats. That is, the Waterloo algebra allows $\emptyset$ as a value, as does VERSO, and thus the intersection of two tuples is allowed to have empty set-valued components.

3) Difference (–): This operator is the same as the SQL/NF difference operator.

4) Projection (π): This is defined to have the same effect as the SQL/NF π^e operator; that is, only top-level (scalar- or relation-valued) attributes may be projected out, and duplicates are eliminated using the $\cup$ operator.

5) Selection (σ): Selection predicates are extended to allow set comparators ($\subseteq$, $\supseteq$, etc.). The objects being compared may be constants (scalar or set-valued), attributes which are defined within the nested relation being selected over, or an entire nested relational algebra expression which operates on relation-valued attributes which are defined within the nested relation being selected over. This is essentially the same as the DASDBS [Sche86] σ operator.

6) Cartesian Product ($\times$): This is defined exactly as in the relational algebra.

7) Nest (ν): This is defined just as in the DASDBS and AIM algebras, with the exception that at least one of the attributes not being nested must be scalar to help ensure the preservation of PNF.

8) Unnest (μ): This is defined as in the DASDBS and AIM algebras.

9) Rename (δ): This facility allows the renaming of any top-level attribute.

10) Subrelation constructor: Notice that the π operator differs from that of the DASDBS algebra in that it is non-navigational in nature. The Waterloo algebra simulates this using a fairly complicated subrelation constructor. Put simply, this operation specifies a path within a top-level relation; this path specifies a point in the structure at which we may now apply nested algebra expressions as if we were at the top level (notice the similarity in effect to the DASDBS π operator). We thus can create a "subrelation" with as many attributes as we like. Each of these attributes will be the result of some algebra expression applied to the path specified in the subrelation constructor. (Note that these algebra expressions may themselves contain subrelation constructors.)

In [Desh87] the join and natural join operators are also defined, but these are not fundamental as they are defined in terms of $\times$ and σ, just as in the relational algebra.

Note that here we have the problem with the unnesting of empty subrelations as we did in VERSO. Since null values are not allowed, and $\varnothing$ obviously can not appear as a value in a flat relation, μ may lose information, so the result of, for example, unnesting two relations then taking their union then renesting them may not be the same as simply taking the union of the original nested relations. The same problem can occur when intersection or difference is used in place of union.

The Waterloo algebra can be viewed as a combination of the DASDBS and SQL/NF algebras; however, due to the PNF restriction, it is equivalent in power to the VERSO, SQL/NF, and NRDM algebras, hence weaker than the DASDBS, AIM, and Vanderbilt algebras. This of course implies that the calculus of [Roth88] could also be adapted for use with the Waterloo algebra with only minor modifications (no calculus has been designed to correspond specifically to the Waterloo algebra). Also, like all the previous algebras, the Waterloo algebra is value-based in nature.

A system known as LauRel [Desh87] is being developed at the University of Waterloo. LauRel is based on the nested relational model and presumably will make use of the Waterloo algebra. Planned extensions to the algebra include support for aggregate computations. The LauRel system also supports references to objects, much as in [Care88b]. The user-level language of this system, called SQL/W, is designed to support null values (the original Waterloo algebra as described here is not). The LauRel system is still in the design phase.

We now give Waterloo algebra versions of our two example queries:

1) Find the names of all publishers of *Moby Dick* who have an office in the state of New York:

π[publ_name] ((σ[σ[state = "NY"](locations)] (Publishers))
|×| [publ_name = name] (μ[copies = (publ_name, num_pgs, date)]
(σ[title = "Moby Dick"] (Books))))

This query is more easily deciphered when presented as in Figure 2.15.

2) Retrieve the title and number of pages of every book by Herman Melville:

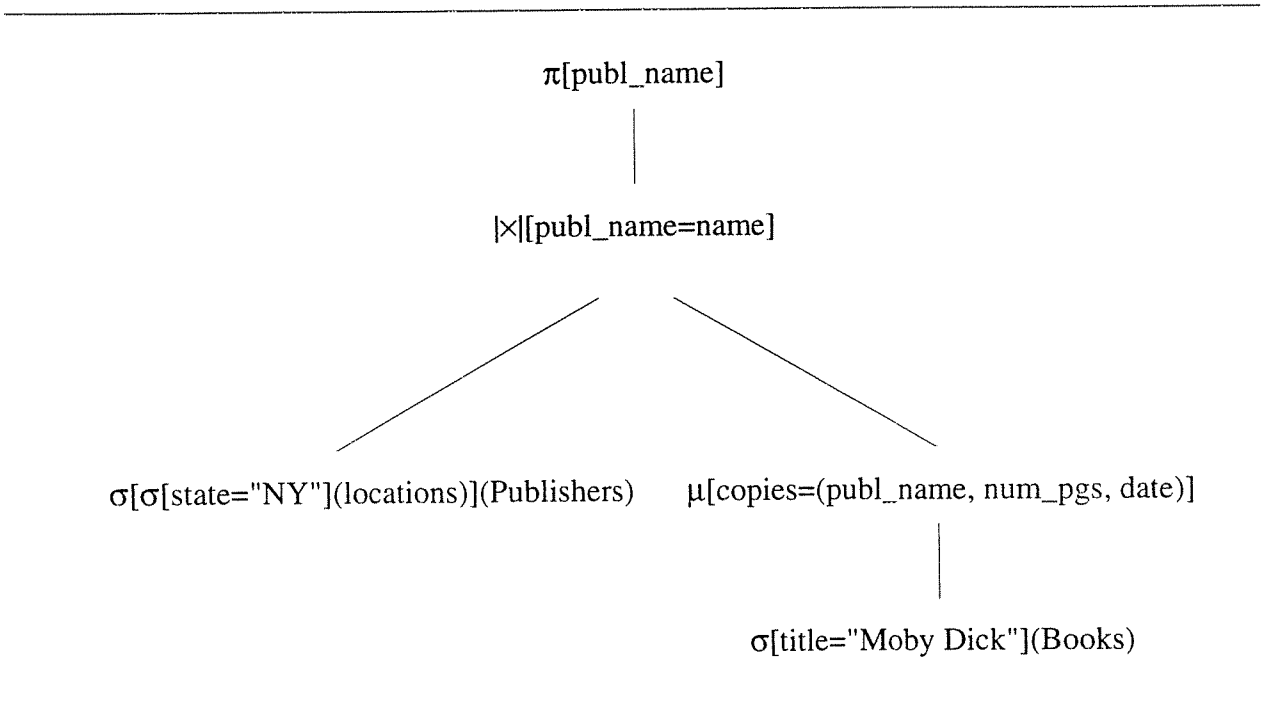

Figure 2.15: Tree representation of Query 1 (Waterloo)

π[title, num_pgs] (σ[author = "Herman Melville"]

(μ[copies = (publ_name, num_pgs, date)] (Books)))

In the first query we make use of the recursive nature of the σ operator; we also use the join operator, which could be simulated easily using σ and ×. Notice that these queries are identical (modulo syntax) to those of the NRDM algebra. This is slightly unfortunate, as the two algebras are different in several respects, but the differences are not captured by these queries. In particular, the ν and join operators are not the same.

2.2.8. The Powerset Algebra

This algebra [Gyss88] came about as a result of the fact that, when the PNF restriction is not imposed, the nested relational algebra is less powerful than the nested relational calculus [Hull87]. This algebra was developed to demonstrate certain theoretical aspects of nested relations, and is not associated with any particular system. There are no restrictions on the nested relations that can be operated upon by this algebra. The operators are:

1) Union (∪): This is defined as a set union, as in the relational algebra.

2) Difference (−): This is set difference, also as in the relational algebra.

3) Cartesian Product (×): This is the same as the relational × operator.

4) Projection (π): This is also like the relational π--only top-level attributes (of any kind) may be projected out. Duplicates are eliminated automatically.

5) Nest (ν): The Powerset algebra's ν operator is identical to the DASDBS ν operator.

6) Unnest (μ): This operator is equivalent to the DASDBS μ operator.

7) Renaming (ρ): Any attribute at any level of the schema may be renamed.

8) Selection (σ): Selection predicates can test for the equality of a top-level attribute (scalar or relation-valued) to another value (scalar or relation-valued). In the case of relation-valued comparisons, equality is tested for at all levels of the structure (deep equality).

9) Powerset (Π): This operator takes a relation (set of tuples) and produces the set of all subsets of the relation, usually written as 2^R, where R is the relation.

There are some fairly clean and interesting results concerning the power of this algebra. For our purposes, it is most important to note that, when it is not restricted to PNF relations, it is equipollent to the NF^2 calculus of

[Roth88] (no calculus was defined specifically for the Powerset algebra). It is also interesting to note that the Powerset algebra is equivalent to adding a least fixpoint operator to the nested algebra without powerset and is also equivalent to adding some programming constructs to the nested algebra without powerset (e.g., looping) [Gyss88]. The addition of Π also makes exactly one of the ν and – operators redundant (but not both) [Gyss88].

Null values are not supported in this algebra (but ∅ is), so we encounter a problem similar to that described above for the VERSO algebra when we wish to unnest empty relations. The algebra is also value-based. Our two examples can be expressed as follows in the Powerset Algebra:

1) Find the names of all publishers of *Moby Dick* who have an office in the state of New York:

$$\pi_{\text{publ\_name}}(\sigma_{\text{name = publ\_name}}((\sigma_{\text{state = "NY"}}(\mu_{\{\text{city, state, phones}\}}(\text{Publishers})))$$
$$\times(\mu_{\{\text{publ\_name, num\_pgs, date}\}}(\sigma_{\text{title = "Moby Dick"}}(\text{Books})))))$$

This query is more easily deciphered when presented as in Figure 2.16.

2) Retrieve the title and number of pages of every book by Herman Melville:

$$\pi_{\text{title, num\_pgs}}(\sigma_{\text{author = "Herman Melville"}}(\mu_{\{\text{publ\_name, num\_pgs, date}\}}(\text{Books})))$$

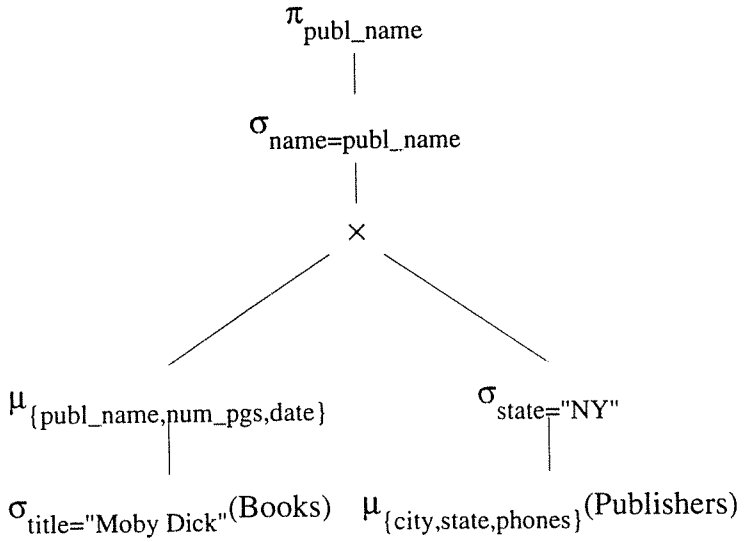

Figure 2.16: Tree representation of Query 1 (Powerset algebra)

Since the Powerset algebra is essentially an extension of the Vanderbilt algebra, it is not surprising that the queries are virtually identical. Of course, these examples do not highlight the salient feature of this algebra, namely the Π operator. An example of a query in which this is necessary is the computation of the transitive closure of a general binary relation. Such a general transitive closure expression can not be formed using the standard NF^2 algebra.

2.3. Other Database Algebras

This section describes algebras for advanced data models other than the nested relational model. Naturally, the same set of example queries as used in the previous section is not adequate to illustrate these algebras, nor is any one set of data definitions or queries. However, we attempt to use similar examples to those of the previous section to facilitate comparisons. Algebras described in detail here include algebras for relational aggregates, summary tables, images, complex objects, object-oriented systems, and office documents. In the final subsection we briefly describe some other algebras encountered in the literature.

2.3.1. An Algebra for Aggregates

In [Klug82] an algebra is developed which extends the relational algebra to include aggregate computations. This algebra is intended to be used by any relational system which might need it, and was also part of a system called ABE (Aggregate-By-Example, an extension of IBM's Query-By-Example, or QBE). The set of objects over which this algebra operates is the set of 1NF relations. The operators are thus identical to the relational algebra operators, with the addition of an operator to handle aggregates, which will be described below. For completeness, we present all of the operators:

1)-4) Projection, cross product, union, and difference are the standard relational operators. The syntax for projection is e[X], where e is some algebraic expression and X is the list of desired attributes.

5) Restriction: This is used in place of selection, and results in an equivalent algebra. It is written as e[X Θ Y](I), where e is an algebraic expression and I is an instance of a relation. It restricts the result to contain those tuples whose X component stands in relation Θ to its Y component, where Θ is either = or <. It differs from selection in that many definitions of selection allow constants to appear in the predicate as well as values from the actual tuples.

6) Aggregate formation: This is denoted by e<X, f>(I), where e is an algebraic expression, I is an instance of a relation, X is a set of attributes of the relation, and f is a function. The result of this operation is the set of tuples over the attributes ($X \cup \{F\}$), where F is an attribute indicating the result of applying the function f to the X attributes of each tuple of the original relation. More precisely, this operator partitions its input on the attributes X, applies the function f to each partition, and returns the X-value and associated f-value for each partition. As a limiting case, X can be empty, and we return simply a single value (for example, the average value of a single column, taken over the entire relation).

One motivation for this algebra is that aggregate computations can accept arguments which are multisets. This causes some of the usual algebraic identities to not hold, making query optimization more difficult. Essentially, the solution adopted was to define aggregate functions to operate over entire relations rather than on projections of these relations. This eliminates the need for the notion of duplicates. In [Klug82] a calculus is defined for this algebra and their equipollence is proved. Clearly, these languages are value-based. Null values are not allowed.

Since the aggregate algebra was not designed to handle NF^2 relations, instead of our two previous examples we provide two examples of relational aggregate usage and their corresponding representation in the aggregate algebra. Suppose we have a relation schema Emp = {Name, Dept, Salary}.

1) To obtain the average salary of all employees, we would pose the following algebraic query:

$$Emp<avg_3>$$

This will return a single 1-ary tuple, with a value which is the average salary of all the employees. The function avg_3 is defined to compute the average of the third column of its input. The empty list of partitioning attributes indicates that there is only one partition to be aggregated over, namely, the entire relation.

2) To obtain the average salary earned by the employees of each department, we would write the following:

$$Emp<2, avg_3>[2,4]$$

Notice that "2" here means the second column, i.e., the Dept attribute, and "4" refers to the result of the aggregate. This query would return a set of binary tuples, one for each department, containing the department name and the average salary of the employees in that department. The projection is performed after the aggregate and keeps only the second and fourth columns of the result of the aggregate.

2.3.2. The Summary Table Algebra

The algebra for summary tables [Ozso83, Ozso87] is basically an extension of Klug's algebra for aggregates (described in the previous section) to handle set-valued attributes. The domain of interest is thus relations that have both scalar fields and fields that are sets of scalars. Note that this is different from the NF^2 models, in which attributes may be sets of tuples but not sets of scalars. Clearly, this model involves no nesting of sets or relations, as set-valued attributes can not contain anything but scalar elements. The operators are defined as follows:

1)-4) Projection, Cartesian product, union, and difference are defined exactly as in the algebra for aggregates, both syntactically and semantically.

5) Restriction: This is also defined as in Klug's algebra, but here Θ can be $=$ or $<$ for scalars, $\equiv$ or $\subset$ for sets, or $\in$ for set membership tests.

6) Aggregate formation: This is defined exactly as in the algebra for aggregates.

7) Unpack (U): This is very similar to an unnesting. For each tuple of R, $U_X(R)$ creates a set of tuples, one for each value appearing in the X attribute of the original tuple. Each such tuple is formed by taking the original tuple and replacing its X value with one of the values present in the X value. These sets of tuples are then unioned to form the result. If X is a scalar attribute, the set from which the new value to be associated with each tuple is drawn is a singleton, and the result is simply R.

8) Pack (P): The effect of $P_X(R)$ is exactly that of $\nu_X(R)$ if X is a scalar attribute. If X is a set-valued attribute, then the sets corresponding to a particular set of values on the non-X attributes are unioned to form the new set. Thus new levels are not created when X is already a set-valued attribute. This is the difference between P and ν.

Three more non-fundamental operators are also defined in [Ozso87]. The Θ-join and selection are defined as a cross-product followed by a restriction. A join is defined only if the join attributes are Θ-compatible; thus either both attributes are scalar or both are set-valued or Θ is $\in$. An operation called aggregation-by-template, which can be defined using aggregate formation (and vice-versa) is introduced. Aggregation-by-template differs from aggregate formation in that aggregate formation is based on a strict partitioning of tuples (i.e., no tuple can appear in more than one partition). Aggregation-by-template is based on "grouping", and here this has the result that a tuple may appear in more than one group. The elements of another relation direct (i.e. form a template for) this grouping; we

do not go into the details here.

An equivalent calculus has been defined for the summary table algebra [Ozso87]. Like the algebra, it is an extension of the calculus defined in [Klug82]. It should also be noted that the aggregates operate only on a single scalar column and that the result of applying an aggregate to an empty set is null, which here means "does not exist". When an aggregate itself encounters a null value, it is ignored. Clearly this algebra is value-based, just like Klug's aggregate algebra.

There are no formal results regarding the expressive power of this algebra, but it is certainly at least as powerful as that presented in the previous section. The algebra has been implemented as the basis of two statistical database languages, STL and STBE. STL (Summary Table Language) is based directly on this algebra, while STBE (Summary Table By Example) is based on a modified version of the calculus equivalent of the summary table algebra.

Once again, we are prevented from expressing our example queries due to the nature of the objects operated on by the algebra. However, we can give two other examples of the summary table algebra in action. Consider the schema Emp = {Name, Dept, Salary} as described in the last section:

1) To obtain the average salary earned by the employees of each department, we would write the following:

$$Emp<2, avg_3>[2,4]$$

This is exactly example (2) of the previous section.

2) Suppose an employee could be associated with more than one department, and assume that salary is functionally dependent on name. To put these department names into a set-valued attribute, we would use the P operator as follows:

$$P_{Dept}(Emp)$$

For each (Name, Salary) pair we will now have a set of Dept attributes.

2.3.3. The LDM Algebra

The Logical Data Model (LDM [Kupe85]) is an extended complex object model which uses surrogates to refer to every object in the database (and in that sense it is an object-oriented model -- everything is an object). Objects are built recursively from scalars, sets, tuples, and the union type constructor. Every object in a set, tuple or union is denoted by an "l-value" (unique surrogate identifier). The union constructor supports constructs for generalization

by stipulating that the domain of a union node in an LDM schema graph is the union of the domains of its children in the schema graph. Cycles are allowed in the schema graph since the object-based nature of the model prevents infinite data from resulting. One interesting thing to note is that all schema graph nodes correspond to collections of surrogates (including tuple nodes, which add an ordering property to their children). Unlike the schema graphs of some other models, then, a tuple node in an LDM schema graph represents a collection of tuples rather than a single tuple, a set node represents a collection of sets, etc. Even leaf nodes of the schema graph can contain more than one surrogate.

The following operators are allowed on objects conforming to an LDM schema (we present the algebra as defined in [Kupe85]):

1) Copying (□): This operator takes either a scalar schema graph node or a simple constant (not yet placed into any actual LDM schema graph node) as input. In the former case, it creates a copy of the node and removes any duplicates (based on equality of the referenced objects, not on object identity). That is, a new surrogate (and a new copy of the value) will be generated for each distinct value referenced by the surrogates in the original node. In the latter case, □ creates a schema graph node of type "scalar" which contains the constant and a surrogate for it.

2) Powerset (O): O(v) for any schema node v is the powerset of all the surrogates present in node v. Each element of this powerset is assigned a new, unique surrogate, and each element of these elements is a copy of a surrogate appearing in v.

3) Cartesian product (⨯): This operation produces the (set-theoretic) Cartesian product of a finite number of input nodes. Each pair in the Cartesian product is represented by a new surrogate, and each element of each pair is a copy of a surrogate from one of the inputs.

4) Generalization (Δ): This operator creates a new schema node which contains a new surrogate for each element of all of its input nodes (all of these nodes must be distinct). The input nodes can be of different, arbitrary types. Each new surrogate refers to a copy of a surrogate from the input node. This operates on actual data, not on metadata.

5) Selection (σ_{iOj}): For a tuple node in a schema graph (i.e., a collection of surrogates referring to tuples), this operator creates a new tuple node with the same children as the input node. The new node contains new sur-

rogates (and copies of the original tuple's fields) for all tuples such that iOj holds for attributes i and j of the tuple. Here, O can test for set membership, equality, identity, projection (i.e., if j is a tuple-valued field, the operator will check if i is in one of its fields), and union type equality (i.e., if j is a union-valued field, is field i the object contained in j?). All of these comparisons (except equality) are based on object identity.

6) Containment selection ($\sigma_{in}(u,v)$): This operator works for any type of schema node (except scalar). If u is a child node of v in the schema graph, then the result is of the same type as u and contains new surrogates (and copies the referenced values) for every object in node u that is "contained" in node v. "Contained" here can mean three different things, depending on the type of v: If v is a tuple node, it means the surrogate from u appears somewhere in some tuple in v. If v is a union node, it means the surrogate from u appears as a value of one of the union-type surrogates in v. If v is a set node, it means that the surrogate from u appears as an element in some set in v.

7) Union ($\cup$): This operator performs the union of any number of nodes as long as all the input nodes are of the same type. The result node is also of this type, and contains new surrogates (and copies of the values) for all of the distinct values appearing in all of its input sets. Duplicates are eliminated based on value, not object identity.

8) Difference (–): For two nodes u and v of the same type, u – v is a node of that type containing newly created object identifiers (and copies of the referenced values) for values that are in u but not in v.

9) Projection (Π): This is nearly identical to the relational π operator. A new node is created and for each tuple in the input tuple node, a new tuple is created which contains only the requested fields. Note that duplicates are eliminated in the sense that tuples with identical surrogates in all fields will appear only once (but tuples with equal values in all fields may appear more than once).

Note that the LDM algebra's operations explicitly create copies of referenced data. In most other algebras such copying is implicit. It was made explicit here to conform to the model's formal definition, which leaves database schemas and instances unchanged during query processing (it merely adds to what is already there).

2.3.4. A Complex Object Algebra

In [Abit88a], an algebra and calculus are developed for a model whose objects are sets of values constructed from the set and tuple type constructors in an arbitrary fashion; this is a pure complex object model. The model is

also purely value-based, thus equality of two entities is determined by traversing their entire structures. Sets in the model are strictly homogeneous, and cyclic schemas are not allowed. The following operators are provided for complex objects:

1)-3) Union ($\cup$), Intersection ($\cap$), and Difference ($-$): Each of these takes two inputs of the same set type and produces a result of that type. They are defined exactly as in set theory. As in set theory, $\cap$ is redundant.

4) Cross product ($cross_A$): This is an n-ary operator which forms a set of n-tuples from n inputs, each of which must be a set. The value is the cross product of the input sets. The subscript indicates the name of the new tuple type contained in the result set.

5) Rename ($rename_{A \rightarrow B}$): This renames all occurrences of type A in the input expression to type B. No values are changed.

6) Powerset ($powerset_A$): This returns the set of all subsets of its input, which must be a set. The subscript indicates the name of the type contained in the result set.

7) Set-collapse (set-collapse): Given a set of sets, the member sets are all unioned together to produce a single set.

8) Replace ($\rho<G>(R)$): This operation iterates through a set of objects and performs the transformations indicated by G. G is a "replace specification". G is built recursively as follows: base specifications are constants, names of types appearing in the input type, and input parameters (each specification G can have input parameters which are algebraic expressions of a specified type; these are distinct from the input to the "replace" operator). Recursive specifications are built up from base specifications using the tuple and set constructors (i.e., a tuple or set of specifications is also a specification). Conditional and applicative specifications are also used recursively to build other specifications. A conditional specification can use the comparators "=" and "$\in$", and returns an object as its result only if the predicate is true. It acts like a relational selection. Finally, an applicative specification applies one of the other algebraic operators to each element of the input set (each element may, of course, have had other specifications applied to it prior to being given to the algebraic operator).

Note that all types are named (e.g., see (4) and (6) above), even those that represent the types of set elements (types of tuple fields are of course named to distinguish them from the other fields). This is simply to facilitate the

expression of queries on nested sets; other mechanisms could be used to achieve the same effect. An interesting theoretical result was obtained by removing the powerset operator from the algebra. The resulting algebra was proved equipollent to a version of the calculus which does not have the capability of specifying a formula which constructs all subsets of a given set. It was also proved that adding arbitrary functions and predicates to both the algebra and calculus does not affect their equivalence [Abit88a]. The algebra can easily simulate the nest and unnest operators of the NF^2 algebras, although they are not primitives in the complex object algebra.

2.3.5. An Algebra for Office Forms

In [Guti89] an algebra for structured office documents is presented. The model is essentially a nested relational model with sets replaced by sequences (ordered lists). The types of the algebraic objects are numeric, boolean, text, and complex, where "complex" indicates nested sequences of tuples. The algebra has 34 operators and 11 predicates. Mathematical operations, a conditional statement, the five standard aggregate functions, and the logical connectives are all operators of the algebra. This algebra was not designed with query optimization in mind. Three types of null values are supported, one of which is interpreted as "no information" and corresponds to the lack of the CWA (closed-world assumption) in this model. Due to the presence of nulls, two forms of equality are supported. Symbolic equality compares two symbols without interpreting them. Semantic equality takes into account the meaning of nulls and returns an appropriate answer.

The following operators are provided:

1)-4) Mathematical operators (+, –, *, /): These have the obvious definitions and operate on numeric scalars only.

5)-7) Logical connectives (**and, or, not**): These are also obvious, and operate only on scalars of type boolean.

8)-12)

Aggregates (**count, avg, sum, min, max**): These operate on sequences of tuples and return a scalar numeric value. **Count** counts the number of elements in a sequence. The other four perform the appropriate computations on a given (numerical scalar) field of a sequence.

13) Conditional (**if <boolean expr> then <expr1> else <expr2> fi**): The first expression is any one which yields a boolean result. The next two algebraic expressions must be of the same type (boolean, numeric, text, or complex). The result is either <expr1> or <expr2>, depending on the result of <boolean expr>.

14)-24)

Predicates (**isempty, exists, forall**, $=, \neq, <, >, \leq, \geq, \subseteq, \sqsubseteq$): Each of these returns a boolean after performing the proper comparison. They all perform standard comparisons except for $\sqsubseteq$, which tests for one sequence being a subsequence of another. For two sequences A and B, A $\sqsubseteq$ B iff there exist two sequences X and Y (possibly empty) such that B = XAY.

25) Union ($\cup$): This is done by concatenating two input sequences of the same type (see the **concat** operator below) the removing duplicates using **rdup** (defined below also).

26) Intersection ($\cap$): For two input sequences of the same type, the result contains all elements that are in both. The ordering is that present in the first sequence.

27) Difference (−): For two input sequences of the same type, the result contains all elements that are in the first but not in the second. The ordering is that present in the first sequence.

28) Cartesian Product (×): For any two sequences, this operates like the relational × operator. The only difference is that an ordering is imposed in the NST algebra. For A = <a1, ..., an> and B = <b1, ..., bm>, A × B = <a1b1, a1b2, ..., a1bm, a2b1, ..., anbm>.

29) Sequence concatenation (**concat**): For any two sequences of the same type, the result is the first sequence followed by the second.

30) Pairing (**pair**): For any two sequences A = <a1, ..., an> and B = <b1, ..., bm>, this operator returns <a1b1, a2b2, ..., akbk>, where k = min(m,n).

31)-32)

Projection (π, ρ): π is exactly like the relational π (except that it preserves ordering and does not remove duplicates). ρ is similar but it specifies fields to leave out of the result rather than fields to include in the result.

33) Sorting (**ord**): This allows a sequence to be sorted with respect to an ordered list of scalar attributes. Ascending or descending order can be specified for each attribute.

34) Duplicate elimination (**rdup**): This operator removes all but the first occurrence of an element from any sequence.

35) Grouping (γ): This is similar in effect to the NF^2 nest operator, except that orderings are preserved in the newly nested sequences.

36) Distribution (δ): This operator takes 2 top-level attributes and makes them attributes of a second-level attribute. Its effect can be simulated by unnesting then renesting in the NF^2 algebra, the difference being that orderings are preserved with δ.

37) Unpacking (μ): This acts like an order-preserving version of the "set-collapse" operator of [Abit88a]. For a sequence of unary tuples, where the field is a sequence, it concatenates all the sequences.

38) Packing (ν): Given a sequence A, ν (A) is a sequence consisting of one unary tuple whose only field is A.

39)-41)

Subsequences (**head[k], tail[k], portion[k1,k2]**): These operators return the first k elements, last k elements, or elements k1 through k2 of a sequence.

42) Reversal (**reverse**): The input sequence is reversed.

43) Selection (σ): This is defined much as in the DASDBS algebra, where select and project expressions may be nested within the top-level selection clause. Here, however, any algebraic expression may appear there (i.e., the selection can be based on any predicate involving any part of the top-level structure).

44) Join ($|\times|$): This is defined as $\times$ followed by σ, just as in the relational model.

45) Looping (λ): This operator loops through a sequence, applying a set of algebraic expressions to each tuple. Each field in a result tuple is either unchanged or is operated on by one of several expressions parameterizing the λ operator. This operator can also add fields to the tuples if desired.

2.3.6. ENCORE/EQUAL

An algebra for the ENCORE/EQUAL object-oriented database system is presented in [Shaw90]. This is a purely object-based system. The model is a standard object-oriented one with the capacity to distinguish between sets of object identifiers and multisets of object identifiers, although the exact semantics of multisets are left unspecified. The operations available are as follows: The select operator is similar to the relational selection. There are "image" and "project" operators which retrieve, for each object in a set, the result of one ("image") or more ("project") functions defined on the objects of the set. These functions can be arbitrary methods, as in most

object-oriented models. There are also union, intersection, and difference operators, all based on object identity. The "flatten" operation is like the "set-collapse" operation of [Abit88a]: it turns a set of sets into a set. The NF^2 operators nest and unnest are also provided. An explicit duplicate elimination operator is available and there is an operator called "coalesce" to eliminate duplicates at a certain level of nesting within an object. That is, if the same attribute of two different tuples refers to objects which are equal (contain the same collection of object identifiers) but have different object identifiers themselves, one of the objects is destroyed and the reference to it is replaced with a reference to the other equal object.

As an example of the coalesce operator, consider the following relation R:

A	B	C
a1	b1	c1
a2	b1	c1
a1	b2	c2
a2	b2	c2

If we perform the operation Nest(R, BC) we get the following result:

A	(B	C)
a1	b1	c1
	b2	c2
a2	b1	c1
	b2	c2

The second components of each of these two tuples contain the same objects. But in an object-oriented system, this operation might result in the creation of two separate set objects which in this case just happen to contain the same elements. The coalesce operator would ensure that only one such set is created in this example, and the (BC) attribute of each result tuple would contain the object identifier of that set. This avoids the creation of multiple copies of the same nested set. Note that it has no use outside a pure object-oriented context.

The system maintains type extents (but allows subsets of these to exist), and queries are always over these collections of objects. Also, these top-level sets (and top-level multisets) contain only OIDs. Query results are collections of an existing type or of a tuple type constructed during the query. Some relational-like query optimization techniques are also presented in [Shaw90].

2.3.7. An Object-Oriented Database Algebra

[Osbo88] describes an object-oriented data model and algebra. Structurally, the model supports scalar values and the tuple and set type constructors, as well as inheritance among tuple types. One important extension, though, is that sets need not be homogeneous. Everything in a database is an object with its own unique identity, thus everything in a field of a tuple and every element in a set (homogeneous or not) is an object identifier. Unlike many object-oriented models, however, this one does not support the notion of methods associated with object types (classes). Tests for both identity and equality of two objects are allowed. Equality here means "deep equality"; i.e., an entire hierarchical structure is traversed, and all parts of it must be equal to the corresponding parts of the other.

The algebra supports the following operators:

1)-3) Union (`Union`), Intersection (`Intersect`), and Difference (`Subtract`): Each of these takes 2 sets of object identifiers and produces a new (set-valued) object. Each is defined exactly like its set-theoretic counterpart, and equality of object identifiers (i.e., identity of the objects referred to) is used to test the set elements for equality.

4) Combining two objects (`Combine`): This operation is defined on any pair of objects such that either i) both are tuples, ii) both are sets, or iii) one is a set and the other a tuple. In case (i), the result is a new tuple object with all the fields of each input tuple. In case (ii), the result is computed by invoking `Combine` on each element of the Cartesian product of the inputs. In case (iii), `Combine` invokes itself on every pair (*t*, *e*), where *t* is the input tuple and *e* is an element of the input set.

5) Partitioning an object ($\texttt{Partition}_{\texttt{attr\_list}}$): The input to this operator must be either a tuple or a set. If it is a tuple, the result is a new tuple containing only the attributes in attr_list (or null if the tuple does not contain all the attributes in attr_list). If the input is a set, the output is a new set containing the result of applying `Partition` to every tuple in the set which contains all of the attributes in attr_list. Thus it can perform relational projections.

6) Predicates ($\texttt{Choose}_{\texttt{Pred}}$): A predicate (Pred) consists of terms of the following form connected by $\wedge$ and $\neg$: obj IsNull, obj1 = obj2, obj1 $\equiv$ obj2 (identity test), and obj InClass C (where C is the name of some class). In the expression $\texttt{Choose}_{\texttt{Pred}}\texttt{(obj)}$, if `obj` is a set, the result is a new set containing (the identifiers of) all objects in the set for which `Pred` is true. Otherwise the result is `obj` if `Pred` applied to `obj` is true

and null if it is false.

Several algebraic transformation rules are given in [Osbo88]. An interesting consideration is whether one wants to allow transformations that preserve equality or only those that preserve identity as well. [Osbo88] also states that the expressive power of the algebra is somewhat limited, as it can not simulate the nest and unnest operators of the NF^2 algebras, nor can it do, for example, general transitive closures. However, some results in [Hull89b] suggest that allowing inhomogeneous sets may actually add a great deal of expressivity to the algebra.

2.3.8. The R^2 Algebra

This algebra [Houb87] is an extension of a nested relational algebra; it is not associated with a particular implementation or system. The goals of these extensions are to capture the notions of aggregates, computed (derived) attributes, and recursion, thus we describe it here rather than with the more "pure" nested relational algebras of Section 2.2. An orthogonal goal of R^2 is the issue of user interfaces: the operators are defined with a mouse-oriented interface in mind. This is especially suited to the two-dimensional nature of nested relations (e.g., graphical representation of schemas). The domain of discourse is the set of all NF^2 relations--no restrictions are imposed on the data. The following operators appear in the R^2 algebra:

1) Union (UNI): This is defined as a set-theoretic union, as in the relational model.

2) Difference (DIF): This is simply set-theoretic difference, also as in the relational model.

3) Join (JOI): This is basically a Cartesian product, the only difference being that top-level attributes (scalar or relation-valued) common to both relations being "joined" appear only once in the result.

4) Selection (SEL): SEL[S; f; L; n] is defined to be the set of all tuples over the schema S which satisfy the boolean function f; f is applied to the attributes in the attribute list L. The name of the result schema is n. The important thing to note about SEL is that f can be an arbitrary boolean function applied to any parts of the schema.

5) Projection (PRO): PRO[S; L; n] is defined to be the set of tuples over the list of attributes L from the schema S. The name of the result schema is n. PRO is similar to the VERSO projection in that only entire subtrees may be cut away from the result schema. That is, any attribute (from any level) appearing in L will carry along with it all of its successors and predecessors. The only difference is that in R^2, we are allowed to project scalar columns out of the result, avoiding excessive nesting.

6) Nest (NES): This is defined as in the DASDBS algebra.

7) Unnest (UNN): This is defined as in the DASDBS algebra.

8) Aggregation (AGG): AGG[S; f; L; R; n] is the application of an aggregate function f (any computable function with a scalar result) to a relation with schema S. The relation is restricted to the attributes of S appearing in L, and the name of the result attribute is R. This result attribute will be a new attribute one level up in the structure from the attributes in L, which must all be siblings and scalar and can not be attributes at the top level of a relation (some combination of NES, UNN, and AGG must be used for this special case). The name of the result schema is n.

9) Computation (COM): Here, an atomic value is computed for a single tuple based on values of other atomic attributes of the tuple, and the computed value becomes a new attribute of the tuple. COM[S; f; L; R; n] is the application of the arbitrary (computable) scalar-returning function f to the (scalar) attributes of schema S appearing in the list L. The name of the new attribute is R, and n is the name of the result schema.

The definition of a renaming operator was consciously omitted from the algebra due to its essential triviality. There is also a facility in R^2 for defining operations as sequences of algebraic operators which will be stored and can then be executed by name (known as canned programs or stored commands in other systems).

The ability to make use of arbitrary computable boolean functions, aggregates, and tuple computations sets this algebra apart from the other algebras studied thus far. No formal results exist describing the expressive power of this algebra, but it is certainly at least as powerful as the normal NF^2 algebra (e.g. DASDBS) and the summary table algebra. In addition, no formal calculus has been defined for the algebra. Whether or not an existing calculus will serve the purpose will be determined by formal equivalence results relating the R^2 algebra to the DASDBS and summary table algebras. This expressive power is a current area of research within the R^2 project, as is the use of stored commands to help express recursive queries. The ability to express recursion was one of the major goals of the project, but it is not clear that the algebra as defined here fulfills that goal directly. It does allow for the presence of $\varnothing$ as a value, but this can lead to the same problem as occurs with VERSO since nulls are not allowed (at least, they are not mentioned). Finally, we note that the R^2 algebra is value-based.

The two example queries can be expressed in the R^2 algebra as follows:

1) Find the names of all publishers of *Moby Dick* who have an office in the state of New York:

PRO[r1; publ_name; result] (SEL[r4; f1; name, publ_name; r1] (JOI[r2; r3; r4]

(SEL[Publisher; f2; locations; r2] (Publishers),

(UNN[r5; copies; r3] (SEL[Book; f3; title; r5] (Books))))

This query is more easily deciphered when presented as in Figure 2.17.

2) Retrieve the title and number of pages of every book by Herman Melville:

PRO[r1; title, num_pgs; result] (SEL[r2; f4; author; r1]

(UNN[Book; copies; r2] (Books)))

In the bracketed lists associated with each operator, recall that the first element (or first two, in the case of JOI) specify the input type(s) and the last specifies the output type of the operator. Here, f1 names a pre-existing function taking two inputs of types name and publ_name and returning a boolean value; in this case, the result of an equality test is returned. Similarly, f2 indicates a function that returns true if its argument has a tuple with state = "NY", and f3 returns true if the title passed to it is "Moby Dick". In the second example, f4 returns true if the author passed to it is "Herman Melville". The unrestricted nature of these functions allows us to express just one selection

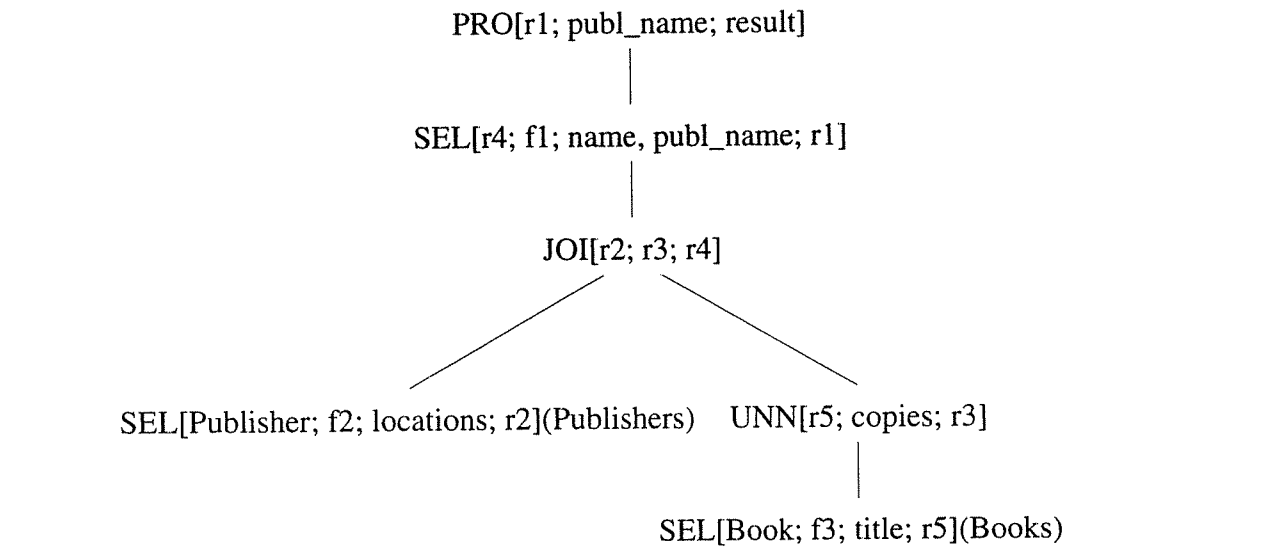

Figure 2.17: Tree representation of Query 1 (R^2)

predicate for f2 where normally we would express two nested selections (e.g. see the DASDBS version of this query).

2.3.9. Other Algebras

Here we briefly mention some other systems which propose or make use of a post-relational algebra. Most of the algebras described in this section are similar enough to one or more of the preceding algebras that it was felt that adding a more complete description of any of these would not really add to the discussion. In addition, some of them are less "post-relational" in the sense that they rely too heavily on existing relational systems and algebras. The first subsection describes algebras which are nearly relational or based mainly on the relational algebra. Algebras closely resembling the NF^2 or complex object algebras are presented in the next subsection.

2.3.9.1. Relational-Like Algebras

In [Clif85] an extension to the relational model is presented which allows for historical queries on a relation. Attributes are viewed as either unchanging, time-varying, or time-valued (i.e., their value is, e.g., a date). Three different forms of nulls are introduced to handle incomplete information about time. A time-varying attribute is really a set of (value, valid-time-interval) pairs. The five relational operators are augmented by two time-oriented operators: "time-slice" (τ), which retrieves the state of a relation as of a given time, and "when" (Ω), which returns a set of times during which the database was in a specified state. An alternate historical algebra is presented in the same paper; this algebra supports the aggregate formation operator as described in [Klug82, Ozso83] and the pack and unpack operators as described in [Ozso83]. For this algebra, the fundamental operators (in addition to the relational operators) are pack, unpack, and two operators which convert time-varying attributes into regular attributes and vice-versa.

An interesting algebra for the relational model with duplicates appears in [Daya82]. This algebra redefines the set-theoretic operations to be the corresponding multiset-theoretic operations and adds a duplicate elimination operator. It also includes the (redundant) multiset intersection operator.

In [Pare85] an algebra for Entity-Relationship [Chen76] databases is presented. Its operations include a relationship join (similar to an outer join) to build results based on stored relationships, Cartesian product (defined as in the VERSO algebra), selection, duplicate elimination, renaming, union, intersection, and difference. Existing relationships are "inherited" by the results of queries in the obvious manner. A similar algebra is defined for the

EXTREM data model [Scho87b]; the operations differ only in some details. EXTREM is a semantic data model implemented on top of a flat relational system and using some of the constructs of the IFO semantic data model [Abit88b]. Chen [Chen84] also proposed an algebra for the E/R model. This algebra operates on data which is restricted to allow only binary directional relationships. The operators provided for this model are as follows: Select, union, difference, and intersection can be applied to sets of values or to entity sets. Cartesian product may be applied to value sets only. There are also operators to compose, decompose, invert, and create relationships. The last operation has the Cartesian product of two entity sets as a special case.

An algebra has also been developed for FAD [Banc87], but in FAD (as in IRIS [Fish87]) the real optimization work is performed by a relational optimizer -- FAD expressions are first translated to flat relational algebra. FAD was used as a backend for the LDL system developed at MCC [Tsur86]. The filter operation, for example, is translated into a series of binary filters which are then processed using relational joins. FAD is an object-based system. The operations include the standard set functions, user-defined ADT or FAD operations, a filter operation which applies a function to all elements of a cross-product of sets, a grouping operation, and a pump operation designed to facilitate parallelization of aggregate computations.

A least fixpoint (LFP) operator is defined in [Aho79] and some algebraic transformations using it are also given. Whether or not a relational algebra with a powerset operator would be equivalent to this algebra is an interesting question (recall that this equivalence holds for the NF^2 algebra). The ALPHA algebra [Agra87] also extends the traditional relational algebra to support recursion, but in this case it is extended with Klug's support for aggregates [Klug82] and one more operator, known as α. The α operator provides support for a limited form of recursion and is integrated into the algebra in such a way as to allow traditional algebraic query optimization to take place. The form of recursion supported is one that traverses a directed graph from a set of initial nodes.

The ASTRID algebra [Gray84] is an extension of the relational algebra which provides the following operations: union, difference, Cartesian product, intersection, join, set division, selection, projection, grouping, renaming, and tuple extension. Only the last three operators extend the original relational algebra. Grouping is similar to the EXTRA/EXCESS GRP operator (see [Vand91] and Chapter 5). The tuple extension operator adds a field (which is functionally dependent on the key) to a tuple. Renaming changes column names.

Codd's RM/T (Relational Model/Tasmania) model [Codd79] is a fairly complicated object-oriented extension to the relational model with both object- and value-based semantics. It also supports many of the concepts

described in the semantic database literature. In addition to the standard relational operators extended to allow null values, it provides operations to apply functions to sets of relations (such sets are algebraic objects just like relations are), operators for forming and partitioning sets of relations, and operators which retrieve information about the meta-data--i.e., the hierarchical structures of the relations in the system. The operators involving the meta-data allow graph traversal and transitive closure. It is interesting to note that in RM/T everything is stored in a relation--i.e., an entire database as described in [Codd79] is a set of relations, as in [Codd70]. There is no notion of "complex objects"; rather, the concepts associated with such objects are modeled using additional relations.

2.3.9.2. NF^2-Like and Other Algebras

The PDM algebra was developed for use with the PROBE project at CCA [Mano86, Daya87]. The model is derived mainly from DAPLEX [Ship81], and it is object-based. The operations of the algebra include those of the relational algebra (projection, Cartesian product, union, difference, intersection, and selection) plus several others: outer versions of the union, difference, and intersection operators (PDM supports two types of nulls, "don't know" and "don't care"). There is also an operator called "apply_append" which applies a function to every object in a set and appends the result for each object to that object. The operation "createobj" creates a new object with its own identity, and the "unique" operation removes duplicate identifiers based on the values of certain columns of the objects pointed to by the identifiers.

In [Gyss89] an algebra was introduced for an intriguing model which represents all hierarchical structures as formal context-free grammars. This seems to be a very powerful formalism, but its utility has yet to be determined. Its operators are renaming, insertion (replace a terminal with a non-terminal), deletion (inverse of insertion), copying, and permutation. No expressivity or complexity results exist for this model yet.

The MAD (Molecule-Atom Data Model) algebra described in [Mits89] has some interesting features. The model consists of "atoms", corresponding to real-world entities, and "molecules", corresponding to collections of atoms related by "links". Links between atoms are preserved (to the extent possible) during query processing. The algebra is an extension to the relational algebra, with the same operators but each extended to operate on molecules.

The ALGRES system [Ceri87] is intended to be used for rapid specification and prototyping of complex database systems. It supports the NF^2 model with several additions: tuple-valued attributes, a closure operator to enable least fixpoint computations, aggregates, and an embedding in a computationally complete logic programming

language. The operations supported are the five relational operators (suitably extended), nest and unnest, closure, aggregates, and type coercion operators. Aggregation operates much as in the summary table algebra. Ordered sets and multisets are also supported. Queries are actually executed over an INFORMIX relational database--that is, an ALGRES query is translated to a flat algebra expression before execution. ALGRES is a value-based system.

SIRIUS is a distributed DBMS developed at INRIA [Scho87b]. It is designed to be a somewhat extended relational (value-based) system which uses an NF^2 model much in the same way that DASDBS does (i.e., as a vehicle upon which more sophisticated models may be developed). In addition to the DASDBS-like NF^2 model, SIRIUS provides user-defined ADTs, derived attributes (as seen in R^2), null values, and full integrity constraint specification capabilities.

Another extended algebra has been developed in the context of the ANNEX project at the University of Nice [Scho87b]. The algebra for their "B-Relational" data model includes the standard NF^2 extensions as found in the DASDBS and AIM algebras as well as support for semantic concepts such as generalization and association.

In [Aris83], Arisawa et al develop update algorithms for nested relations, but along the way define some very primitive operations on tuples (primitive in the sense that they may be used to define more complex operations such as nest and unnest as well as to describe subtler changes to a relation). The composition operator is used to produce a new tuple from two tuples which have the same attributes and the same values on all attributes except one. The new tuple is the same as the old ones but the differing attribute, which must be set-valued, becomes the union of the two sets from the original tuples. This operation can be used in a straightforward way to simulate the nest operation. Similarly, decomposition creates a copy of a tuple by extracting exactly one value from the original tuple and making a new tuple which is a copy of the original except that it has just the one value in the set-valued attribute specified by the operation. This value is removed from the proper attribute of the original tuple. In addition, [Aris83] defines a canonical form for nested relations, which is the nesting of a 1NF relation along each of its attributes in any order. Only one level of nesting is considered.

In [Tans89] an algebra is described which extends the NF^2 algebra to handle time-varying data. The entities in this temporal model are called temporal atoms, which are ordered pairs consisting of a value and a set of times during which that value is valid. The seven standard NF^2 operators are extended with a new one, TRANSFER-TIME. This operator takes two attributes from a tuple and replaces the set of valid times for one attribute with that of the other attribute. In other words, attributes (not tuples) have timestamps. There are also operators to build up and

destroy individual temporal atoms.

The algebra of the Revelation system [Grae88, Dani91] uses extended relational operators and requests objects to "reveal" an expression (in this algebra) for execution (the request may be refused). The algebra of [Kort88] is designed for a similar system and contains several implementations of the join operator on nested relations; the algebra used there is similar to the Vanderbilt algebra. Essentially, the approach is to treat an object-oriented query as a nested relational query, if possible, and to recast the method invocation as a nested algebra expression. The join algorithms proposed all proceed by first unnesting the relations involved. Some restrictions include the requirement that objects be instances of exactly one class and that recursive structures are not permitted.

[Stra90] presents an algebra (and equivalent calculus) for an object-oriented system. The operators provided are union, difference, selection, and mapping. The "map" operator performs a sequence of class method applications to every element in a set. The selection operator returns the members of a set which satisfy a predicate. Each of these operators has an additional argument which is a list of sets of objects. This list is used to parameterize the method invocations (in the case of the "map" operator) or selection formula (in the case of "select"). The semantics in the case of the "map" operator are that each member of the set has the sequence of methods applied to it once for every combination of elements from the list of sets of objects. For the "select" operator, the semantics are that a set member becomes part of the result if it satisfies the predicate for every combination of elements from the list of sets of objects. These two operators can be used, for example, to simulate a relational join.

An algebra for nested relations which defines each of the seven standard operators recursively is described in [Colb89]. The recursive nature of the DASDBS select and project operators is incorporated here, as is the recursive nature of the SQL/NF set operations. Recursive join definitions are also included. The point of all this recursiveness is to eliminate the use of nest and unnest except when they are truly needed; this leads to queries which are more easily optimized. The algebra does not assume PNF.

There are two more algebras which resemble the NF^2 algebra. The first is NTD (Nested Table Data model) [Scho87b], which is an algebra for office forms; it has the same goals and domain of application as the algebra of [Guti89] discussed above. The NF^{2D} model (also described in [Scho87b]) is essentially the NF^2 model with the added ability to provide for schema evolution. With this in mind, the description in [Scho87b] provides an expansion operator to alter the structure of attributes in the schema. This allows an attribute to change its structure dynamically without changing its meaning. Attributes for which this property is desired must be singled out as

dynamic so that the expansion operator can make use of information about the possible domains of that attribute at run-time.

Finally, we mention an algebra defined to operate on the domain of images [Ritt87]. This algebra demonstrates the utility of the algebraic paradigm outside of the standard data models found in the usual DBMS literature, giving further evidence of the universality of the approach. In this algebra, the objects to operated upon are real-valued images (elements of R^n for some n). There are also objects known as "templates", which are capable of defining very general image transformations. The operators available include the usual matrix and vector arithmetic, operators to apply templates to images, and operations between templates, in addition to the normal set-valued operations. This algebra is intended to be used to optimize image processing systems.

This concludes our description of post-relational algebras. The next section gives a brief summary of the survey.

2.4. Summary

We conclude the survey by summarizing the salient features of the algebras described above. Figure 2.18 summarizes the NF^2 algebras presented in Section 2.2. For NF^2 algebras the interesting features include any restrictions (e.g. normal forms) required of a database in order for the operations of the algebra to be applicable; whether or not the algebra allows for value-based or object-based operations or both; and an unannotated list of the algebraic operators provided.

In the preceding sections we have seen many different ways of defining an algebra. What is apparent is the central nature of the relational algebra--virtually all of the algebras described above are based on this algebra in some way. The most important common theme here, clearly, is the inadequacy of the relational model and its algebra. Specifically, these algebraic extensions imply that the relational model falls short in the areas of modelling constructs (not rich enough), expressive power (cannot do, e.g., a general transitive closure), computational power (absence of aggregates, etc.), historical databases (i.e., we cannot express the query "what was true at time x?"), and object-based notions (which can help alleviate some of the anomalies encountered in 1NF relations).

A main purpose of this survey has been to examine the operators defined for post-relational algebras and to identify the ways in which they extend or supplant the relational operators. The most popular sets of primitives seem to be based on the relational algebra, and fairly standard sets of operators seem to be catching hold in the area

	Restrictions	Objects/Values	Operators
DASDBS	none	V	$\times, \cup, -, \sigma, \pi, \nu, \mu$
AIM	none	both	$\times, \cup, -, \sigma, \pi, \nu, \mu, \chi, \rho$
Vanderbilt	none	V	$\times, \cup, -, \sigma, \pi$, NEST, UNNEST, UNNEST*
SQL/NF	PNF	V	$\times, \cup^e, -^e, \cap^e, \sigma, \pi^e, \|\times\|^e, \nu, \mu$
VERSO	PNF and at least one scalar column	V	extension, π, selection, restrict, rename, $\cup$, $\cap$, $-$, join, $\times$, restruct
NRDM	PNF	V	$\cup^e, -^e, \sigma^e, \pi^e, \|\times\|, \nu, \mu$
Waterloo	PNF	V	$\cup, -, \cap, \sigma, \pi, \times, \nu, \mu, \delta$, subrelation constructor
Powerset	none	V	$\cup, -, \sigma, \pi, \times, \nu, \mu, \rho, \Pi$

Figure 2.18: Nested relational algebras

of NF^2 relations. Another interesting point of comparison is the definition of NF^2 join operators. Sometimes join is defined as in the relational model (i.e., as $\times$ followed by σ) and sometimes as an $\cap$-based operator.

But for data more complex than NF^2 relations, a clearly favorite set of primitives has not yet emerged (indeed, only a handful of algebras for complex objects exist). More work remains to be done in this area. In particular, the amenability of such monolithic operators as the VERSO "restruct" and the "replace" of [Abit88a] to standard algebraic optimization techniques needs to be investigated. That is, a small set of extremely powerful operators may allow too few alternatives for query processing. Conversely, a large set of extremely primitive operators may allow too many alternatives (in the sense that optimization may become too slow).

There are several dimensions along which we can compare algebras in general: 1) Is the algebra set-oriented? This makes many proofs much easier (e.g. [Abit88a]) but may lead to less natural data representations. 2) What drove the design of the algebra? Most operators of VERSO [Abit86], for example, were designed as recursive filters for the VERSO virtual machine, while other were driven by the need to correspond to a calculus. 3) Which of its operators are recursive? For example, the NF^2 algebra of [Colb89] has 5 recursive operators, but other NF^2 alge-

bras have 0, 1, or 2, etc. The more recursive operators, the less need for nesting and unnesting. Other models may not be so amenable to recursive operators, as they depend on a known, consistent object structure, which is not present in complex object models. 4) Which of its operators are second-order? A true second-order operator will allow its function parameter to be optimized in the same way as its data input is optimized, removing the need for special sublanguages for operators as required in [Abit88a].

CHAPTER 3

SUMMARY OF MOTIVATIONS AND RELATED WORK

This chapter discusses work that is more directly related to the remainder of the thesis. In Section 3.1 we describe other work that was relevant to the design of EXTRA and EXCESS. Section 3.2 contains a summary of the survey of algebras in Chapter 2 and highlights the deficiencies of these algebras that led to the design of the EXCESS algebra and to certain constructs of AQUA.

3.1. Object-Oriented Models and Languages

Here we discuss the relationship of various features of EXTRA and EXCESS to other systems and languages. EXTRA and EXCESS are described fully in Chapter 4.

The EXTRA data model and EXCESS query language designs represent a synthesis and extension of ideas drawn from a number of other data models. The data structuring facilities of the EXTRA data model are probably closest to those of GemStone [Maie86c], as GemStone provides both tuple and array constructors, and data is organized into user-maintained extents rather than system-maintained type extents. EXTRA's **ref** notion was based on GEM reference attributes [Zani83], and **own ref** is closely related to weak entities in the E-R model [Chen76] and composite objects in ORION [Bane87]. EXTRA also goes beyond these systems in certain ways, however. In particular, EXTRA's mix of **own**, **ref**, and **own ref** attributes yields a relatively unique mix of (structural) object- and value-orientation. Neither GemStone nor Orion have support for **own** attributes (except perhaps in implementing their small atomic types). And while GEM had both **own** and **ref** attributes, its type system was less rich and sets of references were not permitted. Since the original development of EXTRA/EXCESS, O_2 [Banc88] has adopted **own** and **ref** attributes.

The EXCESS query language is related to those of DAPLEX [Ship81], GEM [Zani83], NF^2 systems [Dada86, Sche86], and POSTGRES [Rowe87]. Implicit joins were taken directly from GEM, and originated in DAPLEX. The EXCESS treatment of queries over nested sets is similar in flavor to that of NF^2 query languages, although the path syntax for handling deeply nested queries was influenced by DAPLEX and the early STDM paper [Cope84]. Our handling of range variables was heavily influenced by that of POSTGRES [Rowe87]. In addition, EXCESS

goes beyond its predecessors in several respects. Our mix of object- and value-oriented semantics, again, was unique among query languages, until the introduction of certain constructs to the O_2 [Banc88] query language. Also, EXCESS provides a cleaner treatment of arrays than we have seen elsewhere. The only point for comparison here is POSTQUEL, which only operates on one-dimensional arrays of base types.

Finally, ADT and access method extensibility in EXTRA/EXCESS were heavily influenced by the work of Stonebraker [Ong84, Ston86] and the resulting extension facilities in POSTGRES [Ston87b]. Our work here differs mostly in minor respects. Because ADTs in our system are written E, the system's internal language, adding ADTs is perhaps simpler here. Another difference is that we view ADT operators as synonyms for function calls rather than something to be handled differently for query optimization purposes. Lastly, our approach to user-defined set functions is more general than the corresponding POSTGRES approach to user-defined aggregates. With respect to schema types, our support for EXCESS functions is similar to the functions of DAPLEX [Ship81] and IRIS [Fish87], and also to the parameterized procedures of POSTGRES [Ston87a]. Our approach to user-defined procedures is rooted in the stored commands of the IDM database machine [IDM500], as is our approach to encapsulation through authorization, but EXCESS procedures are a much more general mechanism. The recently proposed procedures of SQL3 are similar to those provided by EXCESS.

3.2. Motivation for the EXCESS and AQUA Algebras

This section discusses specific shortcomings of the algebras of the previous chapter. Based on the information in Chapter 2, and on the goals of the EXTRA/EXCESS and AQUA projects, we outline the issues that the EXCESS and AQUA algebras are intended to address.

In general, we can identify several things that are missing from algebras designed for complex, advanced data models similar to EXTRA and EXCESS. We list them now, then describe them more fully below: transformation rules; support for grouping, inheritance, multisets, and arrays; orthogonality of structures and operators; equipollence proofs; simple operators; support for different forms of equality; support for abstraction; and operators and optimizations for queries over trees, lists, and graphs.

3.2.1. The EXCESS Algebra

Several 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 single tuple is not a singleton set of tuples). This allows more natural algebraic representations of some entities. This many-sortedness allows us easily to 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 new, as are most of the other multiset transformations we provide. ENCORE/EQUAL [Shaw90] 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 (value-based) 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 and all of the array transformation rules are new. There is an array looping operator similar to SET_APPLY. The ARR_EXTRACT operator extracts a single element from an array, and the result is not an array containing the element but simply the element itself. There is also an array creation operator and an operator to collapse an array of arrays. The notion of sequences supported in 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 object identifiers (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). 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 [Shaw90, 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 we 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 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.

Here we briefly review the the contents of the survey in Chapter 2, by tracing 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, Jaes82b]. 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, Hull87] 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, Shaw90]; 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/EXCESS, such as references and grouping.

3.2.2. The AQUA Algebra

Here we address only the features of AQUA that are most relevant to the contributions of this thesis. One of the primary goals of AQUA [Leun93] is to provide a model general enough to simulate the constructs of any object-oriented data model (and most value-oriented models), no matter what choices it makes with respect to certain features (bulk types, encapsulation, identity versus value, notions of equality, inheritance, and operations). Other models have claimed similar goals, but not necessarily in all these areas at once, and our mechanisms for achieving these goals differ substantially from those of our predecessors. Many of the specific constructs of AQUA were inspired by or drawn from the EXTRA/EXCESS system [Vand91], ENCORE/EQUAL [Shaw90], and Revelation [Vanc92].

In attempting to support both values and objects, some systems (e.g. EXCESS [Vand91]) choose to support only values in the type system and to model objects by using explicit identifiers. Other systems (e.g. Smalltalk and ORION [Gold83, Bane87]) choose to support only objects in the type system and to model values as a special case of objects. IQL [Abit89] defines two separate languages, one enforcing object identity and one not supporting it at all. AQUA characterizes the distinction between ''objects'' and ''values'' as the difference between entities (objects) with mutable and immutable semantics; this provides a much cleaner formalism, and was partially inspired by systems such as Larch [Gutt85]. By cleanly separating the notions of type (a syntactic concept) and semantics we provide a model that treats both values and objects as first-class citizens and has a simpler type system. We are not aware of another model that takes this approach, nor of one that takes the clearly-separated, 3-level view of an object that we do (type, semantics, and implementation). Buneman and Ohori exhibit a similar philosophy, though, in their distinction between a *kind* and a *type* [Bune91]. Unlike ILOG [Hull90] and others, we avoid explicit identifiers in the model, viewing them as an implementation concern, and reflecting the distinctions between objects and values by using varying semantics.

It has been pointed out by Atkinson et al [Atki91] that in object-oriented systems, a type may supply its own method for testing equality. This capability, however, introduces problems such as what the meaning is of operators like set union that depend on equality for their own semantics. The AQUA approach to enforcing a notion of equality among the elements of a set is without precedent in that it avoids any built-in notion of equality other than object identity, but still allows the (non-deterministic) creation of sets in which the uniqueness of the members is determined by an arbitrary equivalence relation. Many models (e.g. MDM [Rich91]) do not have this flexibility. The AQUA approach to this problem is closely related to duplicate elimination and non-determinism, as shall be seen.

Duplicate elimination for both sets and multisets, as defined in AQUA, is original. AQUA's **dup_elim** can be thought of as a generalization of other duplicate elimination operators (e.g. that of ENCORE/EQUAL [Shaw90]). A set-theoretic **choose** operator appears in the algebras of Osborne and MDM [Osbo88, Rich92]. Non-determinism is also present in [Abit90], which describes a **witness** operator which operates in a logical (rather than an algebraic) setting and creates a set of possible interpretations of a formula, resulting in non-determinism.

Most ''pure'' object-oriented models (such as [Gold83, Maie86c] and others) provide and enforce encapsulation of data types. In AQUA the notion of type is more general: not everything is forced to be of an encapsulated, abstract data type whose only interface is that provided by the definer of the type. But AQUA does support such types, and does so using the ''abstraction'' type constructor, allowing any database object, encapsulated or not, to be described using a single uniform type system. This is similar to the ADT concept provided by POSTGRES [Rowe87] but much more general in the sense that any type definable in the AQUA type system can be abstracted into a true encapsulated type, and an abstraction in AQUA is a first-class citizen of the type system -- the abstraction constructor has the same status as any other constructor. The distinction is that an object of an abstraction type has only the user-defined methods as an interface, while an object of (for example) a set type has an interface consisting of union, difference, etc. This is similar to POSTGRES's notion of user-defined POSTQUEL functions and the functions and procedures of EXCESS [Care88b], but in those systems, the ability to define functions allows one to add operations to an existing, non-encapsulated type (i.e., encapsulation is not enforced in those systems but it is in AQUA). The use of a type constructor to represent abstraction enables all objects in an AQUA database to exist in one seamless type system. Our approach is similar to that of [Alba91], but in that model not everything is an object, so their equivalent to our abstraction constructor must enforce many more of the facets of ''objectness'' than must ours.

Another goal of AQUA is to provide algebraic support for ordered data types such as lists, trees, and graphs. In Chapter 6 of this thesis we are concerned mainly with trees. For an introduction to AQUA's support for other ordered data types, see [Subr93]. Much of the previous work with ordering deals with order as in sequences or arrays; very little work has been done on trees or graphs. [Beer90] proposes an object-oriented query processing paradigm where the objects are built of primitive objects, an explicit object identity type constructor, and bulk type constructors. Then operations and optimizations are presented which apply to any bulk type constructor definable in their paradigm. In this approach, lists, arrays, and trees can all be defined, and a subset of the useful operations on such structures is described in the paper. These operations include a "pump" function, which is similar to AQUA's **fold** operation [Leun93]. Since the operations described in [Beer90] are intended to be applicable to any bulk type, not just to lists and trees, they are too general for our purposes -- we wish to distinguish between ordered and unordered types, and we provide a richer set of operations. Furthermore, many of their operations are not described precisely, the existence of certain other operations is assumed, and they do not have any operations like the ones we propose. [Ross92] extends [Beer90] by providing precise conversion operations between various bulk types and transformation rules involving the pump and aggregation operators on trees.

We develop a notation for expressing patterns in trees based on results presented in [That68, Done70].

3.2.3. Summary of the Algebraic Goals

The goals of the EXCESS algebra are to provide operators and optimizations for EXCESS, especially for multisets, arrays, optional object identity, and grouping; to formally define the structures of the algebra so that they conform to the specifications of EXCESS; to demonstrate the applicability of the algebra by proving its equipollence to EXCESS and providing a complete semantics for EXCESS; and to discuss some techniques for optimizing certain queries involving overridden methods. In the context of AQUA, the algebraic goals relevant to this thesis are to provide mechanisms for various notions of equality; a consistent treatment of abstraction; and operators and optimizations for selection queries on trees. See [Vand89] for more discussion on algebraic design in general.

CHAPTER 4

EXTRA AND EXCESS

This chapter describes the EXTRA data model and EXCESS query language. The chapter is organized as follows: Section 4.1 presents the data modeling facilities of EXTRA, including a number of examples. In Section 4.2, we present the EXCESS query language, including its facilities for querying complex objects, performing updates, and computing aggregates. Section 4.3 discusses the support provided in EXTRA and EXCESS for user-defined types and operations. [Vand88b] is a complete user's manual for the EXTRA and EXCESS languages, and [Vand88a] describes an EXTRA database for storing catalog information about an EXTRA/EXCESS DBMS.

4.1. The EXTRA Data Model

In the EXTRA data model, a database is a collection of named, persistent objects. These objects can be as simple or as complex as desired; EXTRA does not constrain the type structure of the named (or "top level") objects in the database. The advantages of going beyond a simple record-based model are detailed in [Kent79]. EXTRA separates the definition of types from the declaration of their instances, and it provides a type system based on a type lattice with multiple inheritance. The type system includes tuple, set, and array as type constructors that may be composed arbitrarily to form new types. EXTRA also provides support for user-defined abstract data types. We will elaborate on each of these features in this section, and we will illustrate them by iterative refinement of a simple example database.

4.1.1. The Basic EXTRA Type System

In EXTRA, the definition of a type and the declaration of instances of that type are completely separated from one another. This makes it possible for a database to include more than one collection of instances of a given type, which can be quite useful in scientific and engineering applications [Lohm83, Kemp87]. While this separation is common in programming languages, it is less common in the database world [Bloo87]. As an example, the commands in Figure 4.1 define a new schema type called Person, which is a tuple type. Two sets for storing Person instances are then created, the Students set and the Employees set. (We will explain the **own ref** syntax shortly; it can be ignored for the purposes of the current discussion. For now, the Students and Employees sets can be thought

of as relations.) In EXTRA, sets are allowed to contain duplicates, so when we refer to a "set" in this chapter we really mean a multiset. Duplicates may be eliminated, if desired, using a "unique" clause, as in QUEL. This is similar to the approach taken in most commercial systems, where relations are actually multisets rather than sets, but are still referred to as relations or sets.

EXTRA provides a variety of base types and type constructors for defining schema types, some of which are used in Figure 4.1. Predefined base types include integers of various sizes, single and double-precision floating point numbers, booleans, character strings, and enumerations. EXTRA also supports the addition of new base types (through an ADT facility similar to those of [Ston86, Ston87b]) like the Date type in Figure 4.1. ADT support will be covered in a later section. The type constructors of EXTRA include tuple (e.g., Person is a tuple type), fixed length arrays, variable length arrays, sets, and references. We will have more to say regarding reference, set, and array type constructors later on, when we describe EXTRA's support for complex objects.

The example of Figure 4.1 is unrealistic in the sense that we will probably want to associate more information with Student and Employee objects than that which is contained in a Person object. Figure 4.2 refines our example, illustrating type inheritance in EXTRA and a more realistic declaration of the Students and Employees sets. The Student tuple type inherits all of the attributes of the Person tuple type, and in addition it has a grade point average (gpa) attribute and a department (dept) attribute. The Employee tuple type adds the attributes jobtitle, dept, manager, and salary to those of the Person tuple type.

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

create Students:        { own ref Person }
create Employees:       { own ref Person }
```

Figure 4.1: Creating schema types and instances

```
define type Student:
(
        gpa:            float4,
        dept:           char[30]
)
inherits Person

define type Employee:
(
        jobtitle:       char[20],
        dept:           char[30],
        manager:        char[ ],
        salary:         int4
)
inherits Person

create Students:        { own ref Student }
create Employees:       { own ref Employee }
```

Figure 4.2: Creating subtypes and their instances

A tuple type may also inherit attributes from more than one type, as the general form of the inheritance clause is **inherits** *type1, type2, ...* If a type inherits two (or more) attributes with the same name, a conflict arises. If these attributes are inherited from a single common supertype, conflict resolution unites them into a single attribute for the new type — since the conflicting attributes have the same source, they also have the same meaning. However, if a name conflict arises between attributes from different sources, no attempt is made to handle the problem automatically. Instead, we require the definer of the type to resolve the conflict explicitly via renaming. For example, suppose we wish to create a new type called "WorkStudyStudent" as a subtype of both the Employee and Student types of Figure 4.2. This would be disallowed due to a name conflict under our rule since the dept attribute would be inherited from two sources with two different meanings: the Student type's dept attribute specifies a student's major, while the Employee type's dept attribute specifies the department where an employee works. Figure 4.3 shows how this conflict can be resolved via renaming. POSTGRES would ignore this conflict because the conflicting dept attributes are of the same data type [Rowe87], while TAXIS would simply disallow the conflict [Nixo87]. EXTRA is closest to ORION [Bane87] in its handling of conflicts, except that we provide no automatic resolution. In the absence of explicit renaming, ORION would select one of the two dept definitions automatically, based on the order of entries in the inherits clause, dropping the other one.

```
define type WorkStudyStudent:
(
        weeklyHours:    int4
        majorDept:      Student.dept,
        jobDept:        Employee.dept
)
inherits Student, Employee
```

Figure 4.3: An example with multiple inheritance

4.1.2. Modeling Complex Objects in EXTRA

A complex object is an object that is composed of a number of component objects, each of which may in turn be composed of other component objects. In general, an object can be a component of more than one object. EXTRA provides four type constructors in addition to the tuple type constructor to support complex object modeling: the **ref** type constructor, the **own** type constructor, the set (or "{ }") type constructor, and the **array** type constructor.

The example in Figure 4.4 shows the schema of Figure 4.2 redefined using **ref** attributes for the dept attribute

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

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

create Students:        { own ref Student }
create Employees:       { own ref Employee }
```

Figure 4.4: Reference attribute example

of the Student type and for the dept and manager attributes of Employee. These are like reference attributes in GEM [Zani83], with each Student object containing (a reference to) its corresponding Department object. The referenced department object is required to exist elsewhere in the database (or else the value of the reference must be null). In addition, the referenced object must be of type Department or else some subtype of the Department type.

Another type constructor that EXTRA provides for modeling complex object types is the set constructor, whose use has already been illustrated for creating the equivalent of a relation in EXTRA. EXTRA allows sets of any data type to be defined/created, and such sets can then contain objects of the specified type and also any of its subtypes. Sets of base types, constructed types, and reference types are all possible in EXTRA. This leads to a very powerful facility for modeling complex objects, as nested relations (ala NF^2 data models) can be supported via sets of tuples, and sets with shared subobjects (ala [Bane87, Lecl87]) can be supported via sets of references. As an example, Figure 4.5 shows the definition for the Department type and the creation of a persistent set (Departments) of objects of this type. The definition of the employees attribute of the Department type specifies it as being a set of references to objects of type Employee. Thus, for a given department, the employees attribute effectively contains all of the employees that work for the department.

While reference attributes and their interaction with the set type constructor permit the development of complex object (graph) structures, there are cases where the database designer wishes to treat an object and its components as a single object that simply happens to have a complex value structure. There are also related cases where the components of an object need to be "full-fledged objects," but where the object should still be treated as a whole,

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

create Departments:     { own ref Department }
```

Figure 4.5: Combining sets and references

as in ORION's composite objects [Kim87]. To support these different cases, EXTRA provides three different kinds of attribute value semantics: **own** attributes, **ref** attributes, and **own ref** attributes. An **own** attribute is simply a value, not a first-class object; it lacks identity in the sense of [Khos86]. By default, all attributes are taken to be **own** attributes unless otherwise specified. An **own ref** attribute is a reference with the added constraint that the referenced object is *owned* by the referencing object[1]; thus, if the referencing object is deleted, the referenced **own ref** object is deleted as well. Finally, a **ref** attribute is simply a reference to another (independent) object, as described earlier, without cascaded deletion semantics. In addition to their semantic importance, **own** and **own ref** provide EXTRA with information that can potentially be exploited for performance reasons (e.g., for clustering).

The combination of set and array type constructors, together with **ref**, **own ref** and **own** modes for attributes, yields a flexible and powerful facility for modeling complex object structures. The database designer can tailor a design appropriately, rather than adapting it to one particular semantics for modeling complex objects. For example, we could extend our Employee type with an additional field for keeping track of employees' children by adding the field definition "kids: { **own** Person }". This says that the kids attribute of an Employee is a set of tuples of type Person.[2] Note that these Person instances are tuple *values*, and not tuple *objects*. They do not have *object identity* as objects do, and therefore they cannot be referenced from elsewhere in the database via a reference attribute. In addition, if an employee is deleted, so are his or her kids. This provides a capability very similar to that provided by NF^2 data models [Dada86, Sche86]. If the kids attribute were instead declared to be of type "{ **own ref** Person }", the deletion semantics would be the same, but children could then be referenced from elsewhere in the database (by **ref** attributes of other objects). As with composite objects in ORION [Kim87], however, a Person instance in the kids set of one Employee instance could not be in the kids set of another Employee instance simultaneously. To overcome this limitation, the kids attribute could be defined simply as "{ **ref** Person }," in which case sharing is permitted and the deletion of an employee instance will not automatically delete the kids. As we will see in the next section, despite their semantic differences, **own**, **ref**, and **own ref** attributes are all treated uniformly in the EXCESS query language for query simplicity. Thus, casual users can ignore the distinction, viewing attributes simply as other objects, as in most object-oriented data models [Lecl87, Horn87, Andr87, Bane87, Maie86c].

---

[1] Note: All objects must have a "home" as an **own ref** component of some other object in order to exist in the database. Named, persistent, top-level objects are automatically made **own ref** components of the database in which they are created. This ensures that automatic garbage collection is not needed.

[2] Since **own** is the default, this is equivalent to "kids: { Person }".

In addition to sets and references, fixed and variable-length arrays are provided and are useful for modeling complex objects. Variable-length arrays are "insertable", meaning that array elements can be inserted or deleted anywhere in the array; such an array can be viewed as a sequence or an indexable list. An array type constructor can be used anywhere a set constructor can be applied, which means that (for example) one can have variable-length arrays of objects. This is useful for modeling complex objects where order is important (e.g., documents).

4.1.3. Other Attribute Types

In addition to the facilities described thus far, EXTRA provides support for two other kinds of attributes: attributes of various types defined by users (i.e., user-defined ADTs), and attributes defined in terms of other values in the database, or derived attributes.

As an example of an ADT attribute, if we wished to include a picture of each employee in the database, we could extend our definition of the Employee type by specifying an attribute "face: Picture" for storing this information. Including this in the definition of the Employee object type says that each Employee will have a face attribute of type Picture, which is a user-defined ADT for storing bit-map images. We will describe EXTRA's ADT facilities (and distinguish them from schema types) in more detail in Section 4.3; basically, though, an ADT can be used wherever any of EXTRA's built-in types can appear.

As for derived attributes, EXTRA allows object attributes to be defined via queries in the EXCESS query language. This facility allows objects to appear to contain certain information which, rather than being stored, can be computed on demand from the current database state. As we will discuss in Section 4.3, derived attributes in EXTRA are similar to the derived or "procedural" data notions of other data models, and they are supported through a facility for associating both EXCESS functions and procedures with EXCESS types. For example, we could extend our Person type definition by defining a derived age attribute, which could be referenced just as if it were a real attribute, as follows:

```
define Person function age returns int4
(
        retrieve (Today - this.birthday)
)
```

In this example, *this* is a special range variable that is implicitly bound to the Person instance to which the function is applied, *Today* is a top-level database object of type Date containing today's date, and the minus operator is a

Date operator that computes the difference in years between two Dates. We will say more about EXCESS functions and procedures in Section 4.3, after the EXCESS query language has been described.

4.1.4. Data Model Summary

To summarize, an EXTRA database is a collection of named persistent objects of any type. EXTRA separates the notions of type and instance; thus, users can collect related objects together in semantically meaningful sets and arrays, which can then be queried, rather than having to settle for queries over type extents as in many data models (e.g., [Ship81, Bane87, Lecl87, Mylo80, Rowe87]). EXTRA provides tuple, set, fixed-length array, and variable-length array as type constructors. In addition, there are three kinds of values, **own**, **ref**, and **own ref** (although casual users such as query writers need not be concerned with this distinction). Combined with the other type constructors, these provide a powerful set of facilities for modeling complex object types and their semantics. Finally, EXTRA provides support for user-defined ADTs and for derived attributes. Figure 4.6 illustrates some of the sort of database structures that can be defined in EXTRA[3], showing how one could create Students, Employees, and Departments "relations," a named object for accessing the best employee directly, an array for keeping track of the top ten employees, a calendar object (represented as a nested array of ADT values of type Day), and an object for storing the current date. The next section will explain how such a database can be queried and updated via the EXCESS query language.

4.2. The EXCESS Query Language

While some may question the need for a general-purpose query language in a database system designed to support emerging application areas such as CAD/CAM, we believe that the inclusion of such a language is indeed justified. Functionality like associative searching can be important in any application domain, even CAD/CAM. For example, if reuse of design components is to become a reality, designers will need to be able to query the database of design objects in order to see if an appropriate component already exists. In addition, a full-function query language makes it possible for the same database system to be used for both business and engineering data, supporting queries such as those needed to compute design costs or to order parts for assembling a design object. Lastly, and most importantly for the purposes of this thesis, associative query languages are important because they are

[3] While Figure 4.6 does not show it, named non-tuple types may also be defined. For example, "**define type** Month: **array** [ ] **of** Day" would let "**create** Calendar: **array** [1..12] **of** Month" be used to create the calendar object in our example.

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

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

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

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

create Students:        { own ref Student }
create Employees:       { own ref Employee }
create Departments:     { own ref Department }
create StarEmployee:    ref Employee
create TopTen:          array [1..10] of ref Employee
create Calendar:        array [1..12] of array [ ] of Day
create Today:           Date
```

Figure 4.6: A complete employee database example

amenable to query optimization techniques.

In this section we present the design of the EXCESS query language. While EXCESS is based on QUEL [Ston76], we have borrowed ideas from the QUEL extensions developed for GEM [Zani83] and POSTGRES [Rowe87, Ston87b] as well as work on SQL extensions for handling NF^2 data [Dada86, Sche86]. Salient features of the EXCESS language include: a uniform treatment of all kinds of sets and arrays, including nested sets; a type-oriented treatment of range variables; a clean, consistent approach to aggregates and aggregate functions; and an

update syntax that supports the construction of complex objects with shared subobjects. In the remainder of this section, we will describe each of the major features of the EXCESS query language. Our examples will be based on the employee database in Figure 4.6.

4.2.1. Set Query Basics

EXCESS provides a uniform syntax for formulating queries over sets of objects, sets of references, and sets of (own) values. For example, consider the following EXCESS query:

range of D **is** Departments
retrieve (E.name) **from** E **in** D.employees **where** D.floor = 2

This query finds the names of all employees who work in departments located on the second floor. The initial range statement specifies that the range variable D is to be bound to the set of department objects in the Departments set; this is the conventional (QUEL) use of a range variable. The phrase "**from** E **in** D.employees" in the query specifies that the range variable E is to be bound to the set of employees for each department that satisfies the selection predicate "D.floor=2". Since D.employees is a set of Employee references, this illustrates how sets of references can be easily manipulated via EXCESS queries. The standard predicates on sets are also provided ("<=" for subset, "in" for membership). EXCESS also supports predicates on arrays -- "in" tests for membership in an array. The predicate "<<=" tests for subarrays. We say that A <<= B if all elements of A appear in the same order in B, with no other elements in between them.

As another 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

Despite the fact that E.kids is a set of values rather than a set of objects, this query looks the same as the previous example because all sets are treated alike in EXCESS queries. (Minor differences do arise for updates, though, as we will describe shortly.)

EXTRA also allows the creation of named persistent objects as single instances of any type (e.g., Today and StarEmployee in Figure 4.6). Such objects can be referenced directly in the EXCESS query language. For example:

```
retrieve (Today)
retrieve (StarEmployee.name, StarEmployee.salary)
retrieve (TopTen[1].name, TopTen[1].salary)
```

Finally, any query result may be saved and assigned a name, making it a top-level object in the database. This is achieved using the keyword "into", as in the following example:

```
retrieve (StarEmployee.name) into BestEmpName
```

4.2.2. Range Variables and Their Types

EXCESS provides several different mechanisms for specifying the set of objects over which a variable is to range. Most are similar to the mechanisms of GEM [Zani83] and POSTQUEL [Rowe87], but EXCESS also provides support for universal quantification (to simplify certain kinds of set queries). The simplest form of range statement has the traditional QUEL syntax, i.e., "**range of** <Variable> **is** <Range_Specification>". For this form of range variable, the <Range_Specification> must identify either a named, persistent set (e.g., Employees), array (e.g., TopTen), or subrange of an array (e.g., TopTen[2:5]). Unless restricted by lower and upper bounds, arrays are treated as sets (except for duplicate semantics on updates). Thus, the statement "**range of** E **is** TopTen" results in E ranging over all objects in the TopTen array, while the statement "**range of** E **is** TopTen[2:5]" restricts E to ranging over the 2nd through 5th objects. As far as the user is concerned, it is immaterial whether the set or array contains objects, references, or values. Also, like GEM, an implicit range variable is provided for each set or array specified in the target list or in the qualification of a query. Thus, our earlier query involving children of second floor employees could have been written as:

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

EXCESS also provides a path syntax in order to simplify the task of formulating queries over nested sets of objects [Ship81, Cope84]. As an example, the statement "**range of** C **is** Employees.kids" means that for each employee object in the Employees set, C will iterate over all the children of the employee. If one of the elements of a path is a single object, it is treated as though it were a singleton set. Thus, the statement "**range of** C **is** Departments.manager.kids" results in C ranging over the manager's children for each department.

While the first form of range statement is used strictly for associating a range variable with a persistent, named set or array, the remaining forms are used for associating a range variable with any set or array. One such form of

range statement has the syntax "**from** <Variable> **in** <Set_Specification>[:< *Type*]". Here, <Set_Specification> may be: a persistent, named set or array (e.g., Employees, TopTen, or TopTen[2:5]); an attribute defined using a set or array type constructor (e.g., E.kids or Departments.employees); or a set formed by taking the union (+), intersection (*), or difference (-) of two or more sets (e.g., "TopTen + StarEmployee" or "Students - Employees").

In an EXCESS query, each range variable has an associated type, and the query may only refer to attributes associated with this type. In many cases, the appropriate type can be inferred easily from the query. For example, in our queries involving children of second floor employees, the type of the variable ranging over the Employees set is Employee, the type of the elements in the set. If the set over which a variable ranges is the union, intersection, or difference of two or more sets (e.g., "**from** P **in** Students - Employees"), it is required that the sets share a common supertype. If this supertype is unique, then the type of the range variable is inferred to be the most specific common supertype of the sets' types. If no unique supertype exists, then the normally optional *Type* specification in the range syntax must be used to identify the particular supertype relevant to the query. If the set over which a variable ranges is not a binary combination of other sets, the ":< *Type*" syntax may still be used; in this case it means that all elements that are not of type *Type* (or one of its subtypes) will be ignored during the query. The syntax ": *Type*" is also provided, and means that all elements that are not of exactly the type *Type* will be ignored. For example, ": Person" means ignore all Students and Employees, but process all other Persons.

The last two forms of range statement in EXCESS are used to specify universally or existentially quantified range variables. The syntax for this form of range statement is "**forall** <Variable> **in** <Set_Specification>[: <Type>]" or "**forsome** <Variable> **in** <Set_Specification>[: <Type>]". Its use is illustrated in the following query, which retrieves the names and ages of all highly paid employees whose children are all under 5 years old:

retrieve (E.name, E.age) **from** E **in** Employees
where E.salary > 100000 **and** (**forall** C **in** E.kids C.age < 5)

Universal quantification simplifies the specification of queries where it is desired that all elements of a set or array satisfy some property. Otherwise, such queries must be specified using an awkward combination of aggregates (probably making them more difficult to optimize).

4.2.3. Query Result Types

In the preceding section, the result of each example query was a set of tuples for which the type of each attribute was **own**. In general, however, the result of a query can be a set of objects or a set of tuples in which one or more attributes is an object (**ref** or **own ref**). As an example, consider the following query which, for each employee making over $100,000, returns the name of the employee and his or her department:

range of E **is** Employees
retrieve (E.name, E.dept) **where** E.salary > 100000

If this query were embedded in a programming language, then the object identifier for each department could be bound to a host variable for subsequent manipulation. However, in the case of an ad-hoc query (where **retrieve** really means *print*), printing the object identifier of each department is not acceptable. Since there does not seem to be a single "right" answer to the question of what to print, the solution we have adopted is that when a retrieve query returns an object as the result of an ad-hoc query, the system will print each **own** attribute of the object (recursively, if it is a structured attribute). Thus, the above query is equivalent to:

range of E **is** Employees
retrieve (E.name, E.dept.name, E.dept.floor)
where E.salary > 100000

To facilitate the specification of "deep" retrieval operations, fields in the target list of a retrieve query can be tagged with a "*" operator to indicate that the object plus all of its component (**own ref** as well as **own**) objects are to be recursively retrieved. This capability will be especially useful for those applications (e.g., CAD/CAM) that need to extract an object and all of its component objects from the database with a single retrieve statement.

We now present a few examples to illustrate the structure of the result of an EXCESS query. Consider the following query:

retrieve (K) **from** K **in** Employees.kids

This query returns the set of all children of employees. The elements of this set are of type Person. The following query returns a set of sets of children, one set for each employee:

retrieve (Employees.kids)

These interpretations were chosen for several reasons. First, it is useful to be able to produce results in either form. Second, it is intuitive--the presence of only "K" in the target list seems to indicate that the user wants a set of objects

of the type that "K" ranges over. With "Employees.kids" in the target list, it seems as if a structured result is desired, and this is also consistent with the treatment of multi-argument target lists, such as "**retrieve** (Employees.kids, Employees.name)", in which the kids and the names retain their relation to their containing structure.

Finally, this interpretation is consistent with our interpretation of range variables over nested sets. For example, in the following statements, B takes on values that are sets of int4, C takes on values that are int4, the first retrieval prints out a set of sets of int4, and the last statement prints out a set of int4:

```
create A: { { int4 } }
range of B is A
range of C is B
retrieve (B)
retrieve (C)
```

EXCESS also provides syntax to allow for the creation of a singleton set. For example the following query takes the named, top-level object TL and returns a set containing only TL:

```
retrieve ( { TL } )
```

Similar syntax is provided to create a singleton array (using "[" and "]"), a tuple with one field (using "(" and ")") and a reference to another object (using "<" and ">").

4.2.4. EXCESS Join Queries

Like GEM, EXCESS provides three types of joins: *functional joins* (or implicit joins, using the dot notation); *explicit identity joins*, where entities are directly compared, and traditional relational *value-based joins*. A number of our examples have involved functional joins. As a further example, consider the following query:

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

This query selects the employees living in Madison and then finds the name of their departments using the Department reference attribute of the Employee type as an implicit join attribute. The same query can also be expressed using an explicit identity join:

```
retrieve (D.name) from D in Departments, E in Employees
where E.dept is D and E.city = "Madison"
```

In this form, the value of the reference attribute E.dept is directly compared with the range variable D using the **is** operator. The **is** operator is useful for comparing references, returning true if two references refer to the same

object. Thus, **is** is a test for object equality rather than (recursive) value equality in the sense of [Banc86]. An **isnot** operator is also provided for convenience in testing that two references do not refer to the same object. As in GEM, these are the only comparison operators applicable to references.

It is expected that most joins in EXCESS will be functional joins, as the important relationships between entities can be expressed directly through reference attributes; most other joins are expected to be explicit identity joins. However, traditional value-based joins are also supported in the EXCESS query language, as in GEM. For example, the following query finds the names of all persons living in the same city as Jones:

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

4.2.5. Aggregates and Aggregate Functions

Since aggregates add important computational power to a query language, we felt that it was important to address them carefully in EXCESS. In deciding how to integrate aggregates into the EXCESS query language, we have tried to provide an intuitive semantics for aggregates and range variable binding. We have also tried to make a clear distinction between the attributes used for partitioning, the attributes being aggregated, and other query attributes. The following syntax is used for expressing EXCESS aggregates and aggregate functions:

agg-op(X [**over** Y] [**by** Z]
[**from** <Variable> **in** <Set_Specification>] [**where** Q])

The execution of an aggregate can be viewed logically as follows: First, the qualification Q is applied to each element ranged over by the range variable (i.e., a selection is performed). The resulting set of values is then projected on the attribute(s) specified in Y, with duplicates (if any) being eliminated in the process; if no **over** Y clause is specified, duplicate elimination is not performed.[4] If a **by** Z clause has been specified, the set is then partitioned using the Z attribute(s) as a key. Note that if Z=Y, then each tuple forms its own partition; if no **by** clause is specified, then there is just one partition (in which case the aggregate is being used as a scalar aggregate). Finally, the operation agg-op is applied to the attribute(s) specified by X, yielding one result value for each partition. Both X and Z must be subsets of Y. For agg-op, EXCESS provides all of the usual built-in aggregates (e.g., sum, count,

[4] Using **over** in this way facilitates certain queries that would otherwise be difficult to express using QUEL unique aggregates or aggregates with SQL-like **unique** clauses, and it also renders such uniqueness clauses unnecessary [Klau85].

avg, max, and min).

As a first example, the following EXCESS query will find the average salary of all employees:

retrieve (avgsal = avg(E.salary **from** E **in** Employees))

In order to fulfill our design goal of a clear, consistent rule for range variable binding, we use Pascal-like scoping rules for range variables.[5] First, a range variable declared in the outer query can be used within the where clause of an aggregate; however, if the same variable is redeclared within the aggregate, the effect is to define a new range variable. Second, since aggregates can be viewed as self-contained queries, *all* range variables declared inside an aggregate are strictly local to that aggregate expression. These rules considerably improve the semantics of range variables in aggregates over QUEL. A consequence is that the result of an aggregate must be bound explicitly to the remainder of an aggregate query. For example, the following query finds the name of each employee plus the average salary of all employees who work for his or her department:

range of EMP **is** Employees
retrieve (EMP.name, avg(E.salary **by** E.dept
from E **in** Employees **where** E.dept **is** EMP.dept))

In this example, the variable E ranges over the Employees set within the aggregate function. In order to bind the results of the aggregate function to the outer query, the clause "E.dept **is** EMP.dept" must be included. In QUEL, this binding could be accomplished implicitly by using a single range variable for the entire query.

An example will illustrate the separation of aggregation and partitioning attributes; it will also show how one can operate on attributes from one level of a complex object while partitioning on attributes from other levels. The following query retrieves the name of each employee; for each employee, it also 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 **by** E.dept.floor
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 E within the aggregate is connected to EMP declared in the range statement via a join on Employee.dept.floor. The query aggre-

[5] Our scoping rules are similar to those of POSTQUEL [Rowe87].

gates over Employee.kids, a set-valued attribute, and partitions the data using an attribute of Employee.dept, which is a single-valued reference attribute.

As an example of the use of the "over" clause, consider the following query:

```
range of EMP is Employees
retrieve (avg(EMP.salary over (EMP.salary,EMP.name)))
```

If employees can work for more than one department, the "over" clause ensures that each employee's salary is only counted once in computing the average.

4.2.6. Updates in EXCESS

EXCESS provides four commands for updating a database: **insert**, **copy**, **replace**, and **delete**. The **insert** command takes an object and adds a reference to it to a collection (set or array). The **copy** command is similar to the **insert** command except that it creates a new object, either by making a copy of an existing object or by using values supplied as parameters to the command. To make a copy of an existing object, the **own** components of the object are copied recursively, new objects are created recursively for all components of type **own ref**, and references are simply copied for components of type **ref**. Thus, the original object and the copy will share component objects referred to by attributes of type **ref**. For both **insert** and **copy**, the target collection must already exist.

The following example illustrates the use of the **copy** command to create a new employee object and add it to Employees set. The effect of this update is to add Smith to the Employees set, and to have him work for Jones in the Computer Science Department.

```
copy to Employees
(name = "Smith", street = "1210 W. Dayton St.",
city = "Madison", birthday = "1/1/64", dept = D,
manager = M, jobtitle = "programmer", salary = 50000)
from D in Departments, M in Employees
where D.name = "Computer Sciences" and M.name = "Jones"
```

The following insertion query completes the hiring process by adding a reference to Smith to the employees attribute of the Computer Science object in the Departments set:[6]

---

[6] It would be nice also to ensure that the employees attribute of the appropriate department object is updated whenever an employee is hired or fired. Using the extensibility features described in Section 4.3, it is possible to define a *HireEmployee* procedure that encapsulates the appropriate pair of update queries as a single command. This effect could be achieved automatically by adding a facility for inverse relationship maintenance, similar to that provided by DAPLEX [Ship81].

```
insert into D.employees (E)
from D in Departments, E in Employees
where E.name="Smith" and D.name = "Computer Sciences"
```

To add a child of Smith's to the database, the following query would be used:

```
copy to E.kids
(name="Anne", street = E.street, city = E.city, zip = E.city,
birthday = "10/10/79") from E in Employees
where E.name = "Smith"
```

Since any object can be owned by only one **own ref** referencing object, the following **insert** command will fail:

```
create overpaid {own ref Employee}
range of E is Employees
insert into overpaid (E) where E.salary > 100000
```

The problem here is that employee objects are already owned by the Employees set object. However, this would work if overpaid were created as "{**ref** Employee}"; alternatively, it would work with **copy to** used in place of **insert into** (thereby copying each employee object).

Support is also provided for updates to arrays. For example, the following query will put the employee with the highest salary in the first position of the TopTen array (declared in Figure 4.6).

```
range of E is Employees
insert into TopTen [1] (E)
where E.salary >= max (X.salary from X in Employees)
```

An **insert** or **copy** operation on a fixed length array is performed by replacement beginning at the specified index position. If there is more than one employee with the maximum salary, the next employee will be inserted at Top-Ten[2], and so on. Updates on variable length arrays are performed by inserting immediately before the specified position.

The **replace** command is used to modify objects. For example,

```
replace E (salary = E.salary * 1.1)
from E in Employees where E.name = "Jones"
```

will give Jones a 10% raise. Lastly, to illustrate the use of the **delete** operator, the following pair of EXTRA update queries would be used to fire Smith:

```
delete E from E in D.employees, D in Departments
where D.name = "Computer Sciences" and E.name = "Smith"

delete E from E in Employees where E.name = "Smith"
```

The deletion semantics for **own** values and **own ref** objects are that (i) they are deleted when their owner is deleted, and (ii) deleting them causes them to be removed from the database as well as from the containing object. Thus, when "Smith" is deleted from the Employees set, his object is deleted from the database and his children are recursively deleted. (Deleting a **ref** object deletes only the reference itself, and not the object.)

4.3. Extensibility in EXTRA and EXCESS

A goal of the EXTRA data model and EXCESS query language was to provide users with the ability to define new types, along with new functions and operators for manipulating them. EXTRA supports two kinds of type extensions: schema types and abstract data types (ADTs). Schema types were the subject of Section 4.1, where we described how new schema types can be constructed from base types and other schema types using the EXTRA type constructors. In this section, we describe how the set of base types can be extended by adding ADTs. We then consider schema types again, describing how functions and procedures written in EXCESS can be associated with schema types and how data abstraction can be achieved. We distinguish between these two kinds of type extensions because of the different facilities that are provided for implementing them: ADTs are written in the E programming language [Rich87], using the type system and general-purpose programming facilities provided by E. Schema types are created using the type system provided by the EXTRA data model and the higher-level but more restrictive programming facilities offered by the EXCESS query language. Finally, we discuss how EXCESS may be extended with new set functions (e.g., new aggregates).

4.3.1. Abstract Data Types

New base types can be added to the EXTRA data model via the EXTRA abstract data type facility. To add a new ADT, the person responsible for adding the type begins by writing (and debugging) the code for the type in the E programming language. E is an extension of C++ [Stro86] that was developed as part of the EXODUS project. E serves as the implementation language for access methods and operators for systems developed using EXODUS. It is also the target language for the query compiler, and (most importantly for our purposes here) the language in which base type extensions will be defined. E extends C++ with a number of features to aid programmers in data-

base system programming, including "dbclasses" for persistent storage, class generators for implementing "generic" classes and functions, iterators for use as a control abstraction in writing set operations, and built-in class generators for typed files and variable-length arrays [Rich87].

Suppose that we wanted to add complex number as a new ADT. First, we would need to implement the ADT as a dbclass (much like a C++ class) in E. Figure 4.7 gives a slightly simplified E interface definition for the Complex dbclass. The hidden internal representation of an object of this dbclass stores the real and imaginary components of complex numbers. The Complex dbclass also provides a number of public functions and procedures (known as member functions in C++ terms). In addition to their explicit arguments, all member functions have an implicit first argument, *this*, which is a reference to the object to which the function is being applied. The Complex dbclass has two "constructor" member functions for initializing newly created complex numbers, one that uses externally-supplied values[7] and another that assumes a default initial value (e.g., 0's). In addition, the dbclass has member functions to add and multiply pairs of Complex numbers, and member functions to extract the real and imaginary parts of a Complex number. Finally, the dbclass also has a Print member function. The EXCESS query language interface requires a Print member function for all ADTs.

Once a new data type has been implemented in E and fully debugged, it must be registered with the system so that the parser can recognize its functions and the query compiler can generate appropriate E code. To register the

```
dbclass Complex {
        float           realPart, imagPart;
public:
                        Complex(float&, float&);
                        Complex( );
        Complex         Add(Complex&);
        Complex         Multiply(Complex&);
        float           Real( );
        float           Imaginary( );
        void            Print( );
}
```

Figure 4.7: Contents of /usr/exodus/lib/adts/complex.h

[7] This constructor is invoked when a Complex object is created and arguments are specified, e.g., when the target list for a **copy** includes an entry of the form "... complexValue = (25.0, 10.0)".

new ADT Complex, the implementor of the type would enter the following (assuming that "complex.h" contains the Complex interface definition and that "complex.e" contains the E code that implements it):

```
define adt Complex
(
        interface = /usr/exodus/lib/adts/complex.h,
        code = /usr/exodus/lib/adts/complex.e
)
```

This tells EXTRA/EXCESS where to find the definition and implementation of the type. The system uses the definition to extract the function name and argument type information needed for query parsing and type checking. Types included in the public portion of the definition must also be registered EXTRA base types, of course. The implementation file (or list of files, in general) will be compiled and stored by the system for use when queries that reference the Complex type are encountered.

After Complex has been registered as a new ADT, it can be used like any other base type in EXTRA/EXCESS schemas, queries, and updates. The default notation for member function invocation is similar to that of C++ and E (but "( )" may be omitted for functions with no explicit arguments). For example, if CnumPair is an object in the database with two complex-valued fields val1 and val2 (or a range variable bound to such an object), then the expression "CnumPair.val1.Real" will invoke the member function that extracts and returns the real portion of the val1 field. Similarly, the expression "CnumPair.val1.Add(CnumPair.val2)" will add the val1 and val2 fields together, producing a result of type Complex. Since some users may prefer a more symmetric function call syntax, EXCESS will also accept this expression in the form "Add(CnumPair.val1, CnumPair.val2)".

In addition to supporting standard ADT function invocation, we follow the lead of [Ong84, Ston86, Ston87b] and support the registration of operators as an alternative function invocation syntax. For example, to define "+" as a infix operator synonym for the Add member function of the Complex dbclass, we would enter:

```
define adt-op "+" infix for Complex Add
```

With this definition, the previous example's addition can be rewritten more naturally as "CnumPair.val1 + CnumPair.val2". Existing EXCESS operators can be overloaded, as illustrated here. In addition, it is possible to introduce new operators (any legal EXCESS identifier or sequence of punctuation characters may be used). For new operators, we require the precedence and associativity of the operator to be specified, much as in [Ston87b]. Prefix operators can also be defined, and the number of arguments that an operator can have is not restricted. How-

ever, functions with three or more arguments cannot be defined as infix operators, and functions that are overloaded within a single dbclass may not be defined as operators.

In order to ensure that ADTs fit into the EXTRA query language in much the same way as built-in base types do, we restrict the form and functions of their dbclasses somewhat. In particular, we require that they be true abstract data types — their data portion must not be public. We also require function (and thus operator) results to be returned by value to ensure that expressions behave in the expected manner. And, while procedures are permitted to have side effects, side effects are currently forbidden for functions so that query results will not be dependent upon query predicate evaluation order.[8] ADT functions and operators may be used anywhere that built-in base type functions and operators may appear in queries, but ADT procedure calls may only appear in update query target lists. By default, equality and assignment operators for ADTs are predefined with bit-wise comparison/copy semantics (as in C++ and E); if this is inappropriate for a given type, the implementor of the type can replace the default definitions by explicitly overriding them in the E dbclass definition.

Finally, it is important that the query optimizer be given sufficient information to recognize the applicability of alternative access methods and join methods when optimizing queries involving ADTs. We will follow an approach similar to that outlined in [Ston86, Ston87b] for addressing this issue, with a few differences. First, optimizer-specific information will not be specified via the EXCESS/EXTRA interface. Instead, it will be given in tabular form to a utility responsible for managing optimizer information. The EXCESS query optimizer will do table lookup to determine method applicability for ADTs (so that ADTs can be easily added dynamically). These tables will be similar to the access method templates of [Ston86], but expression-level optimizer information (e.g., associativity, commutativity, complementary function pairs, etc.) will also be represented in tabular form at this level. Second, ADT functions and operators will be treated uniformly for query optimization purposes. This is different than [Ston87b], where operators but not functions are optimized. Third, we will support the addition of both new access methods and new join methods (written by expert users, or "database implementors") to the DBMS through the rule-based optimization techniques detailed in [Grae87].

---

[8] The function/procedure distinction here is based on the existence or absence of a return value. We may relax our side-effect-free assumption for functions eventually, and it is a convention and not a constraint that the system will enforce.

4.3.2. EXTRA Schema Types, Functions, and Procedures

Schema types, which were described in Section 4.1, are defined using the type system provided by the EXTRA data model. In addition, we provide facilities analogous to member functions of ADTs for defining EXCESS functions and procedures to operate on schema types. However, the fact that they are defined using the EXTRA type system and the EXCESS query language means that query optimization techniques can be applied to them. EXCESS functions and procedures are inherited through the type lattice in a manner similar to inheritance for attributes, giving EXTRA an object-oriented flavor. The goal of these facilities is to provide support for derived data, for writing "stored commands" (as in the IDM-500 query facility [IDM500]), and for data abstraction of the kind described by Weber [Webe78].

4.3.2.1. EXCESS Functions

As illustrated in an example in Section 4.1.3, associating functions with schema types provides a mechanism for defining derived attributes. Such functions can be defined via a single EXCESS query of arbitrary complexity, involving other portions (possibly derived) of the object being operated on and/or other named database objects. For example, we could associate a function with the Employee type to return the floor that an employee works on as follows:

```
define Employee function floor returns int4
(
        retrieve (this.dept.floor)
)
```

Such functions can then be used in queries just like regular attributes, e.g.:

```
retrieve (Employees.floor) where Employees.name = "Smith"
retrieve (Employees.name) where Employees.floor > 10
```

EXCESS functions can also have arguments other than their implicit *this* argument. For example, we could define the following function and query:

```
define Employee function
youngerKids(maxAge: int4) returns int4
(
        retrieve (count(C from C in this.kids
        where C.age < maxAge))
)

retrieve (Employees.name, Employees.youngerKids(10))
```

This example, involving a function that counts the number of children under a certain age that an employee has, illustrates several points. First, it shows how function arguments may be defined and used. Second, it shows that a function can be defined in terms of other functions. (Recall that age was defined as a Person function in Section 4.1.3.) EXCESS function and procedure arguments are passed by reference.

A function may return a result of any type, including a schema type, a set of some schema type, etc. Updates through functions are not permitted. By way of comparison, EXCESS functions are similar to functions in DAPLEX [Ship81] and IRIS [Fish87]. They can also be viewed as a simplified form of POSTGRES procedure attributes [Ston87a]; in particular, they are like parameterized procedure attributes. We do not support true procedure attributes, or "EXCESS as a data type," because procedure attributes are not needed for representing complex object structures in EXTRA as they are in POSTGRES. Also, since such attributes are known only to be of type POSTQUEL in POSTGRES, and may contain an arbitrary series of POSTQUEL queries, they are problematic with respect to type-safety and knowing what to expect when writing application programs involving such attributes.

4.3.2.2. EXCESS Procedures

In addition to functions, we support the definition and invocation of EXCESS procedures. These differ from functions in that they have side effects, they can involve a sequence of EXCESS commands (instead of a single command), and they do not return results. As an example, suppose that we wished to write a procedure for the Employee schema type to move an employee from one department to another. We might define such a procedure as follows:

```
define Employee procedure
ChangeJob(oldDept, newDept: Department)
(
        delete E from E in oldDept.employees where E is this
        insert into newDept.employees (this)
        replace (this.dept = newDept)
)
```

Then, if we wanted to move all of the employees in the Database Systems department into the Knowledge Base Systems department, we could use our new procedure to do the job as follows:

```
Emp.ChangeJob(Emp.dept, NewDept)
from Emp in Employees, NewDept in Departments
where Emp.dept.name = "Database Systems"
and NewDept.name = "Knowledge Base Systems"
```

As this example shows, procedures are called using a syntax similar to that of the EXCESS update commands.

Since not all procedures may be naturally associated with a single EXTRA schema type, and some might argue that the example above is a case in point, we also support procedures that are not bound to a particular type. (We support this option for functions as well.) For example, the procedure above could alternatively be defined as "**define procedure** ChangeJob(emp: Employee; old, new: Department)", with Database Systems employees being passed in explicitly. This kind of procedure is similar in flavor to the stored commands of the IDM database machine [IDM500]. However, it is much more general, as we support the use of a **where** clause for binding procedure parameters and we invoke the procedure for all possible bindings (instead of just once, with constant parameters).

Aside from their syntactic differences, the major difference between procedures that are associated with an EXTRA schema type and those that are not type-specific is a matter of inheritance semantics. In the type-specific case, we may choose to redefine the ChangeJob procedure for some subtypes of Employee, such as WorkStudyStudent, but not for others, like NightEmployee. Then, if ChangeJob is invoked on elements of a set containing instances of Employee and its subtypes, WorkStudyStudent's ChangeJob will be used for WorkStudyStudent instances, while Employee's ChangeJob will be used for Employee and NightEmployee instances. That is, procedures can be inherited via the inheritance mechanism outlined in Section 4.1. If ChangeJob is not associated with a particular type, its definition may still be overloaded. However, a single definition (i.e., the definition applicable to the Employee type) will be statically selected for such a query. This is similar to the distinction between virtual member functions and regular member functions in C++ [Stro86].

4.3.2.3. Achieving Data Abstraction

We provide an authorization mechanism along the lines of the System R [Cham75] and IDM [IDM500] protection systems; this mechanism is described in [Vand88a, Vand88b]. Both individual users and user groups (including a special "all-users" group) will be recognized, and protection units will be specified via EXCESS queries. Grantable/revocable rights will include the ability to read and modify specified objects or individual attributes, the ability to invoke specified functions and procedures, the ability to define new functions and procedures for specified types, etc. This protection system will serve two purposes: First, it will act as an authorization mechanism for protecting database objects against unauthorized access and modification, as is typical. Second, it will provide a means for achieving data abstraction (i.e., encapsulation) for EXTRA schema types. For example, one could choose to grant access to a given schema type only via its EXCESS functions and procedures, effectively making the schema type an abstract data type in its own right. (In fact, IDM stored commands are recommended for regulating database activity in a similar way [IDM500].) Features such as the modules of [Webe78] or the object semantics of an object-oriented data model can thus be captured via a single, more general mechanism.

4.3.3. Generic Set Operations

The final form of extension that we intend to support in the EXCESS query language is a facility for adding new, user-defined set functions to the language. At a logical level, such a function can be viewed as taking a set of elements of some type T1 as its argument and returning an element of type T2 as its result. T1 can be any type that satisfies certain constraints, while T2 must be either a specific type or the same type T1. Aggregate functions are a classic example here: The "min" function takes a set of values of any type on which a total order is defined, returning the smallest element of the set. The "count" function takes a set of values as its argument, returning the number of elements in the set (i.e., an integer result). One could imagine adding new functions of this sort, such as "median" or "std-deviation" for use in statistical applications.

Support for this form of extension has been included in POSTGRES, but in a limited form. In particular, one could introduce a "median" aggregate function for sets of integers, but not one that works for *any* totally ordered type [Ston87b]. The EXCESS approach to such extensions is based on features provided by the E programming language [Rich87]. E provides a facility for writing generic functions, and it supports the specification of constraints on the generic type (e.g., any type that has boolean "less_than" and "equals" member functions). E also pro-

vides a construct, called an iterator function, for returning sequences of values of a given type. Implementing a new aggregate will involve writing a generic E function with one argument, an iterator that yields elements of an appropriately constrained type T; the generic function should produce a result of either the same type T or else some base type. This function can then be registered with EXCESS and used with any type that meets its constraints.

CHAPTER 5

THE EXCESS ALGEBRA

As described in Chapter 4, the EXTRA/EXCESS system 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 features is satisfactorily supported in relational algebra or in any of the more recent advanced algebras. This chapter 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. Most such proofs are between algebras and calculi; we omit a calculus and prove direct correspondence with a user-level query language. The proof is constructive and demonstrates some new techniques.

The remainder of the chapter is organized as follows: Sections 5.1 and 5.2 define and motivate the algebra's structures and operators, which are further explained by example in Section 5.3. Section 5.4 demonstrates the algebra's equipollence to EXCESS. 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 5.5. In Section 5.6 we illustrate some of the more interesting new transformation rules of the EXCESS algebra by giving some examples that make use of them. Section 5.7 concludes the chapter with some comments on the implementation of EXTRA/EXCESS. A list of the algebraic transformation rules for EXCESS can be found in Appendix A. The complete equipollence proof is in Appendix B.

5.1. The Algebraic Structures

The full definitions of algebraic structures are presented here 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 $S = (\mathbf{V}, \mathbf{E})$, 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). Notice that the "set" constructor, as discussed in Chapter 4, is really a multiset constructor; "set" is used as a convenient shorthand. There is no pure "set" type in EXTRA. 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 5.1 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 object.

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 ele-

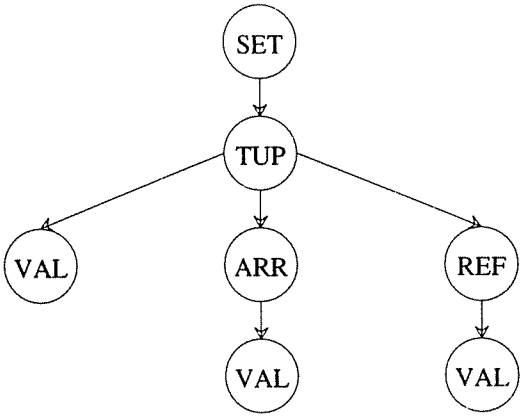

Figure 5.1: A schema

ment 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: $dom(S) = \overset{n}{\underset{i=1}{\times}} dom(S_i)$, 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 (single) component of *S*. The domain is the set of all multisets such that every element of the multiset appears in the domain of the child of the multiset node.

iv) arr: $dom(S) = (\overset{\infty}{\underset{j=1}{\cup}} (\overset{j}{\underset{i=1}{\times}} (dom(S1)))) \cup \{ [ ] \}$, where *S1* is the (single) 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 (single) 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 $m \neq n$, $R(m) \cap R(n) = \emptyset$. 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 5.1 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 OIDs whose value is not available to the user.

Note that so far this definition does not take (multiple) inheritance or null values 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). Also, we provide two types of null values, *unk* and *dne*. The *unk* null appearing as the instance part of a structure indicates that we do not know the value of the structure, but it does have a value. The *dne* null indicates that the structure does not really exist and can be discarded at an appropriate time during query processing (e.g., when no information will be lost by discarding it). More formally, then, we redefine a domain for schema S to be DOM(S), defined as follows:

DOM(S) = {*unk*, *dne*} ∪ *dom(S)* ∪ $(\bigcup_{i=1}^{n} dom(S_i))$, 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. For example, if we have A → B, then our 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 see why this is so, notice that A → B implies that the domain of type A includes the domain of type B. Also notice that the formal definition for the domain of a set of entities of type A includes all subsets of the domain of A. Thus something of type "set of A" can take on a value that is a set of B entities. However, part (v) of our definition says that the domain of the type "ref A" is a set of object identifiers. These include the object identifiers for the type A and (via the DOM definition) the object identifiers for all subtypes of "ref A". But our inheritance lattice consists only of A → B, so we are left with a DOM consisting of only the identifiers for objects of type A exactly, and in particular not containing any identifiers for B objects.

Thus the definitions presented so far do not quite model the EXTRA semantics (substitutability). To reflect those semantics with the current definition we would explicitly need "**ref** A $\rightarrow$ **ref** B", which is different than "A $\rightarrow$ B". This is the motivation for the following results, which explicitly specify that OIDs for subtypes can appear where OIDs for supertypes are expected. Again, the reason for this is that the constructions of parts (i)-(iv) above use actual domains in constructing other domains, but in part (v) we merely used abstract OIDs without explicitly including subtypes.

We present (intuitively and formally) five rules that specify the semantics of OID domains, where *Odom*(A) will indicate the domain of all OIDs for the type named A.

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

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

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 $\mathit{Odom}(S) = \bigcup_{i=1}^{n} \mathrm{Odom}(S_i)$:

$$R \rightarrow S \Rightarrow |\mathit{Odom}(R) - \mathit{Odom}(S)| = \infty$$

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:

$$R \rightarrow S \Rightarrow \mathit{Odom}(S) \subset \mathit{Odom}(R).$$

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:

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

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:

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

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. There is a set-theoretic way of formally specifying the constraints on the domains of schemas of type REF. There is a simple, intuitive construction that satisfies these formal constraints and can be incorporated into the general definition of *dom*(S) for any schema S. We thus have a consistent semantics for the domains of all EXTRA types, including those using multiple inheritance among named tuple types and optional object identity. Note also that these semantics allow type migration [Fish89] to occur, as long as substitutability is preserved along the way.

5.2. 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 A for a list of these rules). At first this may seem to cause an unacceptable increase in the size of the search space the optimizer will need to examine, but as will be seen, 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, reference, and predicate operators, respectively, giving examples of each operator. Finally, we describe how generic multiset functions (which include aggregates) fit into the algebra. Null values are treated as constants unless specified otherwise. Each operator of the algebra takes one or two structures as input and produces a single, entirely new structure as output.

5.2.1. Multiset Operations

A multiset [Knut81, Liu77] (also called a "bag" or "collection") is a set that allows a specific element to appear more than once in the set. Every existing object has an associated cardinality in every multiset S, referred to as $card_S(x)$, and $card_S(x) \geq 0$. Two multisets are equal if every element appearing in either multiset has the same cardinality in both.

We provide the following primitive operations on multisets: difference, additive union, Cartesian product, duplicate elimination, grouping, creation of a multiset, destruction of a multiset, and application of an algebraic expression to all elements of a multiset. It is interesting to note that the identity $A \cap B = A - (A - B)$ holds for multisets (when nulls are not involved) as well as sets.

The first two operators (difference and additive union) and two of the array operators, ARR_CAT and ARR_DIFF, are defined only if the schemas of the elements of the inputs are *compatible*; i.e., if they are the same schema or have a common supertype in the tuple inheritance hierarchy (or, if they are references to tuple types, if their referenced types have a common supertype). Four of the multiset operators (GRP, $\uplus$, $-$, and $\times$) can also have any input be of type array; however, the results are still multisets, so we describe them here. In these operators, if an input structure is an array, we simply ignore the ordering property of the structure and treat it as a multiset.

Finally, three of the operators take parameters that enable them to mimic the ": *Type*" and certain uses of ":< *Type*" syntax of EXCESS. The GRP, SET_APPLY, and ARR_APPLY operators all may take a parameter that describes the types of elements to be ignored during the query. These parameters are described below. Note that in the operator definitions we treat null values in such a way as to preserve as much information as possible.

1) **Difference** ($-_{type}$): This operation is defined as a multiset difference. The subscript indicates the type of the result and will usually be omitted unless the operands are of unequal *compatible* types with no unique least common supertype. The cardinality of an element in A – B is:

$$card_{A-B}(x) = \begin{cases} card_A(x) \text{ if } x = unk, \\ \max(0, card_A(x) - card_B(x)) \text{ otherwise} \end{cases}$$

Example: Let A = {1, 1, 2, 3} and B = {1, 2, 2}. Then A – B = {1, 3}.

2) **Additive Union** ($\uplus_{type}$): The subscript indicates the type of the result (see above). The cardinality of an element in the additive union is the sum of the two original cardinalities:

$$card_{A \uplus B}(x) = card_A(x) + card_B(x)$$

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

3) **Cartesian Product** ($\times$): This is essentially the same as the set-theoretic Cartesian (cross) product operator. Each occurrence of every element in A is paired with each occurrence of every element in B to give a multiset of 2-tuples, each of which has an element of A in the first field and an element of B in the second field.

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

4) **Duplicate Elimination** (DE): This operator converts a multiset into a true set. Any element with cardinality > 1 has its cardinality changed to 1:

$$card_{DE(A)}(x) = \begin{cases} card_A(x) \text{ if } x = unk, \\ 1 \text{ otherwise} \end{cases}$$

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

5) **Grouping** (GRP): This operator partitions a multiset (or array; see above) into equivalence classes based on the result of an 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 A. Each of these multisets will contain the original occurrences appearing in A, not the results of applying the expression to these occurrences. The special symbol "INPUT" is used to indicate an occurrence of the input multiset; it is a primitive form of variable binding[1]. Recall that GRP also has an optional parameter (not shown in the example below) that can be of the form ": *Type*" or ":< *Type*". The first form indicates that the multiset should be interpreted as if only occurrences of type T that are not also occurrences of any subtype of T are in the multiset. The second form indicates that the multiset should be interpreted as if only occurrences of type T and its subtypes are in the multiset; all other elements are ignored.

---

[1] To simplify the presentation we use only this special symbol. More complete and flexible variable binding is easily obtained by specifying an additional parameter to GRP (and to SET_APPLY, ARR_APPLY, and COMP) which indicates the name to be used, instead of "INPUT", to represent successive elements of the input. All comments and results regarding the algebra still hold if this extension is made.

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 later). This GRP operation partitions the tuples of A into equivalence classes based on the value of the first attribute of the tuple.

6) **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. This operator is useful (in conjunction with $\uplus$), for example, when one wishes to add a single element, which does not already occur inside some multiset, to an existing multiset.

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

7) **Destroy a Set** (SET_COLLAPSE): The SET_COLLAPSE operator takes a multiset of multisets and returns the $\uplus$ of all the member multisets. A similar operator appears in [Abit88a].

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

8) **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 instance is simply the multiset consisting of the results of applying E to each occurrence of every element in the input multiset. If the result for any occurrence is *dne*, this constant is not placed in the result multiset. The special symbol "INPUT" is used to indicate an occurrence of the input multiset; it is a primitive form of variable binding. SET_APPLY is an important looping construct without which we could not even simulate the relational algebra. The optional parameter (not shown in this example) can be of the form ":T" or ":<T". The first form indicates that the multiset should be interpreted as if only occurrences of type T that are not also occurrences of any subtype of T are in the multiset. The second form indicates that the multiset should be interpreted as if only occurrences of type T and its subtypes are in the multiset; all other elements are ignored.

Example: Let A = { { 1, 1, 2 }, { 2, 3, 4 }, { 1 } }. Then $SET\_APPLY_{INPUT-\{1\}}(A)$ = { { 1, 2}, { 2, 3, 4}, { } }.

5.2.2. Tuple Operations

The notion of a tuple in EXTRA/EXCESS is the usual notion of tuple. It consists of a fixed number of elements, each of which may be of a different type. The following operations are provided for tuples: concatenation of two tuples, creation of a tuple, and extraction of a field from a tuple (the result of this operator is not a tuple, as in relational projection, but a single field from within the tuple). A projection operator is easily simulated in terms of

these (see Appendix A).

1) **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). Notice that if either input is the empty tuple then the result is simply the other input structure. 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).

2) **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. TUP can be used, for example, along with TUP_CAT to add a field to an existing tuple.

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

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

Example: Let T = (9 18 27). Then $TUP\_EXTRACT_2(T) = 18$.

5.2.3. Array Operations

As is the case with multisets, arrays in EXTRA/EXCESS are homogeneous modulo tuple type inheritance. An EXTRA array is one-dimensional,[2] 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.

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

The operations which can be performed on arrays are: extract a subarray, concatenate two arrays, create an array, extract an element from an array, apply an algebraic expression to all elements of an array, collapse an array of arrays into an array, take the Cartesian product of two arrays, eliminate duplicates from an array, and subtract one array from another.

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 (which could be 0 if the array is empty).

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

2) **Concatenate** (ARR_CAT_{type}): 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). The subscript serves the same purpose as that for multiset –.

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.

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.

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 all occurrences in the input structure, which must be an array. The special symbol "INPUT" is used to indicate an occurrence of the input multiset; it is a primitive form of variable binding. The optional parameter (not shown in this example) can be of the form ":T" or ":<T". The first form indicates that the array should be interpreted as if only occurrences of type T that are not also occurrences of any subtype of T are in the array. The second form indicates that the array should be interpreted as if only occurrences of type T and its subtypes are in the array; all other

elements are ignored. By "ignore" here, we mean "replace by *dne*"[3].

Example: Let A = [(a b) (c d)]. Then ARR_APPLY$_{\pi_1(INPUT)}$(A) = [(a) (c)].

6) **Destroy an Array** (ARR_COLLAPSE): The ARR_COLLAPSE operator takes an array of arrays and returns the concatenation (in order) of all the member arrays.

Example: Let A = [[1, 2, 3], [2, 3, 3]]. Then ARR_COLLAPSE(A) = [1, 2, 3, 2, 3, 3].

7) **Array Difference** (ARR_DIFF$_{type}$): This operation is defined similarly to the multiset difference. The subscript is as for – (see above). The difference of two arrays A and B contains, in order, every element in A which does not also appear in B. If an element appears in A i times and in B j times, this is interpreted as follows for i,j ≥ 1: If j ≥ i then no occurrences of the element in A will appear in the result. Otherwise the last i–j occurrences of the element in A will appear in the result.

Example: Let A = [5 6 5] and B = [7 5]. Then ARR_DIFF(A,B) = [6 5].

8) **Array Duplicate Elimination** (ARR_DE): This operator removes all occurrences but the first of every element in an array.

Example: Let A = [5 6 5]. Then ARR_DE(A) = [5 6].

9) **Array Cartesian Product** (ARR_CROSS): This is essentially the same as the × operator defined for multisets. Every element in A is paired with every element in B to give an array of 2-tuples, each of which has an element of A in the first field and an element of B in the second field.

Example: Let A = [1 2] and B = [3 3]. Then ARR_CROSS(A,B) = [(1 3), (1 3), (2 3), (2 3)].

5.2.4. Reference Operations

References in EXTRA/EXCESS can be thought of as object identifiers (OIDs) which refer to objects that exist in the database independently of objects that reference them (except for their owners; see Chapter 4). Two operators are available on references: dereferencing and creation of a reference from an existing entity.

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

[3] A version of ARR_APPLY in which *dne* nulls are not placed into the result can be simulated using ARR_DE and ARR_DIFF.

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. This might be useful, for example, if we wish to create and manipulate references rather than actual data during the processing of some queries.

5.2.5. Predicates

Since algebras are functional languages, we treat predicates in a functional manner. That is, a predicate is an operation (called COMP) which returns its (unmodified) input exactly when the predicate is satisfied (true). Otherwise a null value is returned. COMP is a partial function. (Similar approaches are taken in [Osbo88, Abit88a].)

Comparisons (COMP): This operator takes a single structure as input and compares it to an arbitrary algebraic predicate 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. There are three truth-values in our predicate system: T (true), F (false), and UNK (unknown). Note that UNK is a truth value, while *unk* is a data value. If the predicate (P) evaluated against the input 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*. *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.

A predicate is defined recursively as follows:

i) For arbitrary algebraic expressions E_1 and E_2, the following are (atomic) predicates: $E_1 = E_2$, $E_1 == E_2$, $E_1 \in E_2$, $E_1 \in\in E_2$, $E_1 \subseteq E_2$, $E_1 \sqsubseteq E_2$, and $E_1 \sqsubseteq\!\!\!\!= E_2$.

ii) For two predicates P and Q, $(P \wedge Q)$ is a predicate and $(\neg P)$ is a predicate.

iii) Nothing else is a predicate.

Expressions in the predicate can contain the shorthand symbol INPUT, referring to the input structure S. The comparison symbols =, $\in$, and $\subseteq$ are given their usual mathematical interpretation. The == and $\in\in$ comparators are explained below; essentially, they represent pure symbolic equality as opposed to placing a semantic interpretation on the null constants. The $\sqsubseteq\!\!\!\!=$ symbol is an array analog to the $\in$ symbol for multisets. We say $A \sqsubseteq\!\!\!\!= B$ for an array B iff A can be obtained from B by performing an ARR_EXTRACT operation on B. The $\sqsubseteq$ symbol is an array analog to the $\subseteq$ symbol for multisets (the same symbol is used for the same purpose in [Guti89]). We say that $A \sqsubseteq B$

for two arrays A and B iff A can be obtained from B by performing a SUBARR operation on B. (Note that A may be obtainable in several different ways; i.e., it may appear more than once as a subarray of B.) Figure 5.2 shows the applicability of the comparison symbols to the different kinds of structures available in EXTRA. The comparators == and $\in\in$ have the same applicability as = and $\in$, respectively. An entry in the table means that the corresponding comparison is defined for the predicate A <op> B, where <op> is the comparison symbol found in the table and A and B are algebraic expressions or integers indicating (a component of) the input structure. Entries in parentheses indicate that the comparison symbol is redundant and merely a shorthand for a more complicated predicate. For example, two tuples are = iff each of their fields is =. The shorthand simply avoids a potentially cumbersome conjunction. The $\sqsubseteq$ and array equality comparators exist for the same reason. For multiset-multiset comparisons, equality is clearly redundant since two multisets are equal if each is a subset of the other. The subset comparator, in turn, can be simulated using lower-level comparators and more operators, but in the algebra this becomes extremely messy, while in a calculus-based language it is trivial. The COMP operation is undefined if an atomic predicate does not follow the rules of this table.

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, as shown in the following table from [Gott88], which shows the truth-values resulting from the comparison of two constants to determine either equality or multiset membership:

	set	tup	arr	ref	val
set	$\in$, ($\subseteq$), (=)		($\sqsubseteq$)		
tup	$\in$	(=)	($\sqsubseteq$)		
arr	$\in$		($\sqsubseteq$), (=), ($\sqsubseteq$)		
ref	$\in$		($\sqsubseteq$)	=	
val	$\in$		($\sqsubseteq$)		=

Figure 5.2: Comparator applicability table

=, ∈	val2	*unk*	*dne*
val1	val1=val2	UNK	F
unk	UNK	UNK	F
dne	F	F	T

Notice that *dne* = *dne*. There is only one instance of "nothing"; it is always the same.

It is also useful to pose queries which treat null values as a symbolic constant rather than as a semantic constant [Zani83, Gott88]. Thus we allow two auxiliary comparators, == and ∈∈, which force the null constants *unk* and *dne* to be treated like any other constants. That is, the above truth-table is replaced with this one:

==, ∈∈	val2	*unk*	*dne*
val1	val1=val2	F	F
unk	F	T	F
dne	F	F	T

The final truth-value of a predicate will be determined using the following truth-tables for ∧ and ¬, also derived from [Gott88]:

P1 ∧ P2	F	T	UNK
F	F	F	F
T	F	T	UNK
UNK	F	UNK	UNK

P	¬P
F	T
T	F
UNK	UNK

Examples:

i) Let A = (1 4 6 4 1). Then $COMP_E(A)$ = (1 4 6 4 1) = A, where E is the algebraic expression $TUP\_EXTRACT_2(INPUT) = TUP\_EXTRACT_4(INPUT)$. The predicate is satisfied, so the qualifying input structure is returned.

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

5.2.6. Generic Multiset/Array Operations and Extensibility

As mentioned in Chapter 4, EXTRA/EXCESS provides generic multiset functions. Such functions take a multiset or array of objects of some type (say T1) as input and produce an object of type T2 as output. T2 is either the same type as T1 or is some other type. T1 may be a type which satisfies certain constraints rather than a specific type (hence the word "generic"). The only such constraints currently provided for are those that specify a collection of operators which must be defined for type T1. Each generic function in effect defines a new operator in the algebra.

When a generic multiset function, say G, is invoked in the query language, semantic checking of the constraints on T1 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 type T2. Since T1 and T2 are EXTRA types, there are no problems in composing G with any of our other operators as defined above. G can treat null values in any way its implementor chooses.

5.3. Algebraic Query Examples

This section 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 4.6 of Chapter 4. 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] and in Section 5.6.

Example 1:

Figure 5.3 is a simple query that returns the name and salary of the 5th element of the TopTen array.

Example 2:

The query in Figure 5.4 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.

5.4. 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 is not whether it can express exactly the queries of some formal calculus but whether it can express exactly the queries of EXCESS. Furthermore, we do not address here the issues of computable queries and complexity classes, as is discussed in [Chan88].

It is, of course, crucial that any EXCESS query be expressible in the algebra. This direction of the proof is constructive, and thus it also provides a complete semantics for the EXCESS query language. The tortuousness of the EXCESS to algebra reduction is a result of EXCESS's derivation from relational query languages. It is more amenable to the standard "set-of-tuple" queries than to anything else in particular.

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--no special "tricks" are needed to bring an optimized query back into a form that represents an EXCESS query. Finally, this also means that any expressiveness results regarding the algebra can be applied to EXCESS as well.

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

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

Figure 5.3: Query Example 1

```
range of E is Employees
retrieve (E.dept.name) where E.city = "Madison"
```

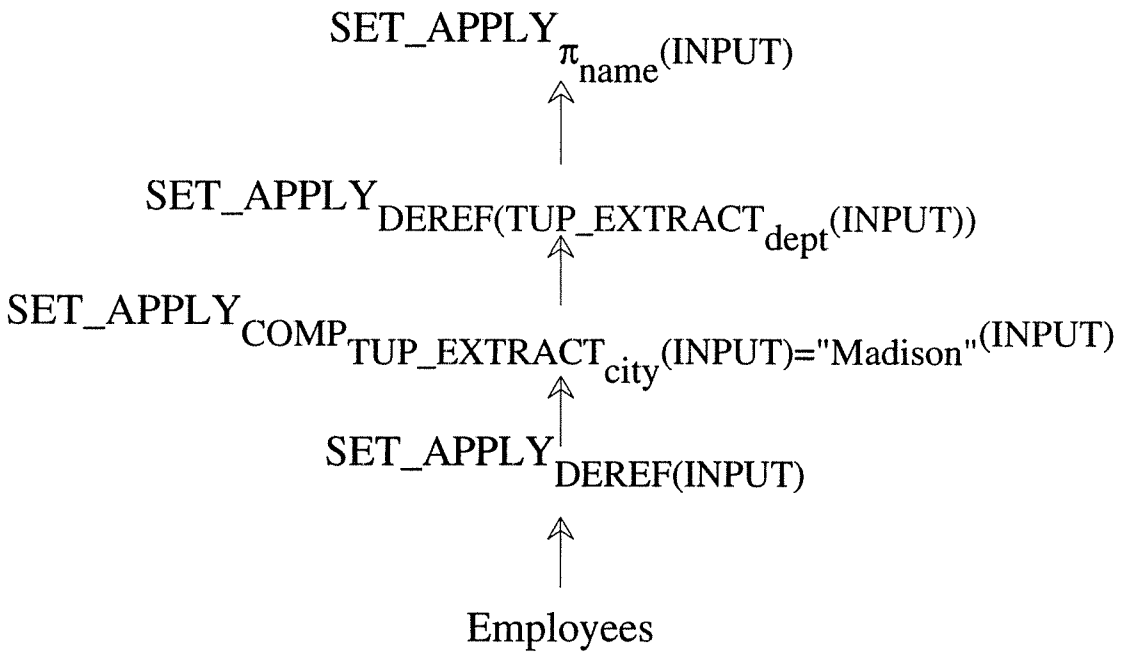

Figure 5.4: Query Example 2

Notice that the proof proceeds without an intermediate calculus--the algebra is reduced directly to the user-level query language and vice-versa. We sketch the proof here and point out its interesting features; the complete proof is in Appendix B.

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 an induction on the syntactic structure of an EXCESS query. It is interesting to note that this inductive proof forms the outline of an algorithm to translate EXCESS to an initial algebra expression for optimization; thus, it is a constructive proof. This algorithm works basically like one of the methods for translating a QUEL-like relational query into relational algebra [Kort91]: 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.

In addition to providing a construction for an initial algebra version of a query, this half of the proof quite naturally gives a complete semantics to EXCESS queries. Specifically, using the constructions of the proof, every possible EXCESS query now has a precise meaning in terms of the formally defined algebra, making the retrieval portion of the EXCESS language completely specified and unambiguous. Also, it is interesting to note that all of the algebra's operators are used in the proof, indicating that the set of operators is probably minimal.

The proof employs an interesting technique in its translation of an EXCESS query, the basic form of which is "**retrieve** (<result_list>) **from** <from_list> **where** <where_clause>". The need for this technique, described below, arises from a consideration of queries such as the following, based on the EXTRA database defined in Figure 4.6:

retrieve (Employees.name, Employees.kids.name, Employees.kids.age)

Clearly the desired result is a set of pairs. Each pair in the result corresponds to an employee, and each of these pairs contains the employee's name and a set of pairs -- one for each child of the employee. These are the semantics of the query. Thus we want a range variable to range over Employees and within each employee, a range variable over his kids. Furthermore, we want the same range variable to be used in the last two elements of the result list, to ensure that a kid's name and age are presented together. We specifically do *not* want a single range variable which ranges over just employees and, for each employee, retrieves his name, the set of all his kids' names, and the set of all his kids' ages. This would lose the relationship between names and ages.

Thus it is important to somehow correlate the two entries of the result list in the above example to ensure that the same range variable, ranging over a set of kids, is used for each of them, and that the names and ages are projected from this same range variable. The technique we introduce to account for these semantics is called *primary components*. Briefly, the primary component of an element in the result list is the largest prefix of the result that is shared with any other result in the result list. In our example, the last two results in the result list have the same primary component, "Employees.kids". The primary component of "Employees.name" is "Employees". The complete proof (presented in Appendix B) describes the details of how primary components are used to ensure the semantics

of EXCESS queries.

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 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
```

The "into" syntax is used because the inductive case assumes that smaller algebra expressions have been retrieved into top-level database objects and these named objects can be used for further processing.

Inductive Case: 1 or more operators in E

Assume that all expressions with < n operators (n ≥ 1) have EXCESS counterparts. There are 23 cases (operators) to consider. These include (among others):

- 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 × 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 EXCESS 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 expressive power are in order. First, it is capable of simulating most of the algebras mentioned in Chapter 2 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 state without proof that our algebra is incapable of expressing the powerset. (A proof would simply involve demonstrating that none of our operators can generate a result whose size is exponential in terms of the size of its inputs -- such an ability is required to compute a powerset.)

This result provides 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 (in at least some algebras) 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).

5.5. 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 4.6) 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 4.6:

```
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 5.5 ($\uplus$ 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 5.3). 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).

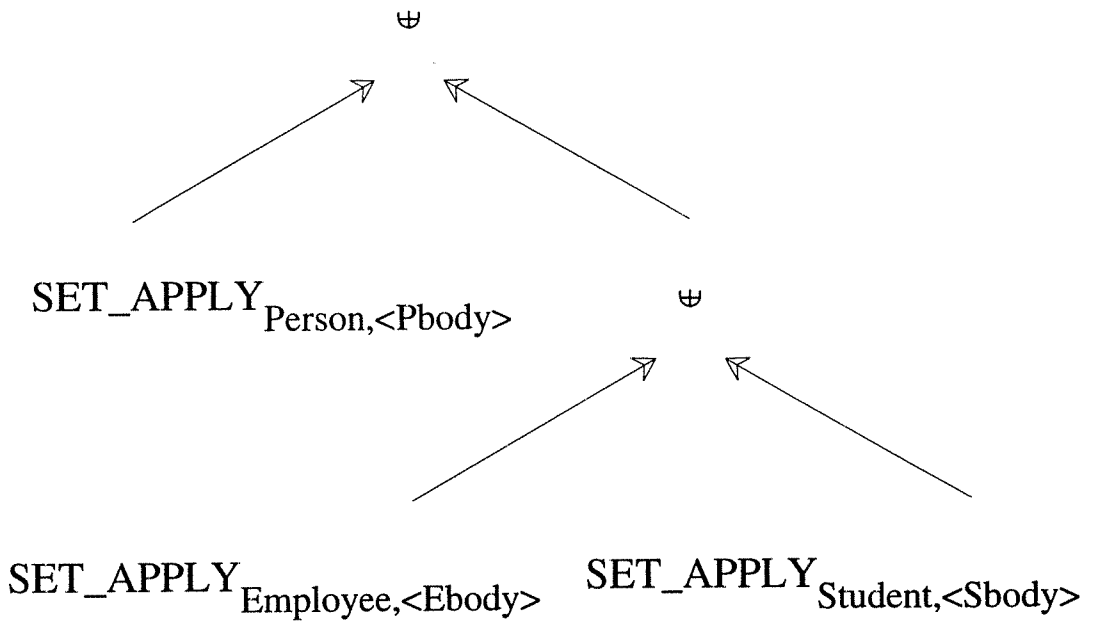

Figure 5.5: A ⊎-based approach to overridden method invocation

In EXTRA/EXCESS both of these solutions to the method overriding problem are feasible. In some cases it may be more efficient to use the first approach and in other cases the ⊎-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 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 ⊎-based approach so that the most expensive part of the query can be optimized at compile time. The ⊎-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 Ψ-based approach disappears.

5.6. 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 list of the rules is in Appendix A, 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 Chapter 2), but here we emphasize the original rules. Each example presents an EXCESS query over the database of Figure 4.6 and a series of algebraic representations of that query, in a manner similar to that of Section 5.3. 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 5.6 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 Appendix A), then groups the result (some details are omitted form this node), performs the final projection (details omitted here also), and eliminates duplicates. Figure 5.7 shows the application of a rule that pushes DE ahead of grouping; 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 5.8 we create another alternative by pushing the DE and relational π past the "join" node. This

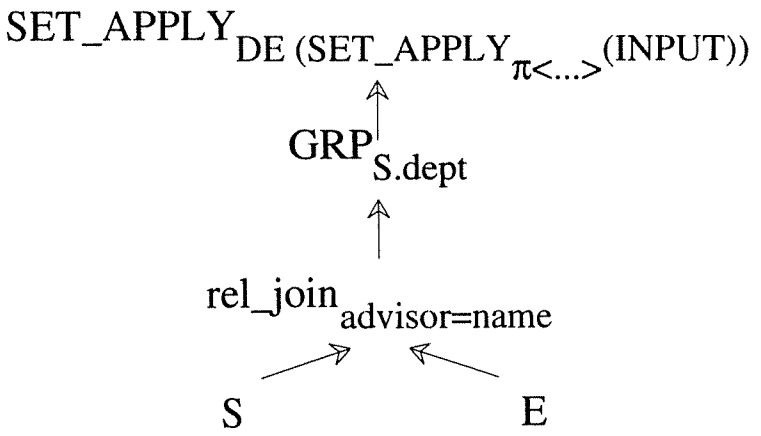

Figure 5.6: Ex. 1, initial

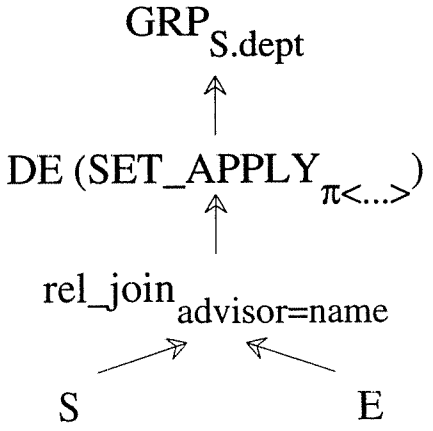

Figure 5.7: Ex. 1, 1st transformation

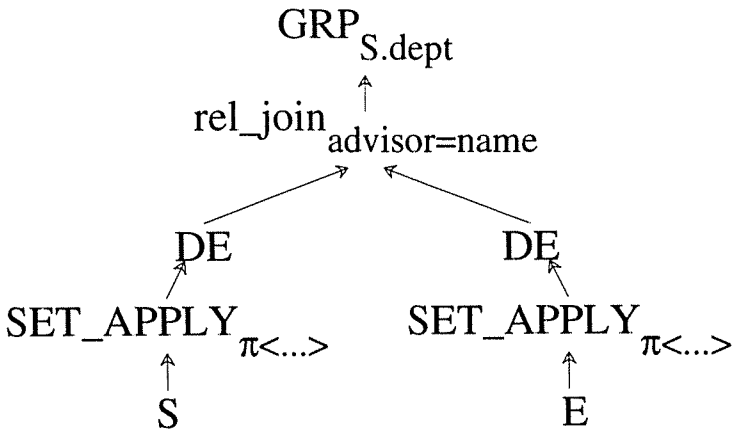

Figure 5.8: Ex. 1, 2nd transformation

results in DE operating on |S|+|E| occurrences rather than (in the worst case) |S|*|E| occurrences. The DE and π have been separated into two nodes in Figure 5.8 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
```

An algebraic representation appears in Figure 5.9. 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 5.9 omit some non-essential details. Figure 5.10 shows one way of optimizing the query: successive SET_APPLYs are collapsed, twice, to get this query. First we collapse the top two nodes of the query in Figure 5.9 into one node to eliminate one scan of the set, then the query of Figure 5.10 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.

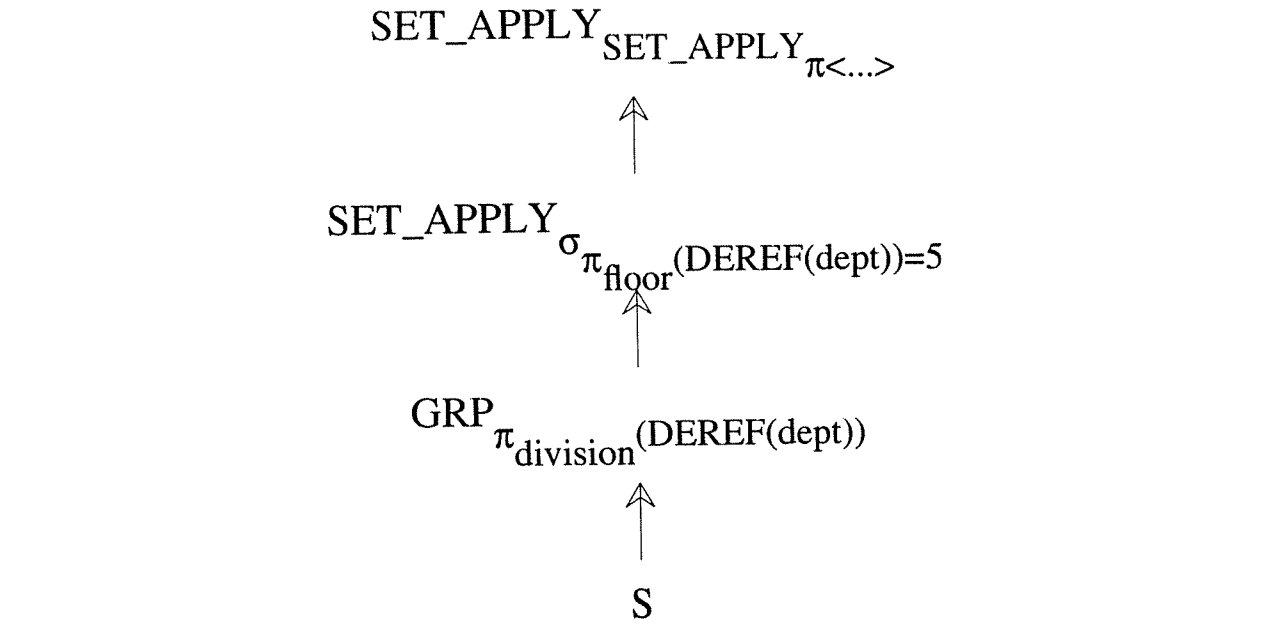

Figure 5.9: Ex. 2, initial

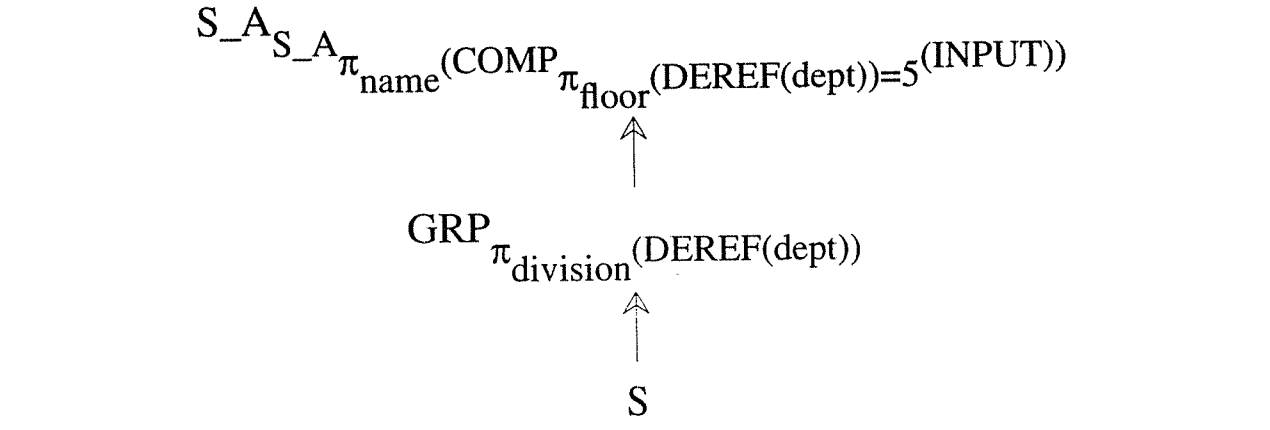

Figure 5.10: Ex. 2, 1st transformation (S_A is SET_APPLY)

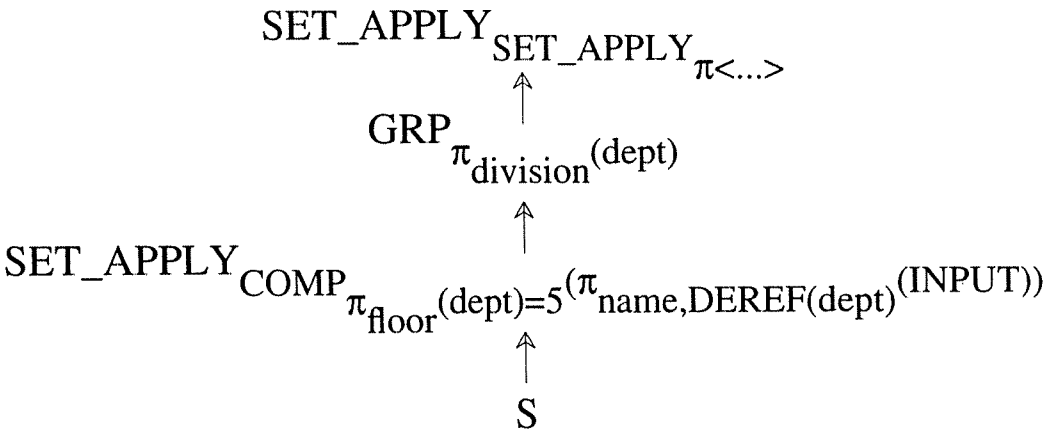

Figure 5.11: Ex. 2, alternative 1st transformation

Another way of optimizing this query is presented in Figure 5.11, which is derived directly from Figure 5.9. Two rules are used to obtain this version of the query. 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 5.11 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 (though 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 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.

5.7. Comments on a Partial Implementation

EXTRA/EXCESS was partially implemented using the EXODUS extensible DBMS toolkit [Care88a] at the University of Wisconsin-Madison. A complete parser and implementation methods for some of the algebraic operators were implemented, as was a partially functional DML processor. For the purposes of this thesis, we are most interested in the query processing component. Query processing is designed to operate as follows: 1) Translate the parse tree into an algebraic operator tree; 2) Optimize this tree using an optimizer generated by the EXODUS optimizer generator [Grae87]; 3) Convert the optimized plan tree into a storable, interpretable plan tree; 4)

Interpret the plan tree. Here we outline what was completed and present a brief critique of the EXODUS optimizer generator.

An algorithm for step (1) was implemented in the E programming language [Rich87]. It is based on the equipollence proof presented in Appendix B. Steps (3) and (4) were partially implemented. The difficult part of the implementation is the production of a working optimizer using the optimizer generator. This is problematic mostly due to the difficulty of implementing auxiliary data structures that the generated optimizer needs in order to perform its function. The generator provides no support for the design of data structures and algorithms to manage non-data arguments to algebraic operators. For example, all code to manage predicates, test their applicability, translate them when a query is being transformed using an algebraic identity, etc. must be provided by the implementor. In a model as complex as EXTRA/EXCESS, this is a massive undertaking.

Another difficulty with the optimizer generator was its somewhat restrictive notion of operators. Specifically, it makes no provision for second-order constructs such as the EXCESS algebra's SET_APPLY operator, which can take an arbitrarily complex query as a parameter to be applied to its input. This parameter is not itself data that is operated upon by the SET_APPLY operation. This distinction is impossible to make in the optimizer generator's formalisms. A less important criticism is that it would be useful to be able to specify that the same rule (e.g. associativity) applies to each of a collection of operators.

However, an input file for the optimizer generator was written anyway, and an optimizer was generated using this file. In order to do this, the optimizer generator had to be updated in several ways to accommodate the use of E (instead of just C) and to allow it to model some of the constructs of EXCESS effectively. The input file is approximately 1300 lines long and expresses 150 rules. This should be compared to the number of rules needed for a similarly-featured relational query optimizer supporting duplicates and duplicate elimination (which most commercial relational systems do). Such an optimizer would need approximately forty rules.

If we consider that over 50 of these rules apply only when arrays are involved in the query, we see that we have not paid too great a price for the ability to optimize complex queries over complex objects. In other words, for any particular query, a large percentage of the rules will not apply simply because they apply to the wrong type constructor. Furthermore, within the rules that apply to a type constructor, a large percentage of the rules will not apply. For example, if a query does not have a "unique" clause, the 12 rules involving the DE operator will never apply. For select-project-join style queries (i.e. those not involving binary set operations such as intersection), the

search space shrinks further to 30 to 40 rules. This means that the search space of alternative plans will not be much larger than it is for a standard relational optimizer. There are a lot of rules because the system is more full-featured, but the number of rules that can be applied at any given time is similar to that of a relational optimizer. Thus we conclude that the optimization problem itself is not a major issue, but the engineering task of designing data structures and cost functions for use by the optimizer is an issue (but is outside the scope of this thesis). We further speculate that the complexity of EXCESS ensures that the same problems of data structures and algorithms for optimizer support would exist no matter what the design of the algebra or query execution paradigm.

CHAPTER 6

THE AQUA ALGEBRA

The goals of AQUA were discussed in Chapters 1 and 3. We first discuss the relationship of AQUA to EXCESS.

There are several useful object-oriented features that EXTRA and EXCESS were not designed to model. First, an abstract notion of object identity is missing from EXCESS. The **ref** construct is a very physical notion of pointers. Furthermore, in EXCESS, not everything is an object, and AQUA is intended to model object-oriented systems. Second, and related to the first point, a goal of AQUA is to permit ways of expressing data semantics. EXCESS is a purely structural model, and provides little support for semantic specification. AQUA's notion of identity versus value is one example of semantics; the meanings and properties of methods, etc. are another. Another goal of AQUA is the modelling of more complex structured data such as trees, unions, N-dimensional arrays, etc. These structures are not supported in EXCESS. Finally, EXCESS queries have limited expressive power. AQUA is designed to model certain recursive queries and other powerful querying facilities such as accumulation [Vanc92].

The AQUA model and algebra are extensions of EXTRA and EXCESS. Any EXCESS algebra query can be expressed in AQUA and can look nearly identical to its EXCESS counterpart, as many of the EXCESS operators have been almost directly incorporated into AQUA. An important exception is the class of EXCESS queries involving the REF and DEREF operators. Object identity in AQUA is a semantic notion, not a structural notion. Thus such EXCESS queries must make use of a facility in AQUA that allows a query to change the semantics associated with an object. In this manner AQUA can simulate the REF and DEREF operators of EXCESS.

In the following subsections we describe the AQUA model and type system, AQUA's approach to modeling abstraction, its support for multiple definitions of equality, and operators and techniques for processing queries over tree-structured objects.

6.1. The AQUA Model and Type System

AQUA [Leun93] is based on an object-oriented data model. All objects have identity, and these identities allow us to distinguish between objects using identity-based equality. AQUA is closed in the sense that all of its operators return objects that are defined in the model.

The type system in AQUA is a 3-tiered system. New types are created using the type constructors and existing types. A type is defined recursively as (*name*, *hier*, $C(m_1 : t_1, ..., m_n : t_n)$) where m_i is a name, t_1, ..., t_n are previously defined types, and *C* belongs to the set of type constructors defined in the model. *Integer, boolean, float,* and *string* are the base types provided. *Hier* is the set of immediate supertypes of *name*. Type equivalence is by name. Also, every type provides a set of interface functions, since everything in AQUA is an object. Normally, these interface functions are automatically defined to be the interface available on the outermost type constructor of the type. For example, a tuple type automatically has projection (among other things) defined on it. To extend or eliminate these automatically provided functions, we provide a type constructor for abstraction, described in Section 6.2.

The notion of a type in AQUA is purely syntactic. However, there are important notions which cannot be captured syntactically, but which are useful in data modeling or in query optimization. We therefore add a second level to our system by requiring each type to have one or more *semantics*. The semantics of a type might loosely be thought of as axioms (in the style of Larch [Gutt85]) that describe properties of the operations on a type. The particular language used for describing semantics is a topic of future research. At the bottom of this two layered system, there is an additional layer that provides for multiple *implementations* (at least one) for each semantics.

This allows the database or the user to select the most desirable implementation at any point in the lifetime of the object. Thus, an object really is an instance of a particular implementation of a particular semantics of a type. The system currently defines two possible semantics for objects, mutable and immutable. Objects with mutable semantics may update their state, while objects with immutable semantics may not.

Making a distinction between the *type* of an object and the *semantics* associated with it allows us to view all objects as abstract data types by moving the traditional distinction between objects and values into the semantics layer. So, the traditional notion of values is supported via immutable semantics for objects.

Another example of the use of semantics is the declaration that certain operations are commutative. A query optimizer could then make use of such information when optimizing a query. The semantics describing commuta-

tivity for deques might be written as *notEmpty* (q) => *enque* (*deque* (q), x) = *deque* (*enque* (q, x)). Commutativity axioms for sets might include: *select* (p2) (*select* (p1) (s)) = *select* (p1) (*select* (p2) (s)), etc. Notice that in this notation the first set of parentheses following *select* contains the predicate to be applied to all elements of the set contained in the second set of parentheses. This is the standard AQUA mechanism for distinguishing between the actual data being input to an operator (in the final set of parentheses) and any other parameters needed by the operator (in the initial sets of parentheses).

One consequence of this approach is that type can be determined statically (at compile-time), but semantics and implementation can only be determined at run time.

One of the primary goals of the algebra and the model has been to support a large number of bulk types in a uniform manner. A type constructor is a metatype which defines a family of types. The *Set* type constructor, for example, defines the family of types that includes *Set[integer], Set[Department]*, etc. AQUA provides the following type constructors: *sets, multisets, tuples, unions, functions, cells, lists, graphs,* and *trees*. The algebra also supports the *abs* constructor that allows creation of new abstract types. The operators of the AQUA algebra are the methods on the AQUA type constructors.

All AQUA objects are unique, and this uniqueness can be detected by the user. AQUA objects have identity, and can be distinguished using equality that is based on identity. We do not specify the implementation of identity, to prevent a fixation on object identifiers. The most important point is that objects are unique, and that given two objects, we can determine if they are the same object or not. Note that two objects of the same type but with different semantics will never be identical. Note also that immutable semantics imply that an object is its own identity, since we can never change an immutable object. E.g., there is only one immutable set containing objects x, y, and z. This set is always equal to itself. There may be many other mutable sets containing the same objects. We say more about this in Section 6.3.

6.2. Abstraction in AQUA

Abstraction is a common feature in many object-oriented models, and AQUA is no exception. Every type is essentially an abstract data type -- however, the base types and all types built using the usual type constructors have a fixed set of standard operations available on them (e.g. **join** and **union** for sets). As mentioned in [Care88b] and [Rowe87], it is often desirable to be able to add other methods to such types. In addition, it is frequently desirable

to specify a type whose representation is completely unknown to the data model of the DBMS.

To solve both of these problems cleanly, while maintaining a uniform type system, we introduce the abstraction type constructor. *Abs*[T, f_1:T_f_1, ..., f_n:T_f_n] is a new type and has a collection of named functions, which form the interface of the abstraction. Each named function (or method) f_i has an associated type (signature) T_f_i. The first parameter to *abs* is an optional type. If present, it signifies the representation type of the abstraction. If absent, it signifies that the representation of the abstraction type is completely unknown. We will illustrate this with examples below.

Abs is a type constructor just like any other -- its use defines a new type and it has operations defined on it. For abstraction, there is only one operation, called **invoke**. It invokes a method on an object whose outermost type constructor is *abs*. In what follows, we use A.B as a shorthand for **invoke** (A, B), meaning that the method named B is being invoked on object A.

We will use two schema definitions to show the use of the abstraction type constructor. The Person1 type is an abstract data type which uses a tuple as its representation, and which supplies some additional methods to access and update its information:

```
type Person1 = abs [tuple [name:string, address:string, birthdate:int]
        set_address (string);
        get_age () -> int
]
```

Notice that we omit the implicit parameter to each function that represents the object being operated on. The operations available on objects of type Person1 are all the operations available on tuples (e.g. concatenate, project) plus the set_address and get_age operations. This is similar to the ability of EXTRA/EXCESS to define new functions and procedures on EXTRA types, but here we make no assumptions about how the user-defined methods are implemented. Person1 is an ADT whose representation is known (and can thus be optimized) and which defines some new operations in addition to those defined on its representation.

Now consider another abstraction type, Person2, defined as follows:

```
type Person2 = abs [
        get_name () -> string;
        get_address () -> string;
        get_birthdate () -> int;
        set_address (string);
        get_age () -> int
]
```

Since no representation type is provided for Person2, we do not know anything about its structure or implementation. We do know that it has methods to retrieve a name, address, birthdate, and age, and to change an address. Thus it is functionally equivalent to the Person1 type, except that the methods implicitly present in Person1 (get_name, get_address, and get_birthdate) are simple field extractions from tuples in the Person1 type but could have any implementation in the Person2 type. Furthermore, Person1 objects may be directly concatenated with other tuples. For a Person2 object, functions must be invoked and a tuple construction operator must be employed to make it into a tuple before any concatenation can occur.

As a specific example of the syntax, consider two objects P1 and P2, of types Person1 and Person2 respectively. An AQUA expression to retrieve the age of P1 is:

P1.get_age()

The same syntax will retrieve the age of P2. Now consider an AQUA expression to retrieve the name of P1:

P1.**select_field**(name)

where "select_field" is a built-in tuple operation that retrieves an attribute from inside a tuple. (The above expression is actually shorthand for "P1.**select_field**(P1, name)".) To retrieve the name of P2 we use the following expression:

P2.get_name()

The distinction is that the query optimizer will have more knowledge about how to get the name of P1 than how to get the name of P2.

Again, a major goal of AQUA is generality, and with this approach we can model a variety of forms of abstraction found in the literature, without creating an entirely new mechanism for ADTs. The entire reason for introducing abstraction as a type constructor is simply to keep the model and algebra as uniform and flexible as possible. In addition, specifying ADTs as regular types in the AQUA type system will allow us to give them appropriate semantics using the semantic definition facilities of AQUA. Thus the clear distinction between syntax and semantics

seems to be a powerful notion, allowing us to say what needs to be said about a type within a single formalism, whether the type is built-in, is constructed with the type constructors, or is an ADT.

6.3. Definitions of Equality

Some types may have more than one useful notion of equality. Every type has a default notion of equality, which is identity. The built-in primitive types (*integer, float, boolean*, and *string*) have the standard definitions for equality. When a user defines a new type, he may also define a notion of equality for that type. Meaningful user-defined equalities should induce an equivalence relation over all the instances of that type. In the algebra, the use of the "=" symbol always involves only the default equality. All other equalities will be user-defined binary predicates, e.g. *my_eq(a,b)*.

Equality is essential to the definition of some operators, like set union. The AQUA **union** operator is a standard set-theoretic union. **Union** needs to eliminate duplicates in the result. A result that is a set of people could be a true ''set'' in the sense of having no two identical objects, but could have duplicates in the sense of two distinct objects being equal according to the user's definition of equality. For example, a user defining a person type may also define an equality predicate which tests for the equality of the social security numbers of a pair of people, assuming that the social security number is an attribute of the *Person* type. Thus two distinct *Person* objects may actually have the same social security number, but for the user's purposes they should be treated as being the same person. The utility of such an ability has also been pointed out in [Atki91].

As other examples of the usefulness of type-specific equalities, we can of course point to shallow and deep equality. There are other less obvious applications, however. For example, one might wish to define an equality predicate for strings that is case-insensitive (the default equality would, of course, distinguish between upper- and lower-case letters). Another example involves the semantics of object attributes. We may wish to say that two tuple-structured objects are equal if all of their *components* are equal, regardless of the values of their *properties*. As a specific example, a motor vehicle database might store cars as tuples with fields representing engine, frame, owner, and color. The engine and frame are components of the car, while the owner and color are simply attributes describing the car's current condition. Thus it is reasonable to consider two cars equal if they have the same frame and the same engine. It might also be useful to use type-specific equalities to allow for tolerances in real number comparisons -- i.e., to check only a certain number of significant digits. This relation, however, is not transitive, and

thus not an equivalence relation. It does seem useful, however, and it may cause a re-examination of the equivalence relation requirement in the future.

Now let us consider sets of objects of some type, and suppose that we wish to compute the **union** of two sets whose elements are of type *Person*. The definition of **union** naturally removes duplicates when computing its result. Identity is the default equality for set elements (and everything else) in AQUA. Thus the result of the union will contain each uniquely identifiable person exactly once. This may be the desired result. However, a user may wish to see only those persons from either set with unique social security numbers, to eliminate any duplication based on incorrect data entry, etc. In that case the **union** operation should compute its result by using this user-defined equality, rather than identity, to determine what belongs in the result.

This suggests that we might wish to parameterize the **union** operation by the form of equality it uses. So, for example, to **union** two sets of *Person* objects, PS1 and PS2, using equal social security numbers as the equality criterion, we might write this algebra expression:

union (essn) (PS1, PS2)

where "essn" is a predefined binary predicate on *Person* objects, returning true if the two *Person*s have the same social security number. This solution works, but there is a simpler, more elegant, more concise, and more illuminating way of approaching the problem.

First, we must realize that other operations of the AQUA algebra might need to make use of such a parameter -- the set **diff**erence and set **intersect**ion operations being obvious examples. So these would need to be parameterized also. In addition, any new operators defined in terms of any of these would need to be parameterized. In general, of course, only operators in which equality comparisons between two objects implicitly occur give us this problem. AQUA's **join** operators have an explicit predicate parameter already, so they pose no problem in this regard. **Group**ing is not a problem as it groups into equivalence classes based on the result of a unary algebraic expression applied to the elements of the set, as in EXCESS. The following AQUA operators use equality implicitly: **union, diff, intersect, nest** (the standard NF^2 nesting operation), and **LFP** (a general least fixpoint operation). The AQUA **nest** operator, however, is not a primitive operator. It is easily defined in terms of **join** and some other operations, so we will not discuss it further.

Second, we observe that isolating the functionality of these various forms of equality into one place in the algebra might help us to reason about them more effectively. Also, the use of equality as a parameter to operators may restrict and complicate the algebraic transformations that can be defined over those operators. These are the motivations for the approach we now describe.

If a user wishes to impose a non-default equality (i.e., an equality other than identity) on a set, he can first perform whatever operations he desires (these will use the default equality). When he needs to use the contents of the set in a fashion that is sensitive to some other notion of equality, he can apply the **dup_elim** operator. This operator takes an equality (in the form of a binary predicate) as a parameter, and eliminates duplicates under that equality. This works because a user defined equality is weaker than the default equality (anything that is a duplicate under the default equality is a duplicate under the user equality, but not vice versa). This allows us to put off using the weaker equality until an equality sensitive operator is needed. The **dup_elim** operator is defined as follows:

$$\textbf{dup\_elim}\ (eq)\ (S) = R \subseteq S \mid \forall\, x,y \in R\ (\neg eq(x,y) \vee x=y) \wedge \forall\, x \in S\ (\exists\, y \in R\ (eq(x,y)))$$

This approach allows any equality to be used at any point during query processing without compromising our notion of ''set''. The operators remain defined in the abstract and we are assured that they can handle any kind of equality that may arise, including anything the writer of a query might wish to pass in as a parameter. Thus the AQUA **union, diff, intersect**, and **LFP** operators do not have an equality parameter. However, we will now present some examples and describe precisely how the equality-parameterized versions of these operators can be defined in terms of the non-parameterized versions, **dup_elim**, and other operators.

Consider the union of two sets A = { (1, a), (2, b) } and B = { (2, a), (2, b) }. Assume we want **union** (A, B) using a notion of equality that says two elements are equal if their fields are pairwise equal, so that the (2,b) in A and the (2,b) in B are equal, but no other pairs are. The result will then have three elements: { (1, a), (2, b), (2, a) }.

Now suppose we want **union** (A, B) using a notion of equality that says two elements are equal if their second fields are equal. The (1,a) in A and the (2,a) in B are now also equal. The required result is then { (1, a), (2, b) } or { (2, a), (2, b) }. Either result is correct since the equality only examines the second element of each tuple. Notice that our definition of **dup_elim** is careful to eliminate the possibility of { (5, a), (2, b) } being a correct answer: each element of the result of **dup_elim** must actually come from the input set, even though (in this case) the con-

union (eq) (A, B)	$\equiv$	**dup_elim** (eq) (**union** (A, B))
LFP (f, eq) (A)	$\equiv$	**dup_elim** (eq) (**LFP** (f) (A))
diff (eq) (A, B)	$\equiv$	**diff** (A, **select** (λ(a)**exists**(λ(b)eq(a,b)) (B)) (A))
intersect (eq) (A, B)	$\equiv$	**diff** (A, **diff** (eq) (A, B))

Figure 6.1: *eq*-parameterized operations in AQUA

tents of the first field are immaterial.

Figure 6.1 contains the definition of equality-parameterized **union** in terms of the primitive operators. It also defines other equality-parameterizable operators in terms of the primitives. Note that there is no simple formulation of **diff** (and thus of intersection) in terms of **dup_elim**. It uses some other AQUA operators, namely **exists** (the standard predicate calculus operation) and **select** (an operation which removes from a set all elements not satisfying a predicate). Intuitively, this construction for **diff** is necessary because the result of the difference does not eliminate all *eq*-duplicates from A -- it only eliminates those that were equal to some element of B. This is the correct semantics of set difference; it removes from A all and only those things that are equal to something in B. AQUA's **LFP** operator uses a standard union-based definition, so as with **union** we can apply **dup_elim** after the computation of the least fixpoint.

The **dup_elim** operator may be applied at any time, so any AQUA expression requiring a set reflecting some non-default definition of equality can operate on such a set by first applying **dup_elim**. The operators of Figure 6.1 were special cases because their definitions implicitly make use of a (potentially non-default) definition of equality.

6.3.1. Duplicate Elimination and Non-Determinism

We now make the interesting observation that **dup_elim** is not a primitive operator in AQUA. **Dup_elim** can be defined in terms of some other AQUA operators, including **group** and **choose**. If eq is a binary predicate, and S is a set,

```
dup_elim (eq) (A) =
        apply (choose)
                (group (λ (a) select (λ (b) eq(a,b)) (A)) (A))
```

First, observe that the parameters to operations that apply functions, such as **group**, **apply**, and **select**, are expressed

using λ-calculus notation to indicate the variable bindings. The λ expressions are used only for that purpose. **Group** (f) (S) groups the elements of S into equivalence classes by using the f parameter, returning a set of sets, just as in the EXCESS algebra's GRP operator. The **apply** operator is quite similar to the EXCESS SET_APPLY operator. **Choose** takes a set and nondeterministically selects one element of the set as its result. The **dup_elim**ed set has no duplicates with respect to the new equality, and it also has no duplicates according to identity.

This definition of **dup_elim** illustrates the fundamental nature of grouping and non-determinism in object-oriented systems which support more than one form of equality. It also demonstrates that only one form of equality is needed in the algebra, that of object identity. All other uses of all other forms of equality can be captured using *dup_elim* and other algebra primitives, and the only meaning of the "=" symbol in the algebra is pure object identity. This is an interesting result when contrasted with the EXCESS algebra, which also has one form of equality, value equality. EXCESS is a value-based system that supports objects rather than an object-based system that supports values. In either approach, however, only one form of equality is ever strictly necessary.

Since both the EXCESS and AQUA approaches can simulate each other, the question as to which approach is better is partially a matter of taste. The EXCESS approach becomes tedious when a DBA wishes all database entities to have identity, for example. The "ref" constructor must be used everywhere. But if a DBA wishes to use value equality as a default, the effort required in AQUA to provide all the appropriate equality predicates might be substantial. The other consideration is that by choosing identity instead of value as the default (as in AQUA), we have a fairly abstract notion of equality. By itself, identity says nothing about what form these identities can take, etc., and thus does not force us into using standard definitions of equality on any of the types in the database, including built-in base types. This extra level of abstraction is of use in reasoning about the algebra and its various versions of equality.

Duplicate elimination based on anything but identity, though, is somewhat unsettling in that the same query may give different results on different executions, even if the database has not changed between executions. See the **union** examples above for an illustration of this phenomenon. Since object identities are not supposed to be visible to users, however, two executions that return the same result modulo object identities are not necessarily a problem. But in our *Person* type examples above, where social security number was declared as the basis for uniqueness, any data other than the social security numbers will be potentially different on different executions of the query.

The consequences of this are that users must be very careful in defining the equalities to be used for a type. Any information that is not used in determining the equality may vary from one run of a **dup_elim** query to the next. Our current solution to this is that any user posing such a query will have to be prepared for possibly inconsistent results. An alternative would be to somehow mask any data that could be inconsistent. In other words, once you ask (for example) for duplicate elimination on a set of tuples based on the values of their first two fields, any other fields should be invisible because they may be inconsistent between executions of the query. This consideration also occurs during query processing. Any subsequent action by the query should not make use of any other fields of the tuple, as this will lead to nondeterministic results as well.

It is also interesting to note that there is a broad spectrum of equalities, with object identity (the default) at one end. Under identity, of course, no two distinct objects are equal. At the other extreme would be an equality that says *any* two objects of a given type are equal -- for two objects A and B, "A = B" is always true. This definition of equality is admittedly not very useful. In between are the more useful user-defined equalities. For any tuple type, for example, there are at least as many possible user-defined equalities as there are combinations of its fields. Some of these equalities are weaker than others (identity being the strongest form of equality), forming a lattice of possible equalities on every type.

6.3.2. Other Approaches

In the context of collections of objects of some type, there are two basic places in which an equality parameter could be placed to indicate which equality to use in determining uniqueness within the collection:

- Equality could be a constraint associated with a collection type
- Equality could be a constraint associated with an instance of a collection type

The first option means that equality is a parameter of the collection type in the same way that the member type is a parameter. For example, we could define the type T1 = Set[Person] with equality based on name and the type T2 = Set[Person] with equality based on social security number as two different types. A disadvantage of this approach is that constraining a set instance to use a different equality notion requires changing the type of that set. Thus, type casting would be necessary in order to union two sets of types T1 and T2. Another disadvantage is the proliferation of types.

The AQUA approach is a variation of the second option listed above. Associating equality with instances of a collection type allows a set to be constrained by different equalities at different points in time. One implication of this is that binary operations such as union might need to be defined to handle any possible combination of equalities. For example, if the Person type has three different forms of user-defined equality, used by different users for different purposes, one might need nine **union** operations in order to be able to perform any possible **union** of Person sets. This is the motivation for parameterizing operations such as set **union** with an equality parameter. The equality is only "visible" when operations (such as printing, **union**, etc.) detect duplication. Thus equality can be a parameter of such operations.

In AQUA we took this reasoning one step farther by demonstrating that such a parameterization is needed in only one place in the algebra, a duplicate elimination operator that transforms a set so that it reflects a certain notion of equality. Even more fundamentally, we showed that this operator can be defined in terms of **group** and **choose**.

6.4. Selection Queries on Trees

This section describes the fundamentals of processing selection queries on trees and sets of trees (forests). The first subsection is an introduction to AQUA's treatment of ordered types in general[1] (lists, trees, and graphs). Section 6.4.2 describes some useful queries on trees and an extension to regular expressions that allows us to specify patterns in trees. Some algebraic operators to model such queries are described in Section 6.4.3. The last section discusses the utility of various forms of indexing over trees and forests and the applicability of some popular tree-matching algorithms to the queries that we have discussed.

6.4.1. Ordered Types in AQUA

Lately there has been a lot of interest in bulk types like lists, trees, and graphs that are not supported by traditional data models and query algebras. This interest is fueled by the fact that much data in the scientific domain is inherently ordered [Fren90, Fren91, Land91]. Scientific applications have a need to store ordered types such as time-series data and genome sequences, and textual databases often store information that is structured as a tree. These applications store huge volumes of data and must locate information from these structures very efficiently. They thus require database support for ordered data structures like lists, trees, and graphs.

[1] Parts of this section summarize [Subr93].

Viewing lists, trees, and graphs as types in their own right allows us to utilize their specialized properties for query optimization and gives us more flexibility in defining operations over them. This distinction might also help in other related areas like specialized storage structures to speed access and specialized index structures for querying. For example, a tree structured collection of objects could suggest a way of clustering the objects on disk. Also, with trees as a type constructor, there is no need for any explicit attributes to indicate the relative ordering among the nodes of the tree. Finally, for many applications, a tree is a natural way of describing the data. We might want to store information about a family's genealogy as a tree, for example.

Graphs are used as the fundamental building block out of which the other bulk types are derived. Sequences (lists) and trees are viewed as specialized graphs. Duplicates are introduced into these graph structures through a notion of a *cell* type (described below). A graph G = (N, E) with node set N and edge set E can be used to express an order over the node set N. We say that for any two nodes $a,b \in N$, $a \leq b$ iff $(a,b) \in T$, where T is the set of edges in the transitive closure of the graph G. Such an order can be restricted further to produce a tree or a list. All graphs in AQUA are directed.

A goal of this work has been to define the operators on all the bulk types so that they are consistent with each other. The operations on a more specific version of a bulk type follow from the operations on a more general version. For example, **select** for a tree is the same as **select** for a graph when the tree is viewed as a graph. A graph with an empty edge set is essentially a set. Thus, this kind of degenerate graph behaves in all ways like a set. Identities that apply to sets apply to graphs with no edges as well.

With this view, all the operators for graphs neatly transform into the corresponding operators for sets. For example, **union** and **intersection** on graphs with empty edge sets are similar to these operations on sets. Similarly, **apply** and **select** behave the same way for graphs with an empty edge set, as for sets. This makes the addition of ordered types into the model seamless and consistent with the other bulk types.

All the ordered types are defined as a set of nodes N, and a set of edges E. However, there are cases when there is a need to allow duplicates. Duplicating nodes could possibly be handled by allowing the node set N to be a multiset, but that would not allow for a distinction to be made between the edges of two identical nodes. Thus, we introduce the concept of a cell. A cell can be thought of as a wrapper around an object that allows us to distinguish between two nodes containing the same object. With cells, we could have the same object represented as two different nodes, as the identity of the cells provides the uniqueness.

Cells have two operators: *Cell* (a) creates a cell containing the object a. *Cell_content* (c) returns the object contained in cell c.

We now focus the discussion more on trees in particular. The AQUA model supports two kinds of trees, ordered and unordered. Ordered trees are trees in which there is an order between the children of a node and unordered trees assume that there is no explicit order between the children. Ordered trees are defined as a set N of nodes and a list E of directed edges (as opposed to a set of directed edges for unordered trees). The relative ordering among the edges from the parent node to the child node in the list E determines the order of the child nodes. We do not discuss unordered trees in this thesis, but most of the results in the remainder of this chapter will carry over to that case.

The type constructor for trees is Tree [T, int], where T is some type. A tree of type T consists of nodes of type Cell[T] and the edges between these nodes. We use cells since it allows us to handle duplicates in a consistent manner. Duplicates in trees and lists present a problem when dealing with operators that could possibly map two or more nodes of the original tree onto the same object. In such a scenario, preserving all the associated edges might violate the tree (or list) structure. As a result, we adopt the "cell" structure to avoid duplicate nodes. For example, in Figure 6.2 we show the result of applying an expression to each node of a tree using the **apply** operator. This particular query generates duplicate node contents, and without the Cell constructor the result would have only one

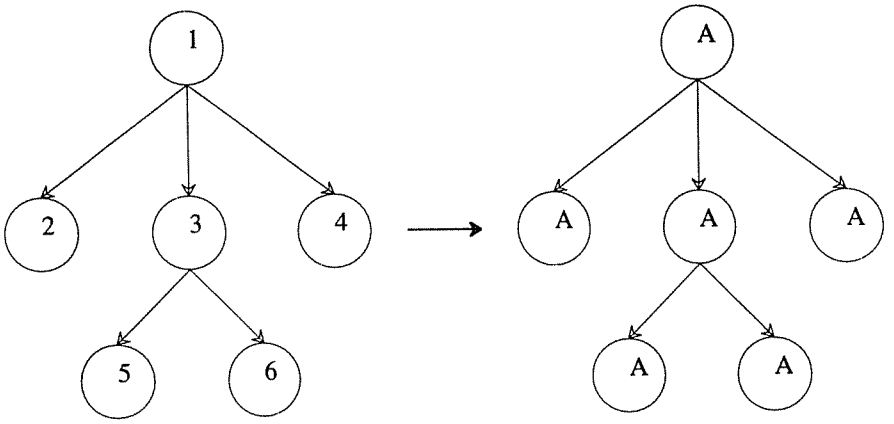

Figure 6.2: Apply (λ(x)A) (T)

node, losing the structure of the original tree. Using a multiset to represent the nodes would lose information about the connections present in the tree.

The "int" parameter to the Tree constructor is optional. It must be non-negative. If present, it indicates the maximum number of children any node in a tree of this type can have. If absent, the number of children is unbounded. If it is 0, the tree is effectively a set. If it is 1, the tree is effectively a list.

We now briefly describe the relevant operators on trees. Most of the tree operators have been derived from the corresponding graph operators (see [Subr93]). The basic tree operators of AQUA that are of interest here are: **sources** (T), **sinks** (T), **im_ancestor** (T, x), **im_descendent** (T, x), **select** (p) (T), **sub_select** (T2) (T1), **apply** (f) (T), and **find_path** (T, x). The other tree operators are described in detail in [Subr93].

Sources (T) returns the set of nodes of T that have no incoming edges. This set will contain only the root of T. **Sinks** (T) returns the set of nodes of T that have no outgoing edges (i.e., the leaves of T). **Im_ancestor** (T, x) returns the set of nodes in T that have edges to x (i.e., the parent of x in T). **Im_descendent** (T, x) returns the set of nodes in T that are children of x.

Select (p) (T) selects all nodes of T that satisfy the predicate p. See [Subr93] for a discussion of how edges are preserved in the result tree. This selection operator is not of primary concern to us here.

Sub_select (P) (T) returns *copies* of all subtrees of T that match P. For now, let us interpret "match" as meaning "having the same structure and the same node contents". We will extend this definition later. As an example, the result of **sub_select** (T2) (T1) on the trees of Figure 6.3 is a set consisting of two trees, which are shown in Figure 6.4. These trees represent the two subtrees of T1 which match T2. These two trees in the result are distinct because no two nodes in a tree have the same identity due to the use of the cell constructor described earlier. The trees in the result, in particular, do not include nodes that are children of the nodes which actually matched the pattern. To see why this is so, consider the case of string matching. If one is searching for all occurrences of "ab" in "dabcabcd", the result will be the two occurrences of "ab". The result will *not* be {"abcabcd", "abcd"}. We explore this correspondence with regular expressions in more depth later. We will also define some operations which are capable of retrieving subtrees that extend "down" (and "up") the tree as far as possible. Initially, however, we will concentrate on finding only matching subtrees, according to our definition (which is standard; see [Karp72, Hoff82]).

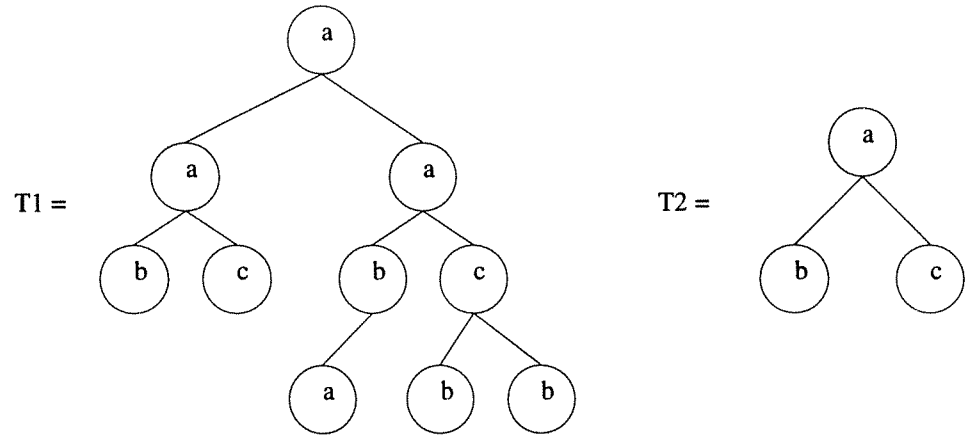

Figure 6.3: A tree (left) and a pattern

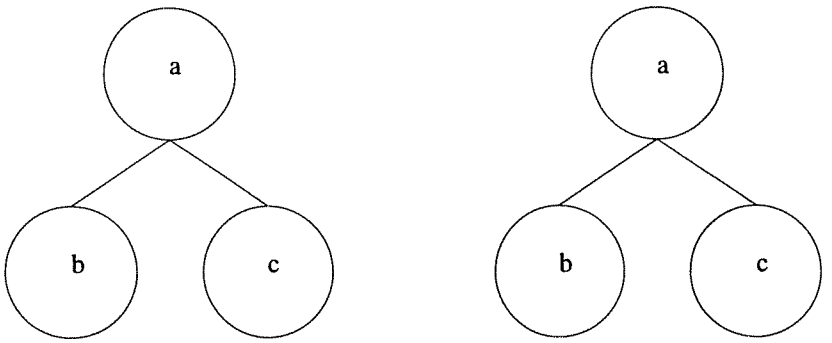

Figure 6.4: Result of a **sub_select**

Note that we could just as easily have defined **sub_select** to return a set of nodes corresponding to the roots of the matched subtrees. Logically, however, we desire the entire subtree. An implementation could of course return only the roots if desired, materializing the rest of the subtree(s) on demand. The next section discusses the **sub_select** operator in more detail and explains its definition further. The definition conforms to that used in [Karp72, Hoff82].

Apply (f) (T) applies the function f to the ''content'' of each cell (node) of the tree to transform the existing object into a new object. The edge set remains the same. This ensures that the structure of the tree is not modified.

The result is built of new cells that contain the transformed objects (see Figure 6.2).

Find_path (T, x) returns a list of nodes encountered on the path from the root of the tree T to the node x. The last node of the result list is x, and the first node is the root of the tree. For example, in the first tree of Figure 6.2, the result of **find_path** (T, 6) would be [1, 3, 6].

Predicates for ordered types are similar to those for the other bulk types: we wish to allow the operations available on the type to appear in predicates involving objects of that type. For example, membership tests and entire selection queries can appear in predicates involving sets. Thus we allow for position dependent functions like **im_ancestor** (G, a) to appear in a predicate involving an object of an ordered type. So, a **select** with such a predicate would return all nodes of graph G which satisfy the condition, i.e. nodes that are **im_ancestor**s of node a:

select (λ (p) p = **im_ancestor** (G, a)) (G)

Ordered bulk types, unlike sets and multisets, have a notion of the "position" of the constituent objects. This opens up possibilities of having a richer predicate language, a regular-expression like language that would allow for queries that would involve matching. For example, in the case of a set of genomes which are sequences of proteins, we might want to select all genomes which contain protein A followed by a sequence of only protein U, followed by protein A again. We would want to express this by something like the following expression:

select (AU^+A) (Set_of_genomes)

The above query can be specified with the AQUA operators. This notion can be extended to include matching predicates for trees (and graphs). Such queries are discussed in more depth in the next section.

6.4.2. Queries on Trees

In this section we present several forms of useful queries on trees and forests. The next section describes some algebraic techniques for processing these queries. In this section we are concerned mainly with the **sub_select** operator. Recall that the standard **select** operator is defined to return a set of nodes based on the properties of the contents of those nodes. The **sub_select** operator returns all subtrees of a tree that satisfy a certain property. In other words, **sub_select** takes connectivity and structure into account while **select** does not.

Section 6.4.2.1 presents a notation for specifying patterns in trees; this notation is an extension to regular expressions for lists (sequences). The next two subsections describe, using algebraic examples, some useful queries.

Section 6.4.3 will deal with alternative algebraic representations for these queries in order to facilitate optimization.

6.4.2.1. Patterns in Trees

The notation we develop in this section is intended to be a part of the formal algebra, not necessarily part of a user-level query language. Ideally, our formalisms would be the result of a simple translation from a graphical user-level query language to a format suitable for internal query processing.

The simplest kind of pattern matching for trees is that exemplified in Figure 6.3. There, we wanted to retrieve all subtrees of a tree T1 that had the same structure and the same node contents as some other tree T2. This is a very simple kind of tree matching. It is analogous to finding all occurrences of some specific string (e.g. "abc") inside some other string. Such a query on trees could be used, for example, in a genealogical database, in which a set of people is structured as a tree in the obvious fashion. In this case the example of Figure 6.3 would correspond to a query such as "retrieve all the portions of this family tree in which somebody named *a* had a first child named *b* and a second child named *c*".

We may wish to ask more sophisticated queries, however. For example, a query such as "retrieve all the portions of this family tree in which somebody named *a* is an ancestor of somebody named *b*". In this case we are not searching the family tree for a specific subtree, but rather for any subtree which matches the predicate "somebody named *a* is an ancestor of somebody named *b*". In the case of lists, such conditions can be stated using regular expressions. For example, to find all substrings of a string in which a comes before b, we would write the regular expression a?*b, where "?" is a symbol representing any character (a "don't care" symbol).

To extend the standard regular expression notation to trees, we build on the results of [Done70, That68]. These papers present generalizations of finite automata which can recognize trees rather than just strings. [That68] proves the closure of these generalized automata under union, concatenation, and Kleene *. These are exactly the three operators of regular expressions, and we now describe a notation that allows their application to terms that represent trees.

To introduce the notation, we must remember that there is a clear distinction between terms that represent trees and terms that represent patterns, just as there is a difference between a regular expression and an actual string. For example, consider the string "abacdeb" and the regular expression "b*de". The regular expression matches the substring "bacde" of the first string. As a simple example of the notation, consider the tree (not pattern) represented by

"a (b c)". This is a tree with "a" at the root, "b" at the left child, and "c" at the right child. In our notation for trees, a node is followed by "(" then by its children (in order -- in this section we do not consider unordered trees, although most of the ideas apply there as well) then by ")". This corresponds to a preorder listing of the nodes.

As another example, consider the tree term T = "a (b (d e) c (f g))", representing a full binary tree with three levels. The notation for tree *patterns* extends the simple tree notation in a manner similar to the extension made to strings by regular expressions. In what follows, it should be clear from the context whether a tree or a pattern is being described by a particular term. As a simple example, consider the *pattern* represented by the term "a (b c)" as a pattern. It matches a *subtree* of T which is represented by the term "a (b c)". Note that just as in the matching of substrings to regular expressions, we are not interested in what *follows* the matching subtree in T. We are interested in finding the matching subtree only, just as in the above regular expression example we noted that the matching substring is ''bacde'', not ''bacdeb''. The **sub_select** operator is defined to return the matching subtrees, not what follows them (see examples below).

The basic notation is the same as that of regular expressions: * for Kleene closure, | for union (disjunction), and "ab" for "a concatenated with b". The only fundamental difference is in the meaning of the concatenation operator. In a regular expression, which always represents a string (i.e., a total ordering), "ab" simply means that b follows immediately after a. However, a node in a tree may have more than one child, that is, more than one node following immediately after it. Intuitively, then, any notation for concatenation of tree patterns must indicate *which child* "b" is supposed to be in the expression "ab".

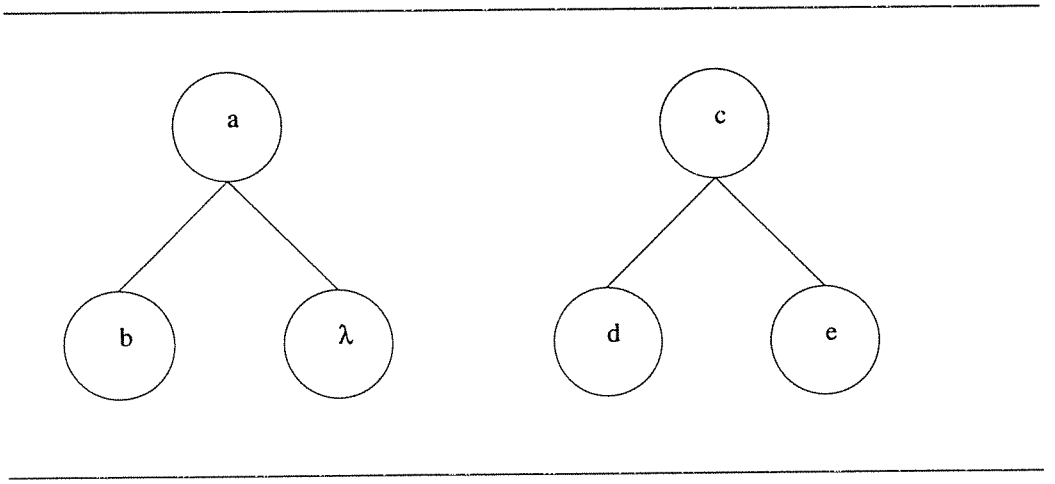

Figure 6.5: Tree patterns

To represent this, we use (as in [That68]) a special symbol to indicate the *concatenation points* -- the points in the expression where the second term is to be appended to the first. Let us first examine this graphically. In Figure 6.5 we have two trees. The special symbol λ in a node indicates a concatenation point. This is not to be confused with the λ of lambda-calculus. Unfortunately, both are standard notations, but it will always be quite clear from the context which λ we are talking about. The concatenation of the left and right trees in Figure 6.5 gives the result in Figure 6.6.

As a slightly more complicated example, consider the trees of Figure 6.7. Here the symbol λ appears twice in the left tree. The meaning of the concatenation point is that *all* occurrences of the concatenation point are to be replaced with the tree on the right, giving the result in Figure 6.8.

The union operation on tree patterns is no different from its regular expression counterpart. The Kleene * operator, however, is based on the concatenation of one pattern onto itself, any number of times. Thus it also needs to make use of the notion of concatenation points. As an example, consider again the left tree of Figure 6.5, and call it T. Then some of the elements of T^* are shown in Figure 6.9.

So far we have considered only one concatenation point symbol, λ. In reality, more than one such symbol is allowed, giving rise to an entire family of concatenation and Kleene * operators, parameterized by the particular concatenation point symbol being used. This results a paradigm for tree pattern matching that is quite similar to the

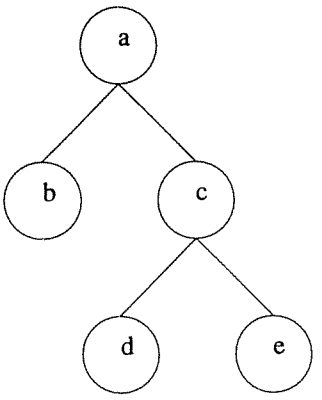

Figure 6.6: A concatenation

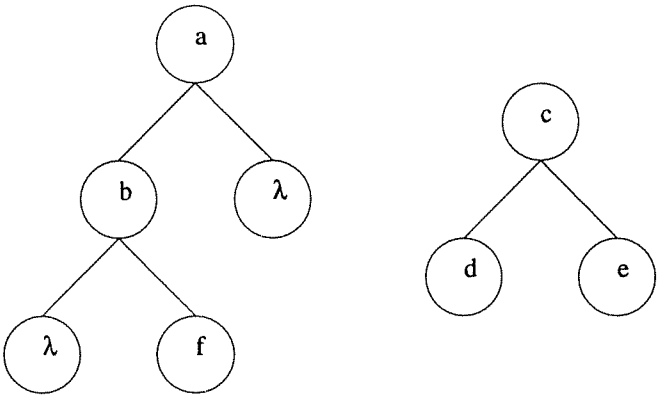

Figure 6.7: Multiple concatenation points

execution model of Prolog and to unification [Seth89]. All of the subsequent results of this chapter hold when multiple concatenation point symbols are allowed. For simplicity of presentation we restrict ourselves to just λ.

We have just defined a semantics for the |, Kleene *, and concatenation operations, based on results of [Done70, That68]. We now describe the syntax that will enable us to express these patterns inside an algebraic query. The basic idea is that any concatenation, including one engendered by a Kleene * operator, must be given

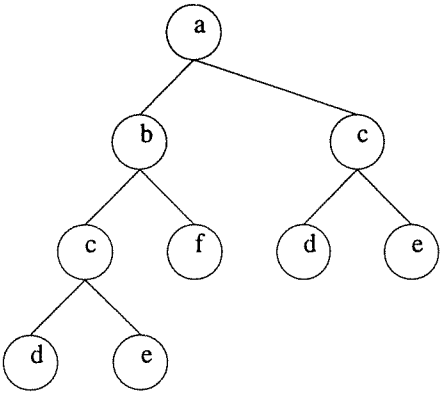

Figure 6.8: Result of Figure 6.7

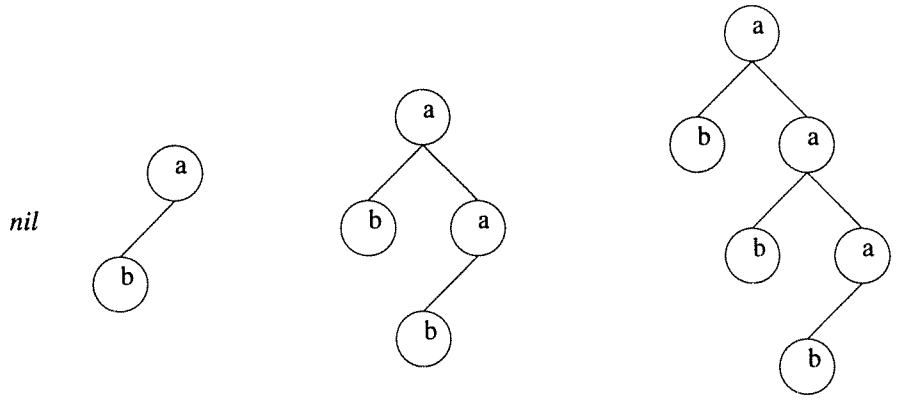

Figure 6.9: Part of a Kleene closure

one or more instances of λ as concatenation points. As a simple example, the concatenation of the two trees in Figure 6.5 would be expressed as

[a (b λ)] c (d e)

The square brackets are simply used for grouping, and in this case are not strictly necessary. The contents of a node is followed by "(" then by each of its children in order then by ")". This corresponds to a preorder traversal of the tree. The result of the concatenation, in Figure 6.6, is expressed as

a (b c (d e))

In this expression, the node "b" is not followed by a "(", indicating that it is a leaf node. Technically, this is shorthand for the following expression, which explicitly includes all nil subtrees using the special symbol *nil*:

a (b (*nil nil*) c (d (*nil nil*) e (*nil nil*)))

We omit the *nil* symbol when the meaning is otherwise clear. To continue the examples, the concatenation of the trees of Figure 6.7 is described as follows in our syntax:

[a (b (λ f) λ)] c (d e)

Note that the λ symbol only has meaning when there is something to replace it with. The expression "a (b (λ f) λ)" by itself is not a well-formed expression.

The Kleene * operator is indicated by placing a * on the appropriate terms of an expression. As an example, T* for the left tree T of Figure 6.5 is expressed as

[a (b λ)]*

Only a node together with all of its children may be the subject of concatenation or the Kleene * operator. For example, the expression "a* (b c)" is not a well-formed expression. This means that the square brackets (used for grouping) in the expression above are actually redundant; the * will be seen as applying to a node and all of its children if it appears after the last of the children.

The union operation for tree patterns is identical to that for regular expressions. As a simple example, the following expression is the Kleene * of the left tree of Figure 6.5, but in which the root can be either "a" or "f":

[[a | f] (b λ)]*

Again, square brackets are used for grouping.

Let us examine a more complicated example of concatenation. Consider the following pattern:

[a (λ λ)] [[c | d] (e f)]

Formally, the result of a concatenation is defined as the set of all trees formed by replacing every λ in every tree matching the first pattern with a tree matching the second pattern. Not every λ need be replaced by the same tree from the second set, but every λ must be replaced by one of them. Figure 6.10 shows two of the four trees that satisfy the previous pattern.

In regular expressions, it is frequently desirable to have a "don't care" or "wild card" character. This concept can be extended to tree patterns as well, with some minor modifications. The symbol "?" indicates that the contents of a node can be anything. Thus a tree with any value at the root and the values "b" and "c" as the left and right children, and no other nodes, would be expressed as

? (b c)

and a tree with "a" at the root, "b" as the left child, and anything as the right child is written as

a (b ?)

This "?" symbol, however, stands only for any possible *contents* of a node, not for any possible subtree. In other words, "?" represents a tree with one node whose contents are unknown. To represent the most general tree pattern,

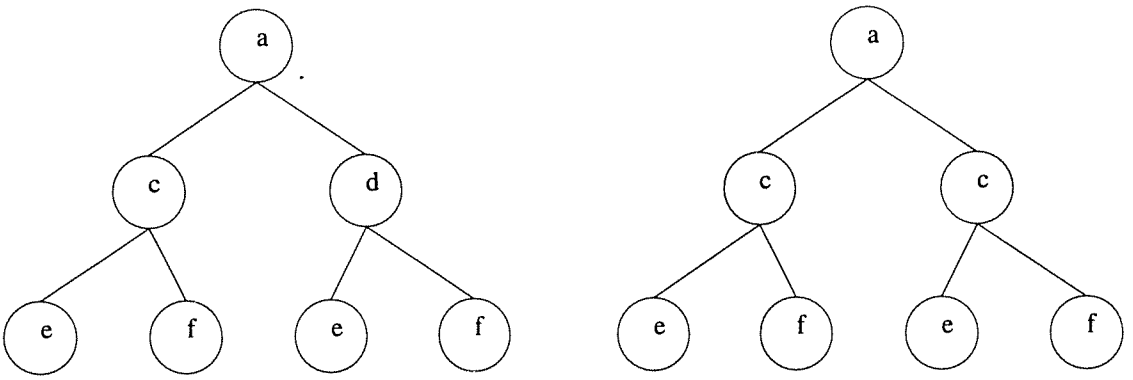

Figure 6.10: Two concatenations of [a (λ λ)] [[c | d] (e f)]

which will match any tree at all, we need an analog to the "?*" of regular expressions. We define the symbol $T_?$ to stand for any binary tree as follows:

$$T_? = [? (\lambda \lambda)]^*$$

This definition works only for binary trees. Clearly a similar definition can be provided for n-ary trees for any n. For trees with unbounded numbers of children, however, such an expression is not possible.

An important special case of tree patterns are those in which we are only interested in the structure of the tree [Karp72], not in the contents of the nodes. The "?" symbol makes such patterns easy to express in our notation.

As a further notational convenience, as mentioned above, we assume that any node not present in a pattern is *nil*. To make this more useful, we would like to be able to number the children in a pattern, so we extend the notation appropriately. For example, the pattern

$$a\ (b_3)$$

matches a tree whose root is "a" and whose third child is "b". All of the other children of "a" are *nil*, as are all of b's children. For n-ary trees, this is merely a shorthand.

To summarize the notation, then, it is an extension of regular expressions in which concatenation and Kleene * apply only to a node together with all of its children. The special symbol λ is used to indicate concatenation points, *nil* indicates the absence of a subtree, and "?" matches any possible node contents. In what follows we will assume

that all of the trees are n-ary for some n (the nodes have no more than n children each). We will use $T_?$ to match any possible n-ary tree.

In the next two subsections we give some examples of algebraic queries using the **sub_select** operator and the patterns defined in this section.

6.4.2.2. Selecting Portions of a Tree

Here we present some examples of queries that select from some tree T all subtrees matching a given pattern. As a realistic application, consider a relational query optimizer which represents queries as trees. All operators are unary or binary, and to simplify the presentation we will ignore additional parameters, so the Tree type constructor would have been invoked by

Query = Tree [OpNode, 2].

Given some query tree Q, the following AQUA algebra expression returns all subtrees of Q representing a join whose left input is also a join, assuming that the left join input is the left child of the join node:

sub_select (join (join ?)) (Q)

We do not specify the children of the inputs to the first join. The query is intended to return only the portion of the tree with this structure, not any of its children.

Now consider a query to retrieve all subtrees of Q representing a join whose left input contains a selection somewhere in it:

sub_select (join ([[? [(λ ?) | (? λ)]]* select] ?)) (Q)

The disjunction ensures that any subtree containing a selection will match the pattern. Intuitively, it ensures that any number of left and right "turns" leading to a selection will qualify. Note that to also retrieve the entire right input to the join (actually, all initial subtrees of the right input), rather than just the root node of the right input, the last "?" in the query would be replaced with "$T_?$".

In these examples we have been assuming very simple node contents -- immutable strings, to be exact. However, the syntax easily accommodates arbitrarily complex node contents. Any algebraic expression which evaluates to something of the appropriate type can be used to specify the contents of a node inside a tree pattern. We have kept it simple here to illustrate the concepts. It would also be possible to define an extended version of **sub_select**

which takes an additional parameter indicating an algebraic expression to be applied to each node. The result of this expression, rather than the actual node, could then be matched against the pattern.

6.4.2.3. Selecting Trees from a Forest

In the previous subsection we saw some simple algebraic queries to retrieve all portions of a tree satisfying some property. Here we look at another type of query. We now wish to find all trees in some set of trees that satisfy some pattern. Continuing with our relational query processor example, suppose we now wish to do multiple query optimization, so we maintain a set QS of queries. Identifying common subexpressions is an important part of multiple query optimization.

The following algebra query will tell us which of our queries contains a join of relations A and B:

select (
    λ(q)**sub_select** (join (A B)) (q)
        ≠ ∅) (QS)

This expression returns a copy of QS containing only those query trees q such that the search for subtrees turned up at least one match.

A more complicated query is "retrieve all queries q whose rightmost base relation input is relation C". This can be expressed algebraically as follows:

select (
    λ(q)**sub_select** ([? ($T_?$ λ)]* C) (q)
        ≠ ∅) (QS)

We have demonstrated the use of patterns for trees and the use of the **sub_select** operator which locates them in a given tree. We now turn our attention to some other operations which will enable some interesting optimizations.

6.4.3. Other Algebraic Operations on Trees

The **sub_select** operator can clearly find all occurrences of any tree pattern inside any tree. However, there are other ways of expressing the same queries. In this section we define some operations that will allow for alternative ways of expressing many interesting queries over trees and forests.

The **PT** (powertree) operator takes a tree T and returns all subtrees of T. The definition of "subtree" is analogous to the definition of "substring". Just as "bc" is a substring of "abcd", the tree represented by

a (c d)

is a subtree of T, the left tree of Figure 6.10. Note that

a (c *nil*)

is not a subtree of T. A subtree must include either all or none of the nodes following a node appearing in the subtree. As an example, the powertree of the tree in Figure 6.6 is shown in Figure 6.11. Like the powerset operator for sets, **PT** is an operator that will hopefully never need to actually be executed. It is, however, a very powerful operator. Using **PT**, we can define a new operator, called **all_desc**, as follows:

all_desc (P) (T) = **select**
(λ(t) **exists** (λ(s) **choose** (**sources** (s)) =
choose (**sources** (t)) ∧ **sinks** (s) ⊆ **sinks** (T))
(**sub_select** (P) (t))) (**PT** (T))

All_desc (P) (T) retrieves copies of all subtrees of T which *start with* the pattern P, and include all descendants of that occurrence of P. As an example, consider the tree T in Figure 6.12. The result of **all_desc** (a (b a)) (T) is pictured in Figure 6.13. The query "**sub_select** (P) (T)" in this case would return three distinct trees of the form "a (b

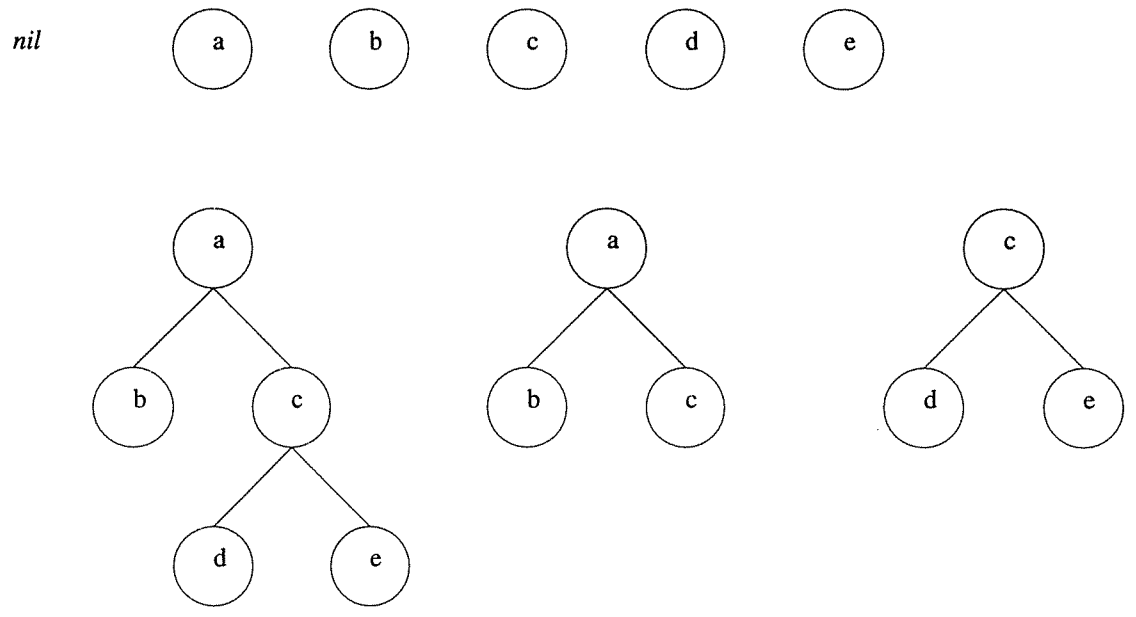

Figure 6.11: A powertree

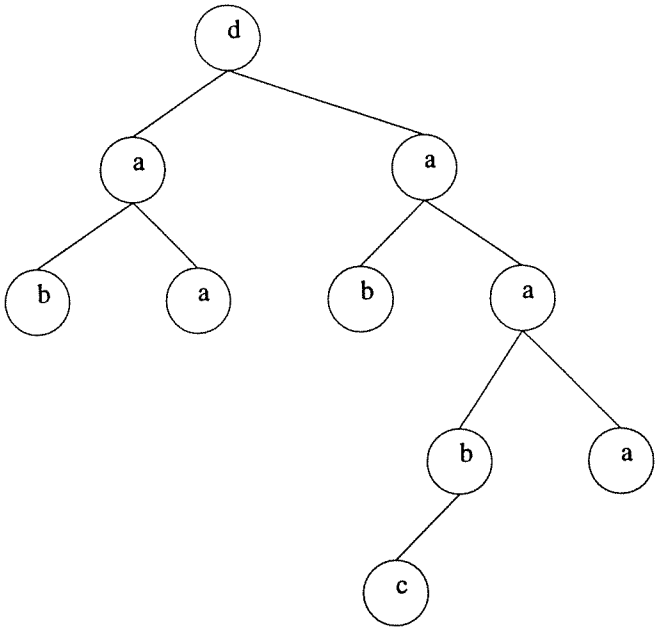

Figure 6.12: A tree

a)".

One motivation for the **all_desc** operator can be illustrated by the following example. Consider the tree T of Figure 6.14 and the query

sub_select (e (a b)) (T).

This query can be rewritten using *all_desc* as follows:

collapse (**apply** (λ (s)
sub_select (e (a b)) (s))
(**all_desc** (e) (T)))

where **collapse** is similar to the SET_COLLAPSE operator of the EXCESS algebra. This version of the query first finds all subtrees of T whose root contains simply "e". The query then finds all subtrees of each of these subtrees that have "a" and "b" as the children and "e" as the root.

The real optimization occurs when we have an index that will return all nodes containing "e". In that case, the **all_desc** operation makes direct use of the index to compute its result. The **sub_select** operations, in this case, will

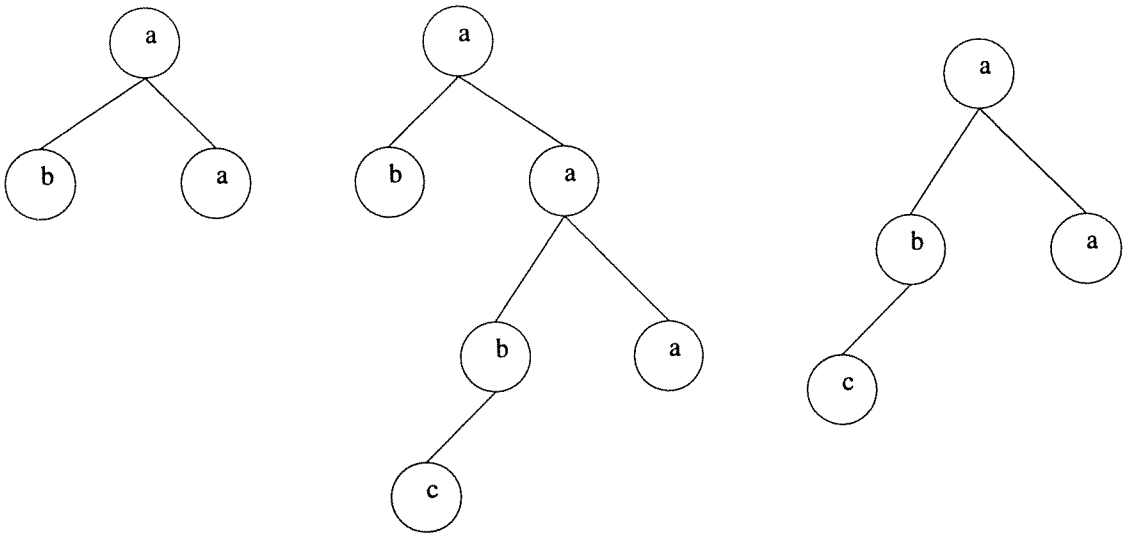

Figure 6.13: A result of an **all_desc** operation

only be examining subtrees with the proper root node. Assuming the situation of Figure 6.14, in which there may be thousands of nodes in the outlined region R, none of which contain "e", the processing time for the query is potentially orders of magnitude faster than in the initial version. We have eliminated examination of all nodes not descended from the node labelled "e" in Figure 6.14. Note that this solution is effective only under the (reasonable) assumption that, given a node via the index, we have a fast way of retrieving its children.

This technique is similar to the technique of splitting a conjunctive selection predicate in relational systems. One of the attributes in the conjunction has an index, so conceptually the selection is split into two selections so that the second comparison only takes place when the first one has succeeded. Note that the rewriting used above does not always result in a more efficient execution of the query, even if an index is used. For example, if "e" occurred many times in the same large subtree, many copies of parts of that subtree would be returned, resulting in a potentially longer search than with the original **sub_select** query.

Suppose now that we have available an index which provides fast access to all nodes containing "e" that also have "a" as their first child. A similar rewriting of the query can facilitate the use of such an index:

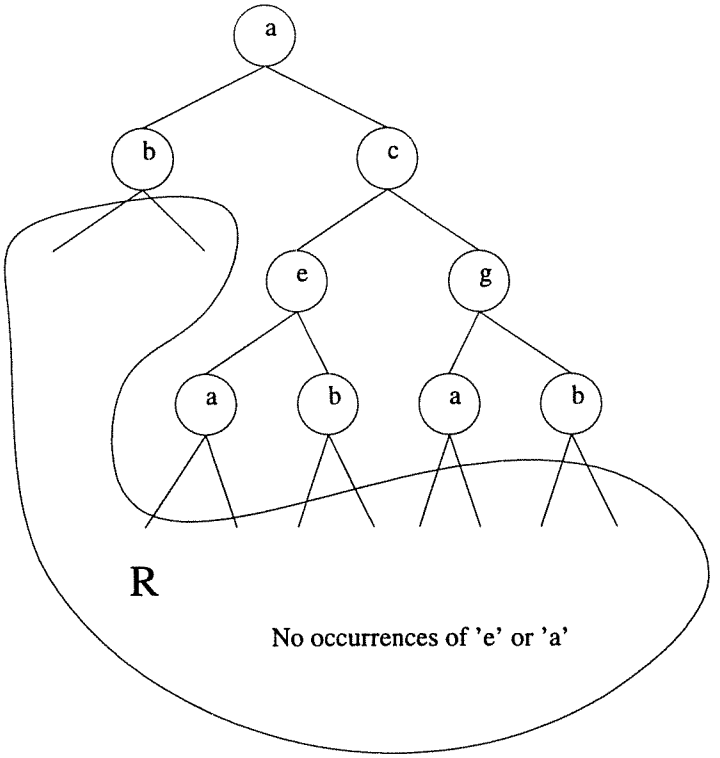

Figure 6.14: A large tree

```
collapse (apply (λ (s)
    sub_select (e (a b)) (s))
        (all_desc (e (a ?)) (T)))
```

In this case, the index is in general even more restrictive, leaving even less work for the expensive (**sub_select**) portion of the query. Clearly, a query optimizer will have to know how to recognize such patterns in queries and to apply the appropriate transformations when indices exist. However, if queries and patterns are represented within the optimizer as trees, then portions of the optimization process can be viewed as queries in their own right, thus giving the optimizer the ability to optimize its own operation as well as that of the user's queries.

We next define an **all_anc** operator, which travels up the tree in the way that **all_desc** travels down the tree. In other words, **all_anc** (P) (T) will retrieve copies of all occurrences of P in T plus the path from the root to P and all immediate children of the nodes on that path. As an example, the query

all_anc (a) (T),

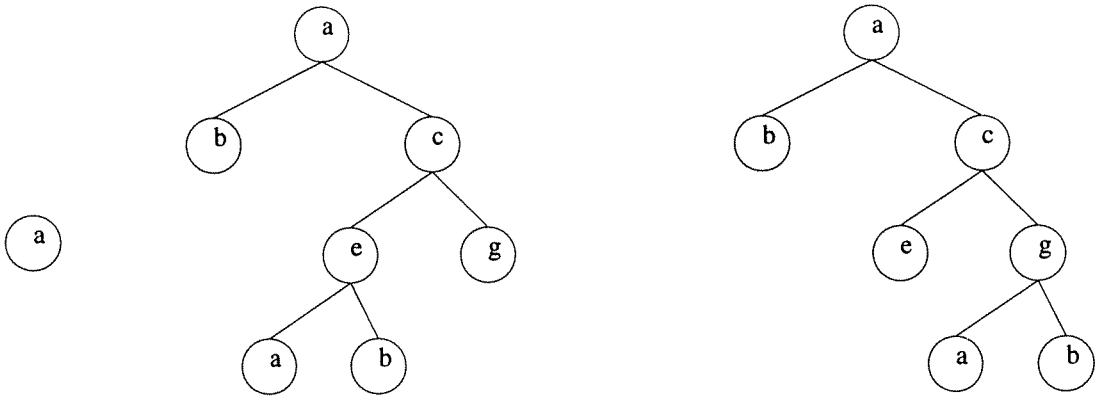

Figure 6.15: Result of an **all_anc** operation

where T is the tree of Figure 6.14, would return the two subtrees of T pictured in Figure 6.15. Using this, we can rewrite the original query as follows:

```
collapse (apply (λ (s)
    sub_select (e (a b)) (s))
        (all_anc (a) (T)))
```

With this rewriting, we can make use of an index that provides fast access to all nodes in T containing "a". Once again, we avoid the large region R of the tree in Figure 6.14, resulting in a potentially massive improvement in execution time. Once again, this solution is effective only under the (reasonable) assumption that, given a node via an index, we have a fast way of retrieving its parent.

The distinction between the **all_anc** and **all_desc** rewritings of the query is simply this: In the **all_anc** rewriting, we used an index to locate all occurrences of "a", and anything preceding "a" in the tree. This works because "a" occurs at the lowest level of the pattern originally supplied to the **sub_select** operator, thus we do not need any of the children of "a". In the **all_desc** rewriting of the query, we used an index to locate occurrences of "e", which was at the root of the pattern originally supplied to **sub_select**. Thus the **all_desc** operation was able to ignore anything occurring before "e" in T.

Finally, consider an index which provides fast access to nodes labelled "e" and to nodes labelled "a". In this case we can use both the **all_desc** and **all_anc** operators to achieve greater optimizations as follows:

```
apply (λ (x) sub_select (e (a b)) (x))
        collapse (apply (λ (s)
                  all_desc (e) (s))
                        (all_anc (a) (T))))
```

This actually gives us an extremely powerful paradigm for optimizing queries involving pattern matching over trees. Given any pattern with a specific root object in the node, we can use any available indices to locate these root nodes by using the **all_desc** rewriting of the query. It doesn't matter if the rest of the pattern is extremely complex -- in fact, the more complex the rest of the pattern, the more we save by avoiding the pattern-matching. This is especially true of patterns including the Kleene * operator below the root.

Given any pattern with some specific leaf object, we can use any available indices to locate nodes containing this object by using the **all_anc** rewriting of the query. Again, the more complex the pattern above this node, the more savings will accrue.

The main thing to remember about the **sub_select**, **all_desc**, and **all_anc** operators is that **sub_select** returns matching subtrees, but nothing coming before or after the match (just like in regular expression matching); **all_desc** returns matching subtrees along with everything following them; and **all_anc** returns matching subtrees and everything preceding them. These operators generalize similar AQUA operators for lists [Subr93].

Note that in general, the **all_anc** operator should return more than just the children of the nodes on the path back to the root. The more levels the pattern has, the more levels of descendants the **all_anc** operation should gather from each node in the path back to the root. To be specific, if the length of the longest path in the pattern is n > 0, it will suffice to retrieve n complete generations of descendants from the nodes along path from the pattern to the root. If n = 0, one generation must be retrieved. For patterns of unbounded depth (i.e. those involving Kleene *), then, the **all_anc** rewriting may not always be practical. It can always be made correct, however, by going as deep as possible in the tree.

A complete definition of **all_anc** thus involves an additional integer parameter to indicate how many levels of descendants need to be returned along with their parent nodes on the path from the root to the pattern:

all_anc (n, P) (T)

operates as described in the previous paragraphs. Given some pattern P and a node in that pattern, it is clearly easy to compute the appropriate value for the parameter n.

6.4.4. Algorithms and Indexes

In this section we briefly discuss the applicability of various tree-matching algorithms and the desirability of various forms of indexing on trees and forests.

Most of the algorithms proposed for tree matching [Karp72, Hoff82] solve the following problem: Given a tree T2 (or a finite set of trees) return all the locations where T2 occurs in tree T1. Note that T2 is a tree, not a pattern. The algorithms match specific subtrees, and do not involve the standard concatenation or Kleene * operators, although they do have the ability to imitate the "?" symbol by simply ignoring the contents of nodes and concentrating on the structure. They also can imitate union to some extent because T2 can be a set of trees.

[Karp72] presents some of the earliest tree-matching algorithms. These are improved on by using additional preprocessing in [Hoff82]. The latter paper presents 8 algorithms for solving the tree matching problem, and describes the space and time complexities of each. These algorithms are all ways to process our algebra query

sub_select $(P_1 \mid P_2 \ldots \mid P_n)$ (T)

where the P_i have no concatenation or Kleene * operators. The algorithms can of course also be used to compute answers to the **all_anc** and **all_desc** operators, with the same pattern restrictions.

The dominant parameters in the complexity analyses include the numbers of nodes in T and the P_i, the number of possible contents for a node (which in many database applications will be infinite), the maximum number of children for a node, and the number of actual matches found. All of the algorithms except the naive one require preprocessing of the "patterns". Many of the algorithms also involve assumptions about the structure of the "pattern" trees and the availability of inexpensive bit-string operations. The crucial question, of course, is which of the algorithms will be efficient in a disk-based DBMS setting. This question deserves further study.

In some of the examples in the previous section we posited the existence of various indices. In particular, standard DBMS technology can be used for many useful indices over tree-structured data. In the case of queries over a single tree, all but one of the query rewritings presented above made use of an index that took some information about a node as input and returned that node as output. Standard B-Tree, ISAM, hashed, etc. indices all serve this purpose fairly well under varying circumstances.

However, in processing queries over forests, we may want more information than a standard index could provide. A standard index would simply reveal which trees in a forest contained a particular value in some node. For

some queries this might be sufficient, but for others we may wish to also know exactly where in the tree the node occurs. Thus an index containing both a pointer to the root of the tree and a pointer to the node within the tree might be useful in certain environments.

A more interesting question is the issue of indexes whose input is a pattern rather than just the contents of a single node. One of the rewritings above made use of such an index. For example, one might desire an index over a genealogy tree which points to all people who have no siblings, or to all people named Fred with a child named George, or to all people named Fred with a descendant named George. This area requires further research as well.

CHAPTER 7

CONCLUSIONS

7.1. Summary

This thesis surveyed the current state of algebraic query processing and optimization in an effort to identify areas for improvement. We then presented an advanced data model with which to study the extension of the algebraic paradigm. An algebra that successfully captures the important features of this model was described. Using an even more powerful model, we extended the algebraic paradigm even farther.

We presented the design of the EXTRA data model and the EXCESS query language. EXTRA and EXCESS represent a synthesis and extension of many ideas from other data models, including GEM, NF^2 models, DAPLEX, ORION, POSTGRES, GemStone, and O_2. The extensions include: complex object support based on an interesting mix of object- and value-oriented semantics; a user-friendly, high-level query language that captures these semantics; support for the storage and manipulation of a database of persistent objects of any type, not just sets of tuples; and support for user-defined types, functions, and procedures, both at the abstract data type level and at the conceptual schema level.

To model queries in EXTRA/EXCESS, we extended the algebraic paradigm 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 thesis 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. A proof of the equipollence of the algebra and EXCESS is provided (in Appendix B), and a list of the algebraic transformation rules (i.e., potential optimizations) is also given (in Appendix A). The rules provide valuable optimizations for object-oriented queries and the proof provides a complete semantics for EXCESS queries as well as an algorithm for translating an EXCESS query into the algebra. Clearly, the algebra is closed.

The AQUA data model and algebra has been proposed [Leun93] as an input language for object-oriented query optimizers. It has been designed to cover the functionality of many existing query languages, and to provide the maximum potential for optimization. AQUA supports abstraction, varying notions of equality among objects, and

ordered type constructors (e.g. trees). We provided algebraic formulations for these features which will allow for query optimization, and described the fundamentals of selection query processing on trees.

7.2. Conclusions

The algebraic approach to database query processing continues to be successful long after its introduction in the relational model. This thesis has demonstrated this using the EXCESS algebra and portions of the AQUA algebra. The EXCESS algebra's utility lies in its provable equipollence to the EXCESS query language and in its flexible operators and transformation rules, which can be applied to other systems as well. As described in the previous section, EXTRA/EXCESS is in fact a generalization of many other data models, thus the extension of the algebraic paradigm to this model is particularly compelling.

The EXCESS algebra contains the following original features: 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 an algebra and a calculus; 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 DDL) and process the queries of EXCESS (the DML). The primitive nature of the algebraic operators allows other operators to be defined in terms of them quite readily. This results in the ability to test a wide variety of algebraic operators for utility and optimizability.

In AQUA, the introduction of abstraction as a type constructor and the use of non-determinism to model certain forms of duplicate elimination allows the algebra to capture more features of the data model and query language. The specific operators and techniques described for queries over trees indicate that the algebraic paradigm will continue to be useful even for complicated forms of structured data and for complex queries.

7.3. Future Work

Future work on these topics falls into two categories, general data modeling and more specific investigation of particular operators.

Current and future research in the area of data modeling will take place in the context of the AQUA system. The plans include investigation of additional operators on ordered bulk types (e.g., a least fixpoint operation) and more expressive predicate specification sub-languages for ordered bulk types. For example, we can specify regular expression-like predicates on lists and trees, but more powerful mechanisms (such as context-free grammars) need to be examined. In this thesis we described algebraic techniques for modeling certain queries on trees. The extension of these ideas to graphs needs to be investigated. In particular, the extension of the predicate specification language may prove especially interesting. Indexable ordered types in AQUA (such as N-dimensional arrays) also need to be examined further. Such structures are becoming increasingly important as the volume of scientific data (and the ease of access to it) increases. Finally, the question of what to do with non-deterministic intermediate results in a query needs to be addressed. The current default is that if a user asks a non-deterministic query, the results are simply unpredictable. An alternative is to somehow restrict access to non-deterministic data, so that only deterministic results are seen.

In the area of specific operators and optimization techniques, future work includes an investigation of cost functions and useful statistics (and how to manage them) for complex object data models. Of particular interest are estimation functions for second-order operators such as SET_APPLY, which takes a query as one of its inputs. More research is needed on indexing for ordered data types (trees, lists, etc.). In this thesis we assumed the existence of various forms of indices on trees, but it is not at all clear what form these indices should take. Implementation techniques for the tree-based operations need to be evaluated under the assumption that some data will reside on disk. The relative efficiencies of the algorithms described in [Hoff82] may no longer hold if disk-based data is taken into account.

APPENDIX A

Transformation Rules of the EXCESS Algebra

This appendix contains a list of transformation rules (involving both primitive and derived operators) in the EXCESS algebra. These transformations capture those of the relational and multiset algebras from the literature [Ullm82, Ullm89, Kort91, Knut81] as well as most transformations described for more advanced data models [Beer90, Scho86]. We do not claim that this list is complete, but we believe that it is either complete or very close to complete. The rules are presented without proof. In most cases the proof is a straightforward application of the operator definitions.

A.1. Some Non-Primitive Operators

Before describing the transformation rules, we define a few operators in terms of the 23 primitive ones:

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

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

Multiset selection: $\sigma_E (A) = SET\_APPLY_{COMP_E} (A)$

Relational-like $\times$: rel_$\times$ =

$SET\_APPLY_{TUP\_CAT(TUP\_EXTRACT_1(INPUT),TUP\_EXTRACT_2(INPUT))}(A \times B)$

Relational-like Θ **join**: $rel\_join_\Theta(A, B)$ =

$SET\_APPLY_{COMP_\Theta(INPUT)} (SET\_APPLY_{TUP\_CAT\ (field1,\ field2)}(A \times B))$,

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 $TUP\_EXTRACT_{field1}(INPUT)$, etc. This kind of join follows the definitions of [Kort91]. 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 $TUP\_EXTRACT_j(INPUT)$.

Array union: ARR_UNION (A, B) = ARR_CAT (A, ARR_DIFF (B, A))

Array selection: $\alpha_E (A) = ARR\_APPLY_{COMP_E} (A)$

Tuple projection: $\pi_{f1,...,fn}(T)$ =

TUP_CAT (TUP (TUP_EXTRACT$_{f1}$(T)), TUP_CAT (.... , TUP (TUP_EXTRACT$_{fn}$(T))))

An intersection operator for arrays could also be defined; the definition would be just like the multiset definition.

A.2. Rules for Multiset Operators

The primitive binary multiset operators are $\uplus$, $-$, and $\times$. The derived binary multiset operations are $\cup$ and $\cap$. The primitive unary multiset operators are DE, GRP, SET, SET_COLLAPSE, and SET_APPLY. The derived unary multiset operator is σ.

A.2.1. Binary Operators: Associativity, Commutativity, Distributivity

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

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

$A \cup (B \cap C) = (A \cup B) \cap (A \cup C)$

$A \cap (B \cup C) = (A \cap B) \cup (A \cap C)$

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

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

A <op> A = A; <op> $\in$ { $\cap$, $\cup$ }.

$A \cup (A \cap B) = A$

$A \cap (A \cup B) = A$

$A \cap (A \uplus B) = A$

$A \cup (A \uplus B) = A \uplus B$

$\varnothing$ <op> A = A; <op> $\in$ { $\uplus$, $\cup$ }.

$(A \cup B) \uplus (A \cap B) = A \uplus B$

$(A \uplus B) - (A \cap B) = A \cup B$

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

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

$A \uplus (B - C) = (A \uplus B) - (B \cap C)$

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

Note that relational × associativity (i.e., A×(B×C) = (A×B)×C) and commutativity (i.e., A×B = B×A) do not hold due to the set-theoretic nature of × in EXCESS. However, we have direct analogs:

$$SET\_APPLY_{TUP\_CAT(\pi_1(INPUT),TUP\_EXTRACT_2(INPUT))}(A \times (B \times C)) =$$
$$SET\_APPLY_{TUP\_CAT(TUP\_EXTRACT_1(INPUT),\pi_2(INPUT))}((A \times B) \times C)$$

$rel\_\times (A, B) = rel\_\times (B, A)$

A.2.2. Unary Operators: Pushing

DE (A <op> B) = DE (DE (A) <op> DE (B)); <op> ∈ { $\uplus$, $\cup$ }.

DE (A – B) = DE (A – DE (B))

$DE (SET\_COLLAPSE (A)) = DE (SET\_COLLAPSE (SET\_APPLY_{DE}(A)))$

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

DE (DE (A)) = DE (A)

DE (SET (A)) = SET (A)

DE (A × B) = DE (A) × DE (B)

$DE (SET\_APPLY_E(A)) = DE (SET\_APPLY_E(DE (A)))$

$GRP_E(DE (A)) = SET\_APPLY_{DE}(GRP_E(A))$

$GRP_E(A \times B) = SET\_APPLY_{INPUT \times B}(GRP_E(A))$; E applies only to A. E will need minor rewriting to operate on A elements.

$GRP_E(A \uplus B) = GRP_E(SET\_COLLAPSE (GRP_E(A) \uplus GRP_E(B)))$

$GRP_E(A - B) = GRP_E(SET\_COLLAPSE (GRP_E(A)) - SET\_COLLAPSE (GRP_E(B)))$

GRP is not reflexive unless the cardinality of the result of one of the GRP expressions is 1.

$GRP_E(SET\ (A)) = SET\ (SET\ (A))$

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

$SET\ (A \uplus B) = SET\ (SET\_COLLAPSE\ (SET\ (A) \uplus SET\ (B)))$

SET (A <op> B) = SET (SET_COLLAPSE (SET (A)) <op> SET_COLLAPSE (SET (B))); <op> $\in \{ -, \times \}$.

$SET\ (DE\ (A)) = SET\_APPLY_{DE}(SET\ (A))$

$SET\ (GRP_E(A)) = SET\_APPLY_{GRP_E}(SET\ (A))$

$SET\ (SET\_COLLAPSE\ (A)) = SET\_APPLY_{SET\_COLLAPSE}(SET\ (A))$

$SET\ (SET\_APPLY_E(A)) = SET\_APPLY_{SET\_APPLY_E}(SET\ (A))$

$SET\_COLLAPSE\ (A \uplus B) = SET\_COLLAPSE\ (A) \uplus SET\_COLLAPSE\ (B)$

$SET\_COLLAPSE\ (GRP_E(A)) = A$

$SET\_COLLAPSE\ (SET\ (A)) = A$

$SET\_COLLAPSE\ (SET\_COLLAPSE\ (A)) = SET\_COLLAPSE\ (SET\_APPLY_{SET\_COLLAPSE}(A))$

$SET\_APPLY_E(A \uplus B) = SET\_APPLY_E(A) \uplus SET\_APPLY_E(B)$

$SET\_APPLY_E(A \times B) = SET\_APPLY_{E1}(A) \times SET\_APPLY_{E2}(B)$; E = E1(E2), E1 applies only to A, and E2 applies only to B.

$SET\_APPLY_E(SET\ (A)) = SET\ (E\ (A))$

$SET\_APPLY_E(SET\_COLLAPSE\ (A)) = SET\_COLLAPSE\ (SET\_APPLY_{SET\_APPLY_E}(A))$

$SET\_APPLY_{E1}(SET\_APPLY_{E2}(A)) = SET\_APPLY_{E1(E2)}(A)$

$\sigma_P(A \cup B) = \sigma_P(A) \cup \sigma_P(B)$

$\sigma_P(A \cap B) = \sigma_P(A) \cap \sigma_P(B)$

$\sigma_P(A - B) = \sigma_P(A) - \sigma_P(B)$

$\sigma_P(A - B) = \sigma_P(A) - B$

$\sigma_{\neg P}(A) = A - \sigma_P(A)$

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

A.3. Rules for Array Operators

The primitive binary array operators are ARR_CAT, ARR_DIFF, and ARR_CROSS. The derived binary array operators are ARR_UNION and α. The primitive unary array operators are SUBARR, ARR, ARR_EXTRACT, ARR_COLLAPSE, ARR_DE, and ARR_APPLY. Here I largely ignore the operators ARR_CROSS and ARR_DIFF.

A.3.1. Binary Operators: Associativity, Identity

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

ARR_CAT (A, B) = A; B = [].

ARR_CAT (A, B) = B; A = [].

A.3.2. Unary Operators: Pushing

$$\text{ARR}(\text{ARR\_EXTRACT}_n(A)) = \text{SUBARR}_{n,n}(A)$$

ARR (ARR_CAT (A, B)) = ARR (ARR_COLLAPSE (ARR_CAT (ARR (A), ARR (B))))

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

$$\text{ARR}(\text{ARR\_DE}(A)) = \text{ARR\_APPLY}_{\text{ARR\_DE}}(\text{ARR}(A))$$

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

$$\text{ARR\_EXTRACT}_n(\text{ARR}(A)) = A$$

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

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

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

$$\text{ARR\_EXTRACT}_n(\text{ARR\_COLLAPSE}(A))$$
$$= \text{ARR\_EXTRACT}_x(\text{ARR\_EXTRACT}_y(A));\ x = n \bmod |\text{elt}(A)|,\ y = n \operatorname{div} |\text{elt}(A)|.$$

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

ARR_COLLAPSE (ARR (A)) = A

ARR_COLLAPSE (ARR_CAT (A, B)) = ARR_CAT (ARR_COLLAPSE (A), ARR_COLLAPSE (B))

ARR_COLLAPSE (ARR_COLLAPSE (A)) = ARR_COLLAPSE ($ARR\_APPLY_{ARR\_COLLAPSE}(A)$)

ARR_COLLAPSE ($SUBARR_{m,n}(A)$) = $SUBARR_{x,y}$(ARR_COLLAPSE (A)); $x = m * |elt(A)|$, $y = n * |elt(A)|$

ARR_DE (ARR_CAT (A, B)) = ARR_DE (ARR_CAT (ARR_DE (A), ARR_DE (B)))

ARR_DE (ARR (A)) = ARR (A)

ARR_DE (ARR_COLLAPSE (A)) = ARR_DE (ARR_COLLAPSE ($ARR\_APPLY_{ARR\_DE}(A)$))

ARR_DE (ARR_DE (A)) = ARR_DE (A)

ARR_DE ($ARR\_APPLY_E(A)$) = ARR_DE (ARR_APPLY_E(ARR_DE (A)))

$SUBARR_{m,n}(SUBARR_{j,k}(A)) = SUBARR_{j+m,j+n}(A)$

$SUBARR_{m,n}$(ARR_CAT (A, B)) = ARR_CAT ($SUBARR_{m,|A|}(A)$, $SUBARR_{1,n-|A|}(B)$); if $m \leq |A|$.

$SUBARR_{m,n}$(ARR_CAT (A, B)) = $SUBARR_{m-|A|,n-|A|}(B)$; if $m > |A|$.

$SUBARR_{1,n}$(ARR (A)) = ARR (A); $n \geq 1$.

$SUBARR_{m,n}$($ARR\_EXTRACT_p(A)$) = $SUBARR_{x+m,x+n}$(ARR_COLLAPSE (A)); $x = p * |elt(A)|$.

$SUBARR_{m,n}$(ARR_COLLAPSE (A)) = $SUBARR_{m \bmod x,(m \bmod x)+m-n}$ (ARR_COLLAPSE ($SUBARR_{m \operatorname{div} x,(n \operatorname{div} x)+1}(A)$)); $x = |elt(A)|$.

$SUBARR_{m,n}$($ARR\_APPLY_E(A)$) = ARR_APPLY_E($SUBARR_{m,n}(A)$); E is not $COMP_p$ for some P.

$ARR\_APPLY_{E1}(ARR\_APPLY_{E2}(A)) = ARR\_APPLY_{E1(E2)}(A)$

ARR_APPLY_E(ARR_COLLAPSE (A)) = ARR_COLLAPSE ($ARR\_APPLY_{ARR\_APPLY_E}(A)$)

$ARR\_APPLY_E(ARR\_EXTRACT_n(A)) = SUBARR_{n*x,n*(x+1)}$(ARR_APPLY$_E$ (ARR_COLLAPSE (A))); $x = |elt(A)|$, E is not $COMP_p$ for some P.

ARR_APPLY_E(ARR (A)) = ARR (E (A))

$ARR\_APPLY_E(ARR\_CAT\ (A, B)) = ARR\_CAT\ (ARR\_APPLY_E(A), ARR\_APPLY_E(B))$

A.4. Rules for Tuple Operators

The primitive binary tuple operator is TUP_CAT. The primitive unary tuple operators are TUP and TUP_EXTRACT. The derived unary tuple operator is π. This section assumes that tuples have named fields.

A.4.1. Binary Operators: Associativity, Commutativity, Distributivity

$TUP\_CAT\ (A, B) = TUP\_CAT\ (B, A)$

$TUP\_CAT\ (A, TUP\_CAT\ (B, C)) = TUP\_CAT\ (TUP\_CAT\ (A, B), C)$

A.4.2. Unary Operators: Pushing

$\pi_l(TUP\_CAT\ (A, B)) = TUP\_CAT\ (\pi_{l1}(A), \pi_{l2}(B))$; $l = l1 \cup l2$, $l1$ applies only to A, $l2$ applies only to B.

$\pi_{l1}(\pi_{l2}(A)) = \pi_{l1}(A)$; $l1 \subseteq l2$

$\pi_l(TUP(A)) = TUP\ (A)$; $l \neq \emptyset$.

$\pi_{l1}(TUP\_EXTRACT_f(A)) = \pi_{l1}(TUP\_EXTRACT_f(\pi_{l2}(A)))$; $f \in l2$.

$TUP\ (TUP\_EXTRACT_f(A)) = \pi_l(A)$; $l = \{ f \}$.

$TUP\_EXTRACT_f(TUP\ (A)) = A$

$TUP\_EXTRACT_f(TUP\_CAT\ (A, B)) = TUP\_EXTRACT_f(A)$; f is a field of A (modulo renaming).

$TUP\_EXTRACT_f(\pi_l(A)) = TUP\_EXTRACT_f(A)$

A.5. Rules for Reference Operators

The primitive unary reference operators are REF and DEREF.

$DEREF\ (REF\ (A)) = A$

The next rule is true only if an assumption is made about the implementation -- namely, that a dereferenced entity still carries its old identity around with it.

$REF\ (DEREF\ (A)) = A$

A.6. Rules Involving COMP

COMP is a unary operator whose input is any algebraic expression.

$COMP_P(E(A)) = E(COMP_{P1}(A))$; $P1(INPUT) = P(E(INPUT))$.

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

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

$COMP_P(\pi_l(A)) = \pi_l(COMP_P(A))$; P involves only fields in l.

$COMP_P(A) = COMP_P(\pi_l(A))$; P involves only fields in l.

APPENDIX B

A Proof of the Equipollence of EXCESS and its Algebra

Here we prove the equipollence of the EXCESS query language and the EXCESS algebra. The equipollence proof proceeds by first proving that every EXCESS statement can be expressed in the algebra and then proving that every algebra query can be expressed in EXCESS. We do not consider ADTs or updates in this proof. We do consider all retrievals, including range statements and user-defined EXCESS functions. Finally, we ignore the ability of EXCESS to rename the fields of result tuples, and assume a standard automatic field-name generation protocol on the part of the algebra. In principle, of course, there would be no difficulty in introducing a "rename" operator into the algebra, but this is not a central issue.

B.1. Reduction of Algebra to EXCESS

Theorem:

Any EXCESS algebra expression can be written in the EXCESS query language.

Proof:

The proof proceeds by induction on the number of operators in an algebraic expression E. An expression in the algebra consists of one or more named, top-level database objects and 0 or more operators. Constants, for the purposes of this discussion, appear only in predicates.

Base Case: 0 operators in E

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

```
retrieve (R)
```

Inductive Case: 1 or more operators in E

Assume that all expressions with < n operators ($n \geq 1$) have EXCESS counterparts. There are 24 cases to consider, one for each operator and one for generic set operations (all of which behave identically for our purposes).

1) $E = E1 \text{ -}_T E2$. 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). The EXCESS code for this query is:

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

2) E = E1 Ψ_T E2. In EXCESS this is written:

```
retrieve (x) from x in (E1 + E2):<T into E
```

3) E = E1 × E2. In EXCESS:

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

4) E = DE (E1). This has a straightforward translation:

```
retrieve unique (E1) into E
```

5) E = $GRP_{T,E1}$ (E2). Here E1 is not the stored result of some other query. Rather, we take advantage of our induction assumption in a different way to define an EXCESS *function* that computes E1. T is a type specifier that is either of the form "t" or "<t" for some type name t. The former indicates that only objects of type t are considered and the latter that only objects of type t or one of its subtypes are considered. The translation has 5 statements (but the `define type` statements may not be needed if these types are already named types in the database):

```
define type e2 : <type(E2)>

define e2 function I () returns e2
(
      retrieve (this)
)

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

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

retrieve ( I ( x by x.g from x in E2:T ) ) into E
```

Here, <type(E2)> refers to the type of E2 (which will of course be a set), <type(elt(E2))> refers to the type of the elements of E2, and <type(E1(elt(E2)))> refers to the type of the result of applying E1 to an element of E2. Each of these types is expressible in the syntax of EXCESS (this is trivial). <E1> is, by the inductive hypothesis, an EXCESS statement (or sequence of statements) that computes E1, with the appropriate parameter substitutions, etc. The identity function I is defined because the "by" keyword of EXCESS (which performs grouping) is only legal within some generic set function. Note that EXCESS functions (and their

corresponding "." notation) may be defined on and used with any EXTRA type, not just tuple types.

6) E = SET (E1). This uses the target list set constructor:

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

7) E = SET_COLLAPSE (E1). In EXCESS, a range variable always takes on values that are elements of the set corresponding to the range. In this example, the range corresponding to x is E1 while the range corresponding to y consists of all the elements of all the elements of the range of x:

```
retrieve (y) from x in E1, y in x into E
```

8) E = $SET\_APPLY_{T,E1}$ (E2). Here, T and E1 act as they did in Case 5 above:

```
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:T into E
```

9) E = TUP_CAT (E1, E2). In this example, note that if no targets are set- or array-valued, then no Cartesian product is formed--the result in such a special case is simply a tuple consisting of the targets in order.

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

10) E = TUP (E1). This uses the target list tuple constructor:

```
retrieve ( (E1) ) into E
```

11) E = $TUP\_EXTRACT_f$ (E1). This (like Case 9) makes use of a special case: if there is only 1 target in the list and it is single-valued, that target itself is the result of the query.

```
retrieve (E1.f) into E
```

12) E = $SUBARR_{l,h}$ (E1). This has a direct analog in EXCESS:

```
retrieve ( E1[l..h] ) into E
```

13) E = ARR_CAT_T (E1, E2). Here T is the same as in Cases 1 and 2. The semantics of "+" in EXCESS are that it maintains order in the result only if both of its inputs are ordered.

```
retrieve (x) from x in (E1 + E2):<T into E
```

14) E = ARR (E1). Here we use the array target list constructor:

```
retrieve ( [E1] ) into E
```

15) E = $ARR\_EXTRACT_e$ (E1). This is similar to Case 11 above:

```
retrieve ( E1[e] ) into E
```

16) E = ARR_COLLAPSE (E1). This is similar to Case 7 above:

```
retrieve (y) from x in E1, y in x into E
```

17) E = ARR_DE (E1). This is similar to Case 4 above, but order is preserved:

```
retrieve unique (E1) into E
```

18) E = ARR_CROSS (E1, E2). This is an order-preserving equivalent of Case 3 above:

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

19) E = ARR_DIFF_T (E1, E2). This is identical to the set difference translation in Case 1:

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

20) E = $ARR\_APPLY_{T,E1}$ (E2). This also has the same translation as its set-based counterpart, but the order of the elements is preserved:

```
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:T into E
```

21) E = DEREF (E1). Note that objects explicitly mentioned in the target list are implicitly dereferenced by the "retrieve" command:

```
retrieve ( E1 ) into E
```

22) E = REF (E1). In this case, the "<>" syntax is part of the EXCESS language, not part of the meta-language as in Case 20 and others. It is the target list constructor syntax for the "ref" type constructor:

```
retrieve ( <E1> ) into E
```

23) E = $COMP_P$ (E1). We prove this case by induction on the number (n) of propositional logic connectives in P.

Base Case: n = 0. Then P is of the form E2 <op> E3. The translation is as follows:

```
retrieve (E1) into E where E2 <EXCESS_op> E3
```

where <EXCESS_op> is determined from <op> as in Figure B.1.

Inductive Case: Assume that predicates with n-1 connectives can be expressed. Then there are 2 cases for proving that a predicate with n connectives can be expressed:

i) P is of the form P1 $\wedge$ P2. Then, by the induction hypothesis, we have `retrieve (E1) into E where <P1>` and `retrieve (E1) into E where <P2>`, where <P1> and <P2> represent the EXCESS forms of the predicates P1 and P2. Thus we can construct the following:

```
retrieve (E1) into E where <P1> and <P2>
```

ii) P is of the form $\neg$ P1. Then by the induction hypothesis we have `retrieve (E1) into E where <P1>`, where <P1> is as in Case 1 above. Then we can construct:

```
retrieve (E1) into E where not <P1>
```

This proves the inductive case. Since the base and inductive cases hold, we see that any COMP query expressible in the algebra is expressible in EXCESS, concluding Case 23 of the main proof. □

24) E = G (E1), where G is some previously defined generic set/array function (which must have EXTRA types as input and output). This has a simple translation:

```
retrieve ( G (E1) ) into E
```

<op>	<EXCESS_op>
=	is when comparing OIDs = otherwise
$\in$	in
$\subseteq$	<=
$\sqsubseteq$	<<=
$\leq$	<=

Figure B.1: Comparison operators

This concludes the inductive case, completing the proof that any query of the algebra can be expressed in EXCESS. □

B.2. Reduction of EXCESS to algebra

The syntax of an EXCESS retrieval query is as follows:

retrieve [unique] '(' **<final_res>** ')' [into **<object>**]

[**<from_clause>**] [where **<where_clause>**]

<from_clause> ::= from (**<from>** ,)* **<from>**

<from> ::= **<var>** in **<target>** [**<set_op>** **<target>**] [**<type-spec>**]

<type-spec> ::= (: | :<) **<type>**

<set_op> ::= +

| *

| -

<target> ::= (**<t_component>** .)* **<t_component>**

<t_component> ::= **<object>** ('[' **<int_const>** [: **<int_const>**] ']')*

<final_res> ::= **<res_list>**

| '{' **<res_list>** '}'

| '[' **<res_list>** ']'

| '<' **<res_list>** '>'

| '(' **<res_list>** ')'

<res_list> ::= (**<result>** ,)* **<result>**

<result> ::= (**<r_component>** .)* **<r_component>**

| **<aggregate>**

<aggregate> ::= **<agg-op>** '(' **<attr_list>** [over **<attr_list>**] [by **<attr_list>**]

[**<from>**] [where **<where_clause>**] ')'

<r_component> ::= (<object> | <var>) ('[' <int_const> [: <int_const>] ']')*

<where_clause> ::= <where_clause> (and | or) <where_clause>

| not <where_clause>

| <univ_clause>

| <ex_clause>

| <compare_clause>

| '(' <where_clause> ')'

<univ_clause> ::= '(' forall <var> in <result> : <where_clause>

<ex_clause> ::= '(' forsome <var> in <result> : <where_clause>

<compare_clause> ::= <result> <comp_op> (<result> | <const>)

The proof will proceed bottom-up. First we prove that any target can be expressed in the algebra. We use this in the proofs that any result list and any "from" component can also be expressed in the algebra. We then constructively prove that every where clause can be expressed as a predicate to the algebra's COMP operator. Finally, we use these results to prove that an entire retrieve statement is expressible in the algebra. When they are obvious from the context, we omit subscripts from the algebraic operators.

We first direct our attention to the "from" clause, and prove lemmas concerning t_components, targets, and then entire from clauses.

LEMMA 1: A t_component has a corresponding algebra expression.

PROOF: By induction on the number of array subscripting operations appearing in the component.

Base Case: t_component = C (no array subscripts). In this case the component is an identifier and must be the name of a top-level EXTRA object. Thus the algebra query to retrieve this object is simply C.

Inductive Case: (i) t_component = C [...] ... [...] [x:y]. By the inductive assumption, everything in t_component preceding the last subscript can be expressed in the algebra by some expression CE. If CE is a reference, it should be replaced by DEREF(CE). The query for t_component then becomes $SUBARR_{x,y}$ (CE). (ii) t_component = C [...] ... [...] [x]. By the inductive assumption, everything in t_component preceding the last subscript can be expressed in the algebra by some expression CE. If CE is a reference, it should be replaced by DEREF(CE). The query for t_component then becomes $ARR\_EXTRACT_x$(CE). □

LEMMA 2: A target whose last component has no subscripts has a corresponding algebra expression.

PROOF: By induction on the number of components in the target.

Base Case: target = C (C is a component). An expression exists for this, by Lemma 1.

Inductive Case: target = $C_1....C_n$. C. By the induction hypothesis and Lemma 1, let E1 be the algebra expression corresponding to $C_1....C_n$. There are six cases, dependent on the type of E1:

(i) E1 is a tuple. Query is $TUP\_EXTRACT_C(E1)$

(ii) E1 is a reference to a tuple. Query is $TUP\_EXTRACT_C(DEREF(E1))$

(iii) E1 is a set of tuples. If C is a set-valued attribute, then the query is $SET\_COLLAPSE(SET\_APPLY_{TUP\_EXTRACT_C}(E1))$. Otherwise, the query is $SET\_APPLY_{TUP\_EXTRACT_C}(E1)$.

(iv) E1 is a set of references to tuples. If C is a set-valued attribute, then the query is $SET\_COLLAPSE(SET\_APPLY_{TUP\_EXTRACT_C(DEREF)}(E1))$. Otherwise, the query is $SET\_APPLY_{TUP\_EXTRACT_C(DEREF)}(E1)$.

(v) E1 is an array of tuples. If C is an array-valued attribute, then the query is $ARR\_COLLAPSE(ARR\_APPLY_{TUP\_EXTRACT_C}(E1))$. Otherwise, the query is $ARR\_APPLY_{TUP\_EXTRACT_C}(E1)$.

(vi) E1 is an array of references to tuples. If C is an array-valued attribute, then the query is $ARR\_COLLAPSE(ARR\_APPLY_{TUP\_EXTRACT_C(DEREF)}(E1))$. Otherwise, the query is $ARR\_APPLY_{TUP\_EXTRACT_C(DEREF)}(E1)$. □

LEMMA 3: A target has a corresponding algebra expression.

PROOF: The only case not covered by Lemma 2 is that in which the last component of a target has array subscripts. Thus the proof here is similar to the proof of the inductive case of Lemma 1. (i) Assume that target = $C_1....C_n$. C [x:y]. Then, by Lemma 2, we know that some expression E1 corresponds to $C_1....C_n$. C. If E1 is a reference, it should be replaced by DEREF(E1). Thus the query becomes $SUBARR_{x,y}$ (E1). (ii) Assume that target = $C_1....C_n$. C [x]. Then, by Lemma 2, we know that some expression E1 corresponds to $C_1....C_n$. C. If E1 is a reference, it should be replaced by DEREF(E1). Thus the query becomes $ARR\_EXTRACT_x(E1)$. □

LEMMA 4: A variable declared in the "from" clause has a corresponding algebra expression.

PROOF: In the following, T, T1, and T2 represent targets, with corresponding algebra queries E, E1, and E2. There are five cases:

(i) variable = T. Then by Lemma 3 we have an expression for this.

In cases (ii)-(v), if E, E1, or E2 is a reference, it should be replaced by DEREF(E), DEREF(E1), or DEREF(E2) respectively.

(ii) variable = T1 + T2. Then if either T1 or T2 represents a set, the query is E1 $\uplus$ E2. Otherwise the query is ARR_CAT (E1,E2).

(iii) variable = T1 * T2. Then if either T1 or T2 represents a set, the query is E1 - (E1 - E2). Otherwise the query is ARR_DIFF (E1, ARR_DIFF (E1, E2)).

(iv) variable = T1 - T2. Then if either T1 or T2 represents a set, the query is E1 - E2. Otherwise the query is ARR_DIFF (E1, E2).

(v) variable = T <type-spec>. Then if T represents a set, the query is $\text{SET\_APPLY}_{\text{<type-spec>,INPUT}}(\text{E})$. Otherwise the query is $\text{ARR\_APPLY}_{\text{<type-spec>,INPUT}}(\text{E})$. □

LEMMA 5: An r_component has a corresponding algebra expression.

PROOF: By induction on the number of array subscripting operations appearing in the component.

Base Case: r_component = C (no array subscripts). In this case the component is either the name of a top-level EXTRA object, in which case the query is C, or is the name of a variable from the from clause, in which case an expression exists by Lemma 4.

Inductive Case: (i) r_component = C [...] ... [...] [x:y]. By the inductive assumption, everything in r_component preceding the last subscript can be expressed in the algebra by some expression CE. If CE is a reference, it should be replaced by DEREF(CE). The query for r_component then becomes $\text{SUBARR}_{x,y}$ (CE). (ii) r_component = C [...] ... [...] [x]. By the inductive assumption, everything in t_component preceding the last subscript can be expressed in the algebra by some expression CE. If CE is a reference, it should be replaced by DEREF(CE). The query for r_component then becomes ARR_EXTRACT_{x}(CE). □

LEMMA 6: A result whose last component has no subscripts has a corresponding algebra expression.

PROOF: By induction on the number of components in the result.

Base Case: result = C (C is a component). An expression exists for this, by Lemma 5.

Inductive Case: result = $C_1....C_n$. C. By the induction hypothesis and Lemma 5, let E1 be the algebra expression corresponding to $C_1....C_n$. There are six cases, dependent on the type of E1:

(i) E1 is a tuple. Query is $TUP\_EXTRACT_C(E1)$

(ii) E1 is a reference to a tuple. Query is $TUP\_EXTRACT_C(DEREF(E1))$

(iii) E1 is a set of tuples. The query is $SET\_APPLY_{TUP\_EXTRACT_C}(E1)$.

(iv) E1 is a set of references to tuples. The query is $SET\_APPLY_{TUP\_EXTRACT_C(DEREF)}(E1)$.

(v) E1 is an array of tuples. The query is $ARR\_APPLY_{TUP\_EXTRACT_C}(E1)$.

(vi) E1 is an array of references to tuples. The query is $ARR\_APPLY_{TUP\_EXTRACT_C(DEREF)}(E1)$. □

LEMMA 7: A result has a corresponding algebra expression.

PROOF: The only case not covered by Lemma 6 is that in which the last component of a result has array subscripts. Thus the proof here is similar to the proof of the inductive case of Lemma 1. (i) Assume that result = $C_1....C_n$. C [x:y]. Then, by Lemma 6, we know that some expression E1 corresponds to $C_1....C_n$. C. If E1 is a reference, it should be replaced by DEREF(E1). Thus the query becomes $SUBARR_{x,y}$ (E1). (ii) Assume that result = $C_1....C_n$. C [x]. Then, by Lemma 6, we know that some expression E1 corresponds to $C_1....C_n$. C. If E1 is a reference, it should be replaced by DEREF(E1). Thus the query becomes $ARR\_EXTRACT_x(E1)$. □

LEMMA 8: A result list has a corresponding algebra expression.

PROOF: Let res_list = R_1, R_2, ..., R_n and let E_i be the algebraic expression corresponding to R_i (by Lemma 7). There are three cases to consider.

(i) If none of the R_i represent sets or arrays, then the query is TUP_CAT (TUP (E_1), TUP_CAT (TUP (E_2), ... TUP_CAT (TUP (E_{n-1}), TUP (E_n)))).

(ii) If at least one of the R_i represents a set, then the query is $E_1 \times (E_2 \times (E_3 ... \times (E_{n-1} \times E_n)))$. If any of the E_i is a reference to a set, it should be replaced by DEREF (E_i).

(iii) Otherwise, the query is ARR_CROSS (E_1, ARR_CROSS (E_2, ... ARR_CROSS (E_{n-1}, E_n))). If any of the E_i is a reference to an array, it should be replaced by DEREF (E_i). □

LEMMA 9: A final result has a corresponding algebra expression.

PROOF: Assume (using Lemma 8) that the algebraic expression corresponding to the result list R is E. There are four cases:

(i) final_res = {R}. Then the query is SET (E).

(ii) final_res = [R]. Then the query is ARR (E).

(iii) final_res = (R). Then the query is TUP (E).

(iv) final_res = <R>. Then the query is REF (E). □

LEMMA 10: A where clause can be translated into a predicate in the predicate sublanguage of the algebra's COMP operator.

PROOF: By induction on the number of predicate calculus constructs appearing in the where clause.

Base Case: A simple compare clause, T1 <op> T2. By Lemmas 4 and 7, T1 and T2 have corresponding algebra expressions E1 and E2. There are five cases (the other comparators are redundant):

(i) T1 is T2. Then the predicate is E1 = E2.

(ii) T1 = T2. Then the predicate is DEREF(E1) = DEREF(E2) if T1 and T2 represent references and E1 = E2 otherwise.

(iii) T1 in T2. Then the predicate is E1 $\in$ E2 if T2 represents a set and E1 $\sqsubseteq$ E2 otherwise. If E2 is a reference to a set or an array, it is replaced by DEREF (E2). If E1 is a reference, it is replaced by DEREF (E1) only if the type of the members of E2 is not a reference type.

(iv) T1 <= T2. Then the predicate is E1 $\subseteq$ E2 if T2 represents a set and E1 E2 otherwise. If either E1 or E2 is a reference, it is to be replaced by DEREF (E1) or DEREF (E2), respectively.

(v) T1 <<= T2. Then the predicate is E1 $\sqsubseteq$ E2. If either E1 or E2 is a reference, it is to be replaced by DEREF (E1) or DEREF (E2), respectively.

Inductive Case: The "forsome" syntax and disjunction are redundant, so we have three cases to consider:

(i) forall x in S1 : S2. Then by Lemma 7 and the inductive hypothesis we have expressions E1 and E2 corresponding to S1 and S2. The predicate is $\text{SET\_APPLY}_{\text{COMP}_{E2}}$(E1). If E1 is a reference, it is replaced by DEREF (E1).

(ii) S1 and S2. By the inductive hypothesis we have expressions E1 and E2 corresponding to S1 and S2. Thus the predicate is E1 $\wedge$ E2.

(iii) not S. By the inductive hypothesis we have expression E corresponding to S. The predicate is $\neg$ E1. □

LEMMA 11: An EXCESS retrieval query without a "unique" clause or aggregation has a corresponding algebra expression.

PROOF: The final query result is obtained by applying the predicate of the where clause to the result built up from the appropriate range variables, and then extracting from this the desired data. This turns out to be much more complicated than the equivalent process for a relational query processor because of the extended "." notation permitted in the result list. It becomes necessary to relate elements of the result list whose data comes from the same source. The details of this are described below.

By the preceding lemmas, we know that each variable V_i in the "from" clause has a corresponding query, call it QV_i. We also know that every result R_i in the result list has a corresponding query, call it QR_i.

Now, in order to ensure that the results in the result list are appropriately correlated, we need an extended notion of "range variable". We will treat every component of every result in the result list as a range variable and will designate one of the components as the *primary* component. As an example of the necessity of this notion, consider the following result list: "(Emps.kids.name, Emps.kids.age)". Clearly the desired result is to return a set of pairs, one for each child. Thus the primary component (or range variable) for each of the results in the result list should be "Emps.kids". If it were "Emps", the query would return, for each employee, a set of names and a set of ages, but the relationships between the kids and their ages would be lost.

For two results R1 and R2, let the greatest common component (GCC) be the greatest common prefix of the results. We identify the primary component of each result in the result list using the following algorithm:

```
FOR EACH result R1 IN result list DO
    FOR EACH result R2 IN result list DO
        G = GCC (R1, R2)
        IF (G > primary_component (R1))
            THEN primary_component (R1) := G
```

Note that a result list with n results may have fewer than n unique primary components. Next we correlate the data retrieved by all results that have the same primary component. For each unique primary component, PC_i, in the result list, we do the following:

(i) Suppose PC_i is the primary component of results R_{r_1}, ..., R_{r_n}.

(ii) Since each of the R_{r_i} has the same primary component, each of them shares a common algebraic subexpression corresponding to this primary component. If we remove this common subexpression from each of the QR_{l_i}, we are left with the partial algebraic expressions PQR_{l_i}, each of which is distinct and takes as input the result of the common algebraic subexpression.

(iii) Construct a partial algebraic expression $APCE_i$ as follows: If n=1, the partial expression is simply PQR_{l_1}. If n > 1, the partial expression is TUP_CAT (TUP(PQR_{l_1}), ... TUP_CAT (TUP($PQR_{l_{n-1}}$), TUP(PQR_{l_n}))).

We now have an expression that can be applied to each element of the range of a primary component to retrieve the appropriate data from that component. Before doing this, however, the predicate of the "where" clause must be applied. By Lemma 10, we can construct such a predicate in the algebra. We will apply this predicate to an algebraic expression PE constructed using the primary components of the result list as follows (recall that Lemma 7 tells us that every primary component PC has a corresponding algebra expression PCE):

(i) If the result list has only one primary component, then PE = PCE.

(ii) Else if none of the PC_i is set- or array-valued, PE = TUP_CAT (TUP(PCE_1), ... TUP_CAT (TUP(PCE_{m-1}), TUP(PCE_m))) (assuming that there are m unique primary components).

(iii) Else if at least one of the PC_i is set-valued, PE = $PCE_1 \times (PCE_2 \times ... \times (PCE_{m-1} \times PCE_m))$. If any of the PCE_i is a reference, it must be replaced by DEREF (PCE_i).

(iv) Else PE = ARR_CROSS (PCE_1, ARR_CROSS (PCE_2, ... ARR_CROSS (PCE_{m-1}, PCE_m))). If any of the PCE_i is a reference, it must be replaced by DEREF (PCE_i).

Now we are ready to apply the predicate P of the "where" clause to PE. Note that some data used in P may not come from PE, but by Lemma 10 this does not matter. There are three cases to consider in constructing PQ, the expression corresponding to the where clause applied to PE:

(i) PE represents a value, a tuple or a reference. Then PQ = $COMP_P$(PE).

(ii) PE represents a set. Then PQ = $SET\_APPLY_{COMP_{P1}}$ (PE), where P1 is P embellished with the appropriate TUP_EXTRACT operations to retrieve data from the Cartesian product of the primary components.

(iii) PE represents an array. Then PQ = $ARR\_APPLY_{COMP_{P1}}$(PE), where P1 is P embellished with the appropriate TUP_EXTRACT operations to retrieve data from the array product (ARR_CROSS) of the primary components.

At this point we are ready to apply the $APCE_i$ to PQ to arrive at the final result of the query. There are three cases to consider:

(i) PQ represents a value, a tuple or a reference. Then there is only one primary component, and all of its corresponding results are single-valued. The final query is then $APCE_1$(PQ).

(ii) PQ represents a set. Then each element of the set has a portion that corresponds to data from each primary component. The final query is then SET_APPLY_E(PQ), where E is an expression that applies each $APCE_i$(PQ) to its corresponding component using TUP_EXTRACT (if necessary) and concatenates the results into a tuple using TUP and TUP_CAT (the specifics are trivial and omitted).

(iii) PQ represents an array. Then each element of the array has a portion that corresponds to data from each primary component. The final query is then ARR_APPLY_E(PQ), where E is an expression that applies each $APCE_i$(PQ) to its corresponding component using TUP_EXTRACT (if necessary) and concatenates the results into a tuple using TUP and TUP_CAT (the specifics are trivial and omitted). □

LEMMA 12: An EXCESS retrieval query without a "unique" clause has a corresponding algebra expression.

PROOF: If the query has no aggregation, Lemma 11 applies, and we are done. Recall that each generic set operator G (such as "min", "max", etc.) implicitly defines a new algebraic operator. The primary component of an aggregate result is defined to be the variable in its from clause (if it has one) or the primary component of the required attribute list specified inside the aggregate. If this attribute list has more than one primary component, the primary component of the aggregate result becomes the Cartesian product (or ARR_CROSS if all are arrays) of the primary components of the attribute list. By preceding lemmas, let PCE be the algebra expression corresponding to the primary component of the aggregate result. In the remainder of the proof of this lemma, note that if either E or the elements of the collection it represents are references, they must be dereferenced before the discussion in the following paragraphs can apply.

If no "over" or "by" clause is present, the expression inside the aggregate is a simple retrieval query, thus by Lemma 11 we have a corresponding algebra expression E that can be applied to the primary component of the aggregate result to obtain the data to which the aggregate function G itself is applied. Since G is also an algebra

operator, G(E) is an expression, and can be used as one of the PQR_{l_i} in Lemma 11, QED.

If an "over" clause is present, the construction is similar to that of the previous paragraph, except that the expression E is replaced by DE ($SET\_APPLY_{\pi_l}$(E)), where l is the list of attributes in the "over" clause. If the primary component of the aggregate result is an array, then we use ARR_APPLY instead of SET_APPLY.

If a "by" clause is present, we need to use the algebra's GRP operator. Here the PQR_{l_i} becomes SET_APPLY_G ($GRP_{\pi_{by}}$(E)), where "by" is the list of attributes specified in the "by" clause and E is the appropriate expression from one of the two preceding paragraphs. □

LEMMA 13: An EXCESS retrieval query has a corresponding algebra expression.

PROOF: If the query has no "unique" clause, Lemma 12 applies. If the query U is "retrieve unique Q", by Lemma 11 we have some algebra expression E that represents Q. If the type of the result of E is a tuple, a reference, or a simple value, then the expression for the entire query U is E. If the type of E is "set", then the algebra expression is DE (E). Otherwise the algebra expression is ARR_DE (E). □

In all of the proofs of these lemmas we have made the assumption that appropriate use of the INPUT symbol is made in the SET_APPLY, ARR_APPLY, GRP, and COMP operators to achieve the correct variable bindings. This is clearly always possible. We have omitted this detail to make the proof more accessible. This concludes the reduction of EXCESS to the algebra. □

Since we have now proved that the algebra reduces to EXCESS and that EXCESS reduces to the algebra, we have proved that the algebra and EXCESS are equivalent in expressive power. □

REFERENCES

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

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

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

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

[Abit90] S. Abiteboul, E. Simon, and V. Vianu, "Non-deterministic languages to express deterministic transformations", *Proc. ACM PODS Conf.fR, Nashville, TN, April 1990.*

[Agra87] R. Agrawal, "Alpha: An Extension of Relational Algebra to Express a Class of Recursive Queries", *Proc. IEEE Intl. Conf. on Data Engineering*, Los Angeles, CA, 1987.

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

[Alba91] A. Albano, G. Ghelli, and R. Orsini, "Objects for a Database Programming Language", in *Bulk Types & Persistent Data: The Third International Workshop on Database Programming Languages*, ed. P. Kanellakis and J. Schmidt, Nafplion, Greece, Morgan Kaufmann, August 1991.

[Andr87] T. Andrews and C. Harris, "Combining Language and Database Advances in an Object-Oriented Development Environment," *Proc. 2nd OOPSLA Conf.*, Orlando, FL, 1987.

[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.

[Astr76] M. Astrahan, M. Blasgen, D. Chamberlin, K. Eswaran, J. Gray, P. Griffiths, W. King, R. Lorie, P. McJones, J. Mehl, G. Putzolu, I. Traiger, B. Wade, and V. Watson, "System R: A Relational Approach to Database Management", *ACM TODS* 1(2), June 1976.

[Atki91] M. Atkinson, C. Lecluse, P. Philbrow, and P. Richard, "Design Issues in a Map Language", in *Bulk Types & Persistent Data: The Third International Workshop on Database Programming Languages*, ed. P. Kanellakis and J. Schmidt, Nafplion, Greece, Morgan Kaufmann, August 1991.

[Banc88] F. Bancilhon, G. Barbedette, V. Benzaken, C. Delobel, S. Gamerman, C. Lecluse, P. Pfeffer, P. Richard, and F. Velez, "The Design and Implementation of O_2 an Object-Oriented Database System," Tech. Report 20-88, Altair, April 1988.

[Banc87] F. Bancilhon, T. Briggs, S. Khoshafian, and P. Valduriez, "FAD, a Powerful and Simple Database Language," *Proc. VLDB Conf.*, Brighton, England, 1987.

[Banc86] F. Bancilhon and S. Khoshafian, "A Calculus for Complex Objects," *Proc. ACM 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.

[Bato87] D. Batory, "Principles of Database Management System Extensibility," *IEEE Database Eng. Bulletin* 10, June 1987.

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

[Bloo87] Bloom, T., and Zdonik, S., "Issues in the Design of Object-Oriented Database Programming Languages," *Proc. 2nd OOPSLA Conf.*, Orlando, FL, 1987.

[Bune91] P. Buneman and A. Ohori, "A Type System that Reconciles Classes and Extents", in *Bulk Types & Persistent Data: The Third International Workshop on Database Programming Languages*, ed. P. Kanellakis and J.

Schmidt, Nafplion, Greece, Morgan Kaufmann, August 1991.

[Care88a] M. Carey, D. DeWitt, G. Graefe, D. Haight, J. Richardson, D. Schuh, E. Shekita, and S. Vandenberg, "The EXODUS Extensible DBMS Project: An Overview", in *Readings in Object-Oriented Database Systems*, ed. S. Zdonik and D. Maier, Morgan Kaufmann, San Mateo, CA, 1990.

[Care88b] M. Carey, D. DeWitt, and S. Vandenberg, "A Data Model and Query Language for EXODUS," *Proc. ACM SIGMOD Conf.*, Chicago, Illinois, 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.

[Cham75] D. Chamberlin, J. Gray, and I. Traiger, "Views, Authorization, and Locking in a Relational Database System," *Proc. Nat'l. Computer Conf.*, Anaheim, CA, 1975.

[Chan88] A. Chandra, "Theory of Database Queries", *Proc. ACM PODS Conf.*, 1988.

[Chen76] P. Chen, "The Entity-Relationship Model — Toward a Unified View of Data," *ACM TODS* 1(1), March 1976.

[Chen84] P. Chen, "An Algebra for a Directional Binary Entity-Relationship Model", *Proc. IEEE Intl. Conf. on Data Engineering*, Los Angeles, CA, April 1984.

[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," *CACM* 13(6), June 1970.

[Codd79] E. Codd, "Extending the Relational Model to Capture More Meaning," *ACM TODS* 4(4), Dec. 1979.

[Codd80] E. Codd, "Data Models in Database Management," *Proc. Workshop on Data Abstraction, Databases, and Conceptual Modeling"*, Pingree Park, Colorado, 1980.

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

[Cope84] G. Copeland and D. Maier, "Making Smalltalk a Database System," *Proc. ACM SIGMOD Conf.*, Boston, MA, 1984.

[Dada86] P. Dadam, K. Kuespert, F. Andersen, H. Blanken, R. Erbe, J. Guenauer, V. Lum, P. Pistor, and G. Walch, "A DBMS Prototype to Support Extended NF^2 Relations: An Integrated View of Flat Tables and Hierarchies," *Proc. ACM SIGMOD Conf.*, Washington, DC, 1986.

[Dani91] S. Daniels, G. Graefe, T. Keller, D. Maier, D. Schmidt, and B. Vance, "Query Optimization in Revelation, an Overview", *IEEE Data Engineering Bulletin*, 14(2), June 1991.

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

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

[Desh88] A. Deshpande and D. Van Gucht, "An Implementation for Nested Relational Databases," *Proc. VLDB Conf.*, Los Angeles, CA, 1988.

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

[Ditt86] K. Dittrich, "Object-Oriented Database Systems: The Notion and the Issues," *Proc. Int'l. Workshop on Object-Oriented Database Systems*, Pacific Grove, CA, Sept. 1986.

[Done70] J. Doner, "Tree Acceptors and Some of their Applications", *J. Computer and System Sciences* 4, 1970.

[Dorn78] L. L. Dornhoff and F. E. Hohn, *Applied Modern Algebra*, Macmillan, New York, 1978.

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

[Fish89] D. Fishman, J. Annevelink, E. Chow, T. Connors, J. Davis, W. Hasan, C. Hoch, W. Kent, S. Leichner, P. Lyngbaek, B. Mahbod, M. Neimat, T. Risch, M. Shan, and W. Wilkinson, "Overview of the Iris DBMS", in *Object-Oriented Concepts, Databases, and Applications*, ed. W. Kim and F. Lochovsky, Addison-Wesley, Reading, MA, 1989.

[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.

[Fren90] J. French, A. Jones, and J. Pfaltz, "Summary of the Final Report of the NSF Workshop on Scientific Database Management", *SIGMOD Record* 19, 1990.

[Fren91] K. Frenkel, "The Human Genome Project and Informatics", *CACM* 34, 1991.

[Gold83] A. Goldberg and D. Robson, *Smalltalk-80: The Language and its Implementation*, Addison-Wesley, Reading, MA, 1983.

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

[Grae89] G. Graefe, ed., *Workshop on Database Query Optimization*, CSE Tech. Report 89-005, Oregon Graduate Center, Portland, Oregon, May 30, 1989.

[Grae87] G. Graefe and D. DeWitt, "The EXODUS Optimizer Generator," *Proc. ACM 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.

[Gray84] P. Gray, *Logic, Algebra, and Databases*, Ellis Horwood Ltd., West Sussex, England, 1984.

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

[Gutt85] J. Guttag, J. Horning, and J. Wing, "The Larch Family of Specification Languages", *IEEE Software*, 2(5), September 1985.

[Gyss89] M. Gyssens, J. Paredaens, and D. Van Gucht, "A Grammar-Based Approach Towards Unifying Hierarchical Data Models", *Proc. ACM SIGMOD Conf.*, Portland, Oregon, 1989.

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

[Hoff82] C. Hoffmann and M. O'Donnell, "Pattern Matching in Trees", *JACM* 29(1), January 1982.

[Horn87] M. Hornick and S. Zdonik, "A Shared, Segmented Memory System for an Object-Oriented Database," *ACM Trans. Office Info. Sys.* 5(1), Jan. 1987.

[Houb87] G. J. Houben and J. Paredaens, "The R^2-Algebra: An Extension of an Algebra for Nested Relations," Tech. Report 87/20, Dept. of Math. and Computing Sci., Computing Sci. Section, Eindhoven Univ. of Tech., December 1987.

[Hull87] R. Hull, "A Survey of Theoretical Research on Typed Complex Database Objects", in *Databases*, ed. J. Paredaens, Academic Press, London, 1987.

[Hull88] R. Hull, "Four Views of Complex Objects: A Sophisticate's Introduction", draft, Dept. of Computer Science, Univ. of Southern California, Los Angeles, California, May 1988.

[Hull89a] R. Hull and J. Su, "On Accessing Object-Oriented Databases: Expressive Power, Complexity, and Restrictions", *Proc. ACM SIGMOD Conf.*, Portland, Oregon, 1989.

[Hull89b] R. Hull and J. Su, "Untyped Sets, Invention, and Computable Queries", *Proc. ACM PODS Conf.*, 1989.

[Hull90] R. Hull and M. Yoshikawa, "ILOG: Declarative creation and manipulation of object identifiers", *Proc. VLDB Conf.*, Brisbane, Australia, August 1990.

[IDM500] IDM 500 Software Reference Manual, version 1.4, Britton-Lee Inc., Los Gatos, CA.

[Jaes82a] G. Jaeschke, "An Algebra of Power Set Type Relations," Tech. Report 82.12.002, IBM Heidelberg Scientific Center, Dec. 1982.

[Jaes85a] G. Jaeschke, "Recursive Algebra for Relations with Relation Valued Attributes," Tech. Report 85.03.002, IBM Heidelberg Scientific Center, March 1985.

[Jaes85b] G. Jaeschke, "Nonrecursive Algebra for Relations with Relation Valued Attributes," Tech. Report 85.03.001, IBM Heidelberg Scientific Center, March 1985.

[Jaes82b] G. Jaeschke and H.-J. Schek, "Remarks on the Algebra of Non First Normal Form Relations," *Proc. ACM PODS Conf.*, Los Angeles, CA, 1982.

[Jark84] M. Jarke and J. Koch, "Query Optimization in Database Systems," *ACM Comp. Surveys* 16(2), June 1984.

[Karp72] R. Karp, R. Miller, and A. Rosenberg, "Rapid Identification of Repeated Patterns in Strings, Trees, and Arrays", *Proc. 4th Annual ACM Symp. on Theory of Comp.*, Denver, CO, 1972.

[Kemp87] A. Kemper and M. Wallrath, "An Analysis of Geometric Modeling in Database Systems," *ACM Comp. Surveys* 19(1), March 1987.

[Kent79] W. Kent, "Limitations of Record-Based Information Models," *ACM TODS* 4(1), March 1979.

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

[Khos86] S. Khoshafian and G. Copeland, "Object Identity," *Proc. 1st OOPSLA Conf.*, Portland, OR, Sept. 1986.

[Khos87] S. Khoshafian and P. Valduriez, "Sharing, Persistence, and Object Orientation: A Database Perspective," Tech. Report DB-106-87, MCC, April 1987.

[Kim88] W. Kim, "A Model of Queries for Object-Oriented Databases", Tech. Report ACA-ST-365-88, MCC, Austin, TX, November 1988.

[Kim87] W. Kim, J. Banerjee, H.-T. Chou, J. Garza, and D. Woelk, "Composite Object Support in an Object-Oriented Database System," *Proc. 2nd OOPSLA Conf.*, Orlando, FL, 1987.

[King81] J. King, "QUIST: A System for Semantic Query Optimization in Relational Databases", *Proc. VLDB Conf.*, Cannes, France, August 1981.

[Klau85] A. Klausner and N. Goodman, "Multirelations — Semantics and Languages," *Proc. VLDB Conf.*, Stockholm, Sweden, 1985.

[Klug82] A. Klug, "Equivalence of Relational Algebra and Relational Calculus Query Languages Having Aggregate Functions," *JACM* 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.

[Kort91] H. Korth and A. Silberschatz, *Database System Concepts*, second edition, McGraw-Hill, New York, 1991.

[Knut81] D. Knuth, *The Art of Computer Programming, Volume 2: Seminumerical Algorithms*, 2nd edition, Addison-Wesley, Reading, Massachusetts, 1981.

[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.

[Land91] E. Lander, R. Langridge, and D. Saccocio, "Mapping and Interpreting Biological Information", *CACM* 34, 1991.

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

[Leun93] T. Leung, G. Mitchell, B. Subramanian, B. Vance, S. Vandenberg, and S. Zdonik, "The AQUA Data Model and Algebra", *Fourth International Workshop on Database Programming Languages*, New York, New York, 1993, to appear.

[Liu77] C. L. Liu, *Elements of Discrete Mathematics*, McGraw-Hill, New York, 1977.

[Lohm83] G. Lohman, J. Stoltzfus, A. Benson, M. Martin, and A. Cardenas, "Remotely-Sensed Geophysical Databases: Experience and Implications for Generalized DBMS," *Proc. ACM SIGMOD Conf.*, San Jose, CA, 1983.

[Maie86a] D. Maier, "A Logic for Objects," Tech. Report CS/E-86-012, Oregon Grad. Center, Beaverton, Oregon, Nov. 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.

[Maie86c] D. Maier, J. Stein, A. Otis, and A. Purdy, "Development of an Object-Oriented DBMS," *Proc. 1st OOPSLA Conf.*, Portland, OR, 1986.

[Maki77] A. Makinouchi, "A Consideration on Normal Form of Not-Necessarily-Normalized Relations in the Relational Data Model", *Proc. VLDB Conf.*, Tokyo, Japan, 1977.

[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.

[Mits89] B. Mitschang, "Extending the Relational Algebra to Capture Complex Objects", *Proc. VLDB Conf.*, Amsterdam, The Netherlands, 1989.

[Mylo80] J. Mylopoulos, P. Bernstein, and H. Wong, "A Language Facility for Designing Database-Intensive Applications", *ACM TODS* 5(2), June 1980.

[Nixo87] B. Nixon, L. Chung, D. Lauzon, A. Borgida, J. Mylopoulos, and M. Stanley, "Implementation of a Compiler for a Semantic Data Model: Experiences with TAXIS," *Proc. ACM SIGMOD Conf.*, San Francisco, CA, 1987.

[Ong84] J. Ong, D. Fogg, and M. Stonebraker, "Implementation of Data Abstraction in the Relational Database System INGRES," *SIGMOD Record* 14(1), March 1984.

[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 TODS* 12(4), Dec. 1987.

[Ozso83] Z. Ozsoyoglu and M. Ozsoyoglu, "An Extension of Relational Algebra for Summary Tables," *Proc. 2nd Intl. Workshop on Statistical Database Mgmt.*, Lawrence Berkeley Labs., Univ. of California, Berkeley, 1983.

[Pare88] J. Paredaens and D. Van Gucht, "Possibilities and Limitations of Using Flat Operators in Nested Algebra Expressions," *Proc. ACM PODS Conf.*, 1988.

[Pare85] C. Parent and S. Spaccapietra, "An Algebra for a General Entity-Relationship Model," *IEEE Trans. Software Eng.* 11(7), July 1985.

[Pist86] P. Pistor and F. Andersen, "Designing a Generalized NF2 Model with an SQL-Type Language Interface," *Proc. VLDB Conf.*, Kyoto, Japan, Aug. 1986.

[Rich92] J. Richardson, "Supporting Lists in a Data Model (A Timely Approach)", *Proc. VLDB Conf.*, Vancouver, Canada, 1992.

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

[Rich91] J. Richardson and P. Schwarz, "MDM: An object-oriented data model", in *Bulk Types & Persistent Data: The Third International Workshop on Database Programming Languages*, ed. P. Kanellakis and J. Schmidt, Naffplion, Greece, Morgan Kaufmann, August 1991.

[Ritt87] G. X. Ritter and J. N. Wilson, "Image Algebra: A Unified Approach to Image Processing", *Proc. SPIE Medical Imaging Conf.*, Newport Beach, CA, February 1987.

[Ross92] P. Ross, "Bulk Data Types: A Theoretical Approach", MSc Thesis, Computer Science Department, Hebrew University, Jerusalem, Israel, September 1992.

[Roth87] M. Roth, H. Korth, and D. Batory, "SQL/NF: A Query Language for ¬1NF Relational Databases," *Inform. Systems* 12(1), 1987.

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

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

[Scha86] C. Schaffert, T. Cooper, B. Bullis, M. Kilian, and C. Wilpot, "An Introduction to Trellis/Owl," *Proc. 1st OOPSLA Conf.*, Portland, OR, Sept. 1986.

[Sche85] H.-J. Schek, "Towards a Basic Relational NF^2 Algebra Processor," *Proc. Int. Conf. on FODO*, Kyoto, Japan, 1985.

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

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

[Scho87a] M. H. Scholl, H.-B. Paul, and H.-J. Schek, "Supporting Flat Relations by a Nested Relational Kernel," *Proc. VLDB Conf*, Brighton, England, Sept. 1987.

[Scho87b] M. H. Scholl and H.-J. Schek, eds., *Theory and Applications of Nested Relations and Complex Objects: An International Workshop*, Darmstadt, Germany, April 1987.

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

[Seth89] R. Sethi, *Programming Languages Concepts and Constructs*, Addison-Wesley, Reading, MA, 1989.

[Shaw90] G. Shaw and S. Zdonik, "A query algebra for object-oriented databases", *Proc. IEEE Intl. Conf. on Data Engineering*, Los Angeles, CA, Feb. 1990.

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

[Shen87] S. Shenoy and Z. Ozsoyoglu, "A System for Semantic Query Optimization", *Proc. ACM SIGMOD Conf.*, San Francisco, California, May 1987.

[Ship81] D. Shipman, "The Functional Data Model and the Data Language DAPLEX", *ACM TODS* 6(1), March 1981.

[Ston76] M. Stonebraker, E. Wong, P. Kreps, and G. Held, "The Design and Implementation of INGRES", *ACM TODS* 1(3), Sept. 1976.

[Ston87a] M. Stonebraker, J. Anton, and E. Hanson, "Extending a Database System with Procedures," *ACM TODS* 12(3), Sept. 1987.

[Ston87b] M. Stonebraker, J. Anton, and M. Hirohama, "Extendability in POSTGRES," *IEEE Database Eng. Bulletin* 10, June 1987.

[Ston86] M. Stonebraker, "Inclusion of New Types in Relational Database Systems," *Proc. IEEE Intl. Conf. on Data Engineering*, Los Angeles, CA, Feb. 1986.

[Stra90] D. Straube and M. Ozsu, "Queries and query processsing in object-oriented database systems", *ACM Trans. Office Info. Sys.*, 8(4), Oct 1990.

[Stro86] B. Stroustrup, *The C++ Programming Language*, Addison-Wesley, Reading, MA, 1986.

[Subr93] B. Subramanian, S. Zdonik, T. Leung, and S. Vandenberg, "Ordered Types in the AQUA Data Model", *Fourth International Workshop on Database Programming Languages*, New York, New York, 1993, to appear.

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

[That68] J. Thatcher and J. Wright, "Generalized Finite Automata Theory with an Application to a Decision Problem of Second-Order Logic", *Math. Sys. Theory* 2(1), 1968.

[Tsur86] S. Tsur and C. Zaniolo, "LDL: A Logic-Based Data Language", *Proc. VLDB Conf.*, Kyoto, Japan, August 1986.

[Ullm87] J. Ullman, "Database Theory — Past and Future," *Proc. ACM PODS Conf.*, San Diego, CA, March 1987.

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

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

[Vanc92] B. Vance, "Towards an object-oriented query algebra", Tech. Report CS/E91-008, Dept. of Computer Science and Eng., Oregon Graduate Institute, Beaverton, OR, January 1992.

[Vand88a] S. Vandenberg, "The EXTRA/EXCESS Data Dictionary," EXODUS working document, Univ. of Wisconsin-Madison, 1988.

[Vand88b] S. Vandenberg, "EXTRA/EXCESS User Manual," EXODUS working document, Univ. of Wisconsin-Madison, 1988.

[Vand89] S. Vandenberg, "Practical Complex Object Algebras," in [Grae89].

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

[Vand91] S. Vandenberg and D. DeWitt, "Algebraic Support for Complex Objects with Arrays, Identity, and Inheritance", *Proc. ACM SIGMOD Conf.*, Denver, CO, May 1991.

[VanG87] D. Van Gucht, "On the Expressive Power of the Extended Relational Algebra for the Unnormalized Relational Model," *Proc. ACM PODS Conf.*, 1987.

[VanG86] D. Van Gucht and P. Fischer, "Some Classes of Multilevel Relational Structures," *Proc. ACM PODS Conf.*, 1986.

[Webe78] H. Weber, "A Software Engineering View of Database Systems," *Proc. VLDB Conf.*, 1978.

[Yu91] L. Yu and S. Osborn, "An Evaluation Framework for Algebraic Object-Oriented Query Models", *Proc. IEEE Intl. Conf. on Data Engineering*, Kobe, Japan, April 1991.

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