

ALGEBRAIC LANGUAGES FOR XML DATABASES

by

MARIA G. CHINWALA

mariagc@arches.uga.edu
Tel: 706-380-6959

JOHN A. MILLER
(contact author)

Department of Computer Science
The University of Georgia
415 Boyd Graduate Studies Research Center
Athens, GA 30602-7404

jam@cs.uga.edu
Tel: (706) 542-3440 Fax: (706) 542-2966

ABSTRACT

XML is becoming increasingly popular as a means of exchanging a wide variety of data on the web. It is anticipated that in the future, many websites will be built from XML documents. XML databases would be required to manage these websites and also provide a way for users to search their contents. In this paper we give an overview of the development of query algebras from the original relational algebra to the algebras for extended-relational, object-oriented and semi-structured or XML data models. We discuss in detail and provide a cross comparison of the various algebras proposed for XML. We also discuss the implementation of a particular algebra in the context of MMXDB, which is a main-memory XML database system that we are developing.

INDEX WORDS: XML, Query, Algebra, Semi-structured, Database.

1. Introduction

XML is becoming increasingly popular as a means of exchanging a wide variety of data on the web. It is anticipated that in the future, many websites will be built from XML documents. XML databases would be required to manage these websites and also provide a way for users to search their contents. A standard XML Query Language and Algebra are required, which should be based on a standard Data Model. The XML Query Working Group of the World Wide Web Consortium (W3C), established in September 1999 is working towards developing these standards for XML databases. This paper discusses the work done by major research groups towards developing XML algebra. A good algebra should help in generating query plans that lead to efficient query execution. Since XML data is semi-structured, the algebras developed for relational or object-oriented data cannot directly be used for XML queries. However, research in these areas has influenced the design of XML algebras and thus we have provided an overview of the development of relational and object algebras as well. The review of XML algebras was carried out with the intention of choosing one of them to express queries which would be input to an XML database system, MMXDB, that we are developing. MMXDB is a main memory database system designed for storage and querying of XML documents. Using a particular XML algebra in the development of MMXDB gave us the opportunity to study the implementation issues involved in generating query plans.

The organization of the rest of the paper is as follows. In section 2, we give an overview of the various query algebras developed for extended-relational, object-oriented and semi-structured or XML data models. In section 3, we analyze the various XML algebras and discuss the choice of a particular algebra for implementation. In section 4, we present the implementation of this algebra and our findings. We conclude in section 5 and discuss directions for future work.

2. Review of Query Algebras

Most database query algebras are based on the algebra for the relational model proposed by Codd in 1970 [6] and later formalized in [36]. 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). These fields were restricted to contain only atomic

(scalar, non-composite) values. This is known as the first normal form (or 1NF) restriction. A relation (or table) is a set of tuples with identical layout. This layout is called the relation's schema. Relations are manipulated using relational algebra, which has the expressive power of relational calculus. The five standard relational algebra operators are Union (\cup), Difference ($-$), Cartesian Product (\times), Projection (π) and Selection (σ).

2.1 Extended Relational Algebras

After Codd proposed the relational algebra in 1970, several researches developed algebras, which operated on non-first normal form (or NF^2) relations. An NF^2 or nested relation is one in which attribute values are not restricted to scalars – they can be relations (sets of tuples) as well. The use of nested relations was first advocated by [35] in the late 70's. However, the nested relation model as discussed in this paper and its associated algebras were mostly developed in the mid to late 80's. Some of these NF^2 algebras are DASDB [25], AIM [8], Vanderbilt [11], Verso [1], SQL/NF[23], NRDM [9] and Powerset [13]. All of these algebras except [8] are value-based (as opposed to identity-based), like the relational algebra. This means that to get a handle on a tuple one must use its value(s), not some sort of name or identifier or surrogate for the tuple. Most of these algebras, except [23] have no formal calculus defined. A study of these algebras [28] showed that many of them had common features or operators which were more or less based on Codd's original relational algebra operators. Table 1 lists the extended relational algebras and their operators.

The union, difference, selection, projection and cartesian product operators are defined as in relational algebra except selection and projection, which have been extended in different ways. The selection operator has been extended to allow set comparators and also to allow algebraic expressions to appear as a selection condition. The algebras of [25, 8, 1] have extended projection by allowing arbitrary algebraic expressions to appear in projection lists in order to facilitate retrieval of nested data from a relation. The extended union operator (\cup^e) is defined recursively for pairs of relations with the same schema. For flat schema, it works exactly like the relational \cup . However,

Table 1. Relational and extended-relational algebra operators		
Algebra	Operators	Year
Relational	$\sigma \pi \cup - \times$	1970
Vanderbilt	$\sigma \pi \cup - \times$ NEST UNNEST UNNEST*	1983
AIM	$\sigma \pi \cup - \times \nu \mu \chi \rho$	1986
DASDB	$\sigma \pi \cup - \times \nu \mu$	1986
Verso	$\pi \cup \cap - \times$ extension selection restrict rename join restruct	1986
SQL/NF	$\sigma \pi^e \cup^e \cap^e -^e \times \times ^e \nu \mu$	1988
Powerset	$\sigma \pi \cup - \times \nu \mu \Pi$	1988
NRDM	$\sigma^e \pi^e \cup^e -^e \times \nu \mu$	1988
Complex Object	$\cup \cap - \times$ replace(ρ) powerset set-collapse	1988

when a relation-valued attribute is encountered, the \cup^e operator is applied to this relation, recursively. That is, 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 . Figure 2 illustrates union and extended union of relations in Figure 1. All the algebras listed in the table are NF^2 algebras except the Complex Object Algebra [2] which is defined on their complex object data model. We will shortly describe how this model differs from the extended-relational model. The operator $-^e$ is defined recursively in a fashion similar to the \cup^e operator. The π^e operator consists of a normal relational operator 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. The renaming operator (ρ) is used to rename attributes of a nested relation. The Powerset algebra has powerset operator (Π) which takes a relation and produces all subsets of the relation. Complex Object Algebra too has the powerset operator and also the set-collapse operator which when given a set of sets, unions all of them to produce a single set. The operators, which appear in almost all NF^2 algebras, are the nest (ν) and unnest (μ) operators. The nest 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. Unnest is the inverse of nest. An example

dept	projects		
	code	staff	
		name	age
Sales	A	Mike	30
	B	Pete	25
IT	A	Bob	40
		Ann	36

dept	projects		
	code	staff	
		name	age
Sales	A	Bob	40
		Tom	35
IT	A	Bob	40
		Ann	36
		Luke	22
	C	Jim	21
		Ed	50

Figure 1. Nested relations r1 and r2

dept	projects		
	code	staff	
		name	age
Sales	A	Mike	30
	B	Pete	25
IT	A	Bob	40
		Tom	35
Sales	A	Bob	40
		Ann	36
IT	A	Bob	40
		Ann	36
		Luke	22
	C	Jim	21
		Ed	50

dept	projects		
	code	staff	
		name	age
Sales	A	Mike	30
		Bob	40
		Tom	35
Sales	B	Pete	25
IT	A	Bob	40
		Ann	36
		Luke	22
	C	Jim	21
		Ed	50

Figure 2. Union and extended union of relations r1 and r2

should help illustrate the use of the nest and unnest operators. Figure 3 shows the result of applying the unnest operator twice to relation r1 of Figure 1.

$$\mu_{\text{projects}} = \{\text{code}, \text{name}, \text{age}\} (\mu_{\text{staff}} = \{\text{name}, \text{age}\} (r1))$$

dept	code	name	age
Sales	A	Mike	30
Sales	B	Pete	25
IT	A	Bob	40
IT	A	Ann	36

Figure 3. Result of applying unnest operator on relation r1

We can get the relation r1 of Figure 1 by applying the nest operator twice to the relation of Figure 3.

$$\nu_{\text{projects}} = \{\text{code}, \text{staff}\} (\nu_{\text{staff}} = \{\text{code}, \text{name}, \text{age}\} (r1))$$

The Vanderbilt algebra has the UNNEST* operator which simulates a sequence of unnest operations on a relation to transform it into a flat (1NF) relation. The most important property of the nest and unnest operators is that while unnest is always the inverse of nest, it is not always the case that $\text{nest}(\text{unnest}(r)) = r$. This issue and associated operators are discussed in appendix A. Verso has the extension and restruct operators which operate on the schema to extend and restructure it respectively.

In the early 90's researchers worked on developing algebras for complex object data models, which extend the NF² 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 in sets), or any combination of the set and tuple type constructors (so called because they are used to construct types). One such algebra is the Complex Object Algebra [2]. Here the replace (ρ) operator is a restructuring operator that takes a function parameter describing the restructuring.

2.2 Object Oriented Algebras

Algebras were also developed for object-oriented data models [26, 27], which are extensions of the complex object models with the notions of object identity, inheritance and methods. Unlike the NF^2 algebras, no common set of operators emerged, although we can say that the general approach to the manipulation of complex objects evolved from NF^2 algebras as well as Functional Programming Languages [15]. Functional programming concepts were needed in order to perform restructuring operations on complex objects. The relational model did not require sophisticated restructuring operations since, in the relational model, restructuring is limited to adding or deleting fields of tuples. In complex object models, restructuring can be defined by a function to be applied to each member of a set. For example, the “image” and “project” operators of the ENCORE/EQUAL algebra [26] and the “replace” operator of [2] are higher order operators with function parameters that describe the restructuring to be applied to a set of objects. Higher order operators are operators which can have other operators or functions as parameters. The need for these operators emphasized the view of an algebra as a functional language. Use of higher order operators gives the advantage of a functional programming language by allowing construction of complex queries incrementally from simpler ones. The most recent object algebras are AQUA [17] and QAL [24]. AQUA was proposed in 1993 and was a result of joint effort among researchers, who had participated in the design of previous algebras including [26, 27], whereas, QAL was proposed in 1998. Table 2 shows the operators for three object algebras. The list of operators is not complete and this listing illustrates lack of a well-defined set of common operators other than operators derived from relational algebra.

The select operator of ENCORE/EQUAL algebra is similar to the relational selection. As mentioned earlier the image and project operators retrieve, for each object in a set, the result of one (image) or more (project) functions defined on the objects of the set. The union, intersection and difference operators are all based on object identity. The flatten operator is like set-collapse of Complex Object algebra. The NF^2 operators, nest and unnest are also provided. An explicit duplicate elimination operator is available and there is an operator called coalesces to eliminate duplicates at a certain level of nesting

Algebra	Operators	Year
ENCORE/EQUAL	select image project union intersection difference flatten duplicate_elimination coalesce nest unnest	1990
AQUA	apply select exists forall mem set choose group dup_elim nest unnest convert join tup_join outer_join union intersect diff multiset	1993
QAL	select union differ intsc unnest group apply tuple close apply_at	1998

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 reference to it is replaced with a reference to the other equal object. This operator helps avoid the creation of multiple copies of the same nested set.

As compared to ENCORE/EQUAL, AQUA has a rich set of operators. The authors explicitly state that AQUA is not a minimal set of operators. AQUA was designed to be very general in order to more easily accommodate many query languages. AQUA was also designed to support many different collection types in a uniform manner although currently only defined to operate on sets and multisets. For instance, the join operator has been generalized to take an extra function argument in addition to the two input sets or multisets and a matching predicate. This function argument is a “combining” function which allows us to specify how to combine pairs of objects from the two input sets that satisfy the predicate. If a combining function is not required, the standard tup_join operator can be used. AQUA also has a unique approach to enforcing a notion of equality among elements of a set. Equality is essential to the definition of some operators like set union. Object identity is the default equality for set elements. However if the contents of a set are needed in a fashion that is sensitive to the notion of equality then the dup_elim operator can be used. This operator too has been generalized to take an equality (in the form of a binary predicate) as a parameter and eliminate duplicates

under that equality. The group operator takes a function parameter and uses it to group elements of a set into equivalence classes. Choose takes a set and nondeterministically selects one element of the set as its result. The set and multiset operators are set definition operators and convert is used to convert from one to the other. The apply operator as described earlier is a mapping operator that applies a given function to the set of input objects.

Operators of the QAL algebra too evolved from relational algebras and the FQL family of functional languages [4]. The “select”, ”union”, ”differ”, ”intersect”, ”nest”, ”unnest”, ”apply” and ”group” operators are defined as in earlier object algebras. The main new operator proposed is the `apply_at` operator which extends the apply operator by taking a path expression (e.g. `dept.projects.code`) as an input in addition to the function parameter. This is a very useful operator since it makes the algebra expression simpler by eliminating use of restructuring operators such as `nest`, `unnest`, `apply` etc. to query nested components of complex objects. The `close` operator takes a function parameter and computes the closure of the input set using this function. It is provided as a simple tool for the manipulation of recursive data structures. The `tuple` operator is similar to the `select` operator of other object algebras. QAL also has a set of operators for querying database schema which are not discussed here. Since QAL is the most recent object oriented algebra, it has been heavily influenced by earlier work and at this time seems the most simple (least number of operators) and complete set. Some other early object-oriented algebras developed are [16, 21].

2.3 Semi-Structured Algebras

Semi-structured data is data whose structure is irregular or loosely defined. Semi-structured documents may be used to represent any kind of known data sources; relational data, object-oriented data etc. XML documents can be viewed as semi-structured data sources in the Web context. In this section we will focus on work done on developing an algebra for XML queries. Most of the XML algebras that we will discuss have operators similar to the extended-relational and object algebra operators. These operators have been modified so that they can be applied on the XML data model which we will describe shortly.

The World Wide Web Consortium or the W3C established the XML Query Working group in September 1999 in order to establish standards for XML databases. These include standards for an XML data model, query language and algebra. For a detailed discussion on different XML query languages and algebras see [5, 32]. Several proposals for XML Algebras such as [3, 10, 14, 7] were submitted to the W3C, of which the AT&T algebra [10] was selected and the working draft [29] was published in December 2000. In this section we will discuss various proposals submitted to the W3C as well as recent work done on XML algebras after the working draft was published.

The XML Query Working group published the XML Query Data Model draft [30] in May 2000. The XML Query Data Model formally defines the information contained in the input to an XML Query Processor. In other words, an XML Query Processor evaluates a query on an instance of the XML Query Data Model. Consider the XML document and its associated DTD of Figure 4. Figure 5 is a simplified version of the tree based data model.

<pre> <dbgroup> <member ssn="123456789"> <name>Smith</name> <age>28</age> <office><building>A</building><room>6</room></office> </member> : <member ssn="101112131"> <name>Clark</name> <age>35</age> <office><building>A</building><room>7</room></office> </member> </dbgroup> </pre>	<pre> <!DOCTYPE dgroup [<!ELEMENT dbgroup (member+)> <!ELEMENT member (name, age, office)> <!ATTLIST member ssn CDATA #REQUIRED> <!ELEMENT name (#PCDATA)> <!ELEMENT age (#PCDATA)> <!ELEMENT office (building, room)> <!ELEMENT building (#PCDATA)> <!ELEMENT room (#PCDATA)>]> </pre>
---	---

Figure 4. XML document dbgroup.xml and its associated DTD

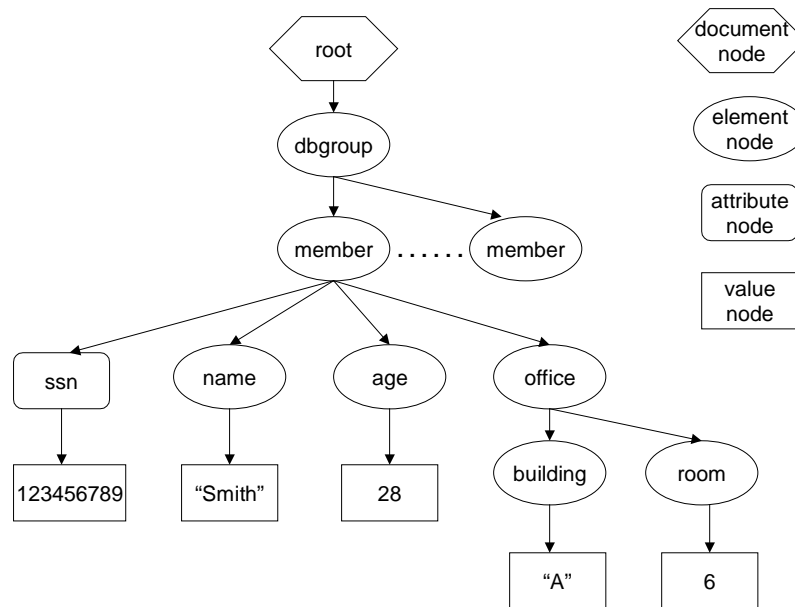


Figure 5. XML data model

We will first discuss the IBM algebra [3] which was proposed in September 1999 by researchers from IBM, Microsoft and Oracle. This algebra has some operators which are similar to the object algebra operators. Then we will discuss Niagara [12], which is the most recent algebra proposed in 2001 by researchers from The University of Wisconsin. This algebra was developed after the authors unsuccessfully tried to implement some of the existing algebras including the IBM algebra. Next, we will discuss some system specific algebras such as YATL [7] and Lore [14]. YATL algebra was proposed in May 2000 and was developed for the YAT XML-based integration system. Lore was also specifically developed for the Lore database system and was originally proposed in 1997 by researchers from Stanford University. Finally we describe the AT&T algebra [10] which is an updated version of their algebra originally proposed in June 2000 by researchers from AT&T and Bell Labs.

2.3.1 IBM Algebra

The IBM Algebra [3] provides algebra operators that operate on a graph-based data model, proposed by them. This is purely a logical model and nothing is specified about the underlying storage representation or physical operators. In addition to standard retrieval and querying operators, IBM algebra also has what are called reshaping operators which help create new XML documents from fragments of selected documents. The path navigation operator (ϕ) is expressed as

$$\phi [edgetype, name](vertex-expression)$$

The *edgetype* can be E (element), A (attribute) or R (reference). The *vertex-expression* describes the collection of vertices in the data model graph from which edges with the given *edgetype* and *name* should originate. The (ϕ) operator returns this set of edges. For simplicity, $\phi[E,b](a)$ is written as a/b . Consider query Q1 based on the XML document of Figure 4.

Q1: Find all members of the database group whose age is less than 30 years.

This is expressed as shown below:

$$Q1: \quad \sigma[value(x/member/age/~data)<30](x:child(/dbgroup))$$

Here (σ) is the selection operator and *child* is the property of an edge that returns the vertex referred to by the edge. The selection operator returns a collection of vertices. All the operators operate on collections of edges or collection of vertices. The join (\otimes) operator takes two collections of vertices as arguments along with a condition. For each pair for which the condition evaluates to true a virtual reference edge is created between the two vertices. The expose (ϵ) and return (ρ) operations are provided for queries which may want to expose components or fragments of documents. Numerous other operators are also provided such as sort (Σ) which allows reordering of a set of edges and map (μ) which applies a specific function to a collection of edges or vertices. Although not directly mentioned by the authors, this algebra seems to have been influenced by the concept of an algebra being a functional language since here too all the operators are defined as functions, which operate on collections of edges or vertices. The main

drawback of this algebra, we felt, was the lack of optimization rules to reduce the cost of query execution.

2.3.2 Niagara Algebra

This algebra [12] is similar to the IBM algebra since its operators are designed to operate on a graph-based model. However the developers of Niagara discuss some of the drawbacks of the IBM algebra which they discovered while trying to implement it. These drawbacks prompted the design of Niagara algebra. The first drawback of the IBM algebra is that the algebraic framework assumes, that at the time of writing the query, the type of each vertex (attribute, element or reference) is known. Secondly, the algebra defines a large collection of operators and explicitly preserves all variable bindings appearing in a query, by creating multiple clauses per query and assigning intermediate results to these bindings. This leads to complex structures when representing a query, which makes the operators hard to deal with in an implementation. And finally, as we have also pointed out earlier, the lack of optimization rules. One of the new ideas proposed in the Niagara Algebra is that instead of having the operators operate on a set of vertices, they operate on a set of bags of vertices. Each bag contains the vertex being operated upon as well as all the vertices already visited along the path to that vertex. To understand the need for this approach, consider a document, which consists of book elements each of which has author sub-elements. If an operator filtered out books based upon specific author and then another operator considered the authors of books, the fact that a specific author was “filtered out” was lost since all authors “came along” with the book element. This would not have been a problem if instead of starting out with a set of set of book elements; we started out with a set of bags where each bag consisted of a book element and only one of its author sub elements. Figure 6 shows the difference between these two sets. To differentiate between different elements of a bag, each bag element is annotated with the path expression that corresponds to its contents. To trigger an evaluation of a path expression on a specific element or bag, the algebra uses the bag element annotations as entry points. For example, to use the book element of a bag to get the lastname of authors the entry point notation would be *author.lastname* and to use author element to get the lastname the entry point notation would be

lastname : *book.author*. Use of entry point notation makes the algebra expression less complex. The query Q1 is expressed in this algebra as:

$$Q1: \quad \Sigma (*.\text{member}) [\sigma *.\text{member.age} < 30 [\phi(*.\text{member}) [s (\text{dbgroup.xml})]]]]$$

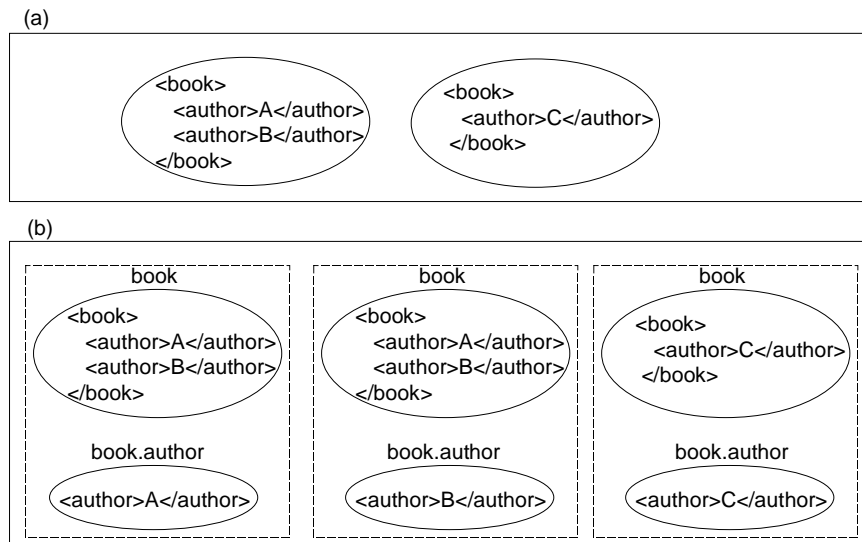


Figure 6. (a) Set of vertices (b) Set of bags of vertices

(*s*) is the source operator used to specify the source of the XML document; (ϕ) is the follow operator to follow a specified path inside the elements of a bag; (σ) is the select operator and (ϵ) is the expose operator to expose specified path expressions of incoming bags. The other operators of this algebra are the relational operators union (\cup), intersection (\cap), difference ($-$), join (\bowtie) and cartesian product (\times); rename (ρ) to rename an entry point; vertex (ν) to create a new vertex and group (γ) to group the bags on a specified list of elements. The authors have specified explicit optimization or rewrite rules for the algebra.

2.3.3 YATL Algebra

This algebra [7] was developed for YAT, an XML-based integration system which integrates data from different sources such as object databases or semi structured

web repositories. The authors have proposed only two new operators, namely the *Bind* and the *Tree* operators. All others are standard operators drawn from earlier work on Relational Algebra and Object Algebra. Starting from an arbitrary XML structure, the *Bind* operation is applied, whose purpose is to extract the relevant information and produce a structure called *Tab*, which is comparable to a \neg 1NF relation. Then, on the *Tab* structures, standard operators such as *Join*, *Select*, *Project* can be applied. Finally, the inverse operation to *Bind*, called *Tree* generates a new XML Structure. Thus our query Q1 can be represented as shown in Figure 7. This algebra is independent of any underlying storage structure. Since the main focus of this paper is on efficient query evaluation for an integrated system the authors have discussed optimization/rewrite rules only for the *Bind* and *Tree* operators. *Bind* is quite a powerful operation providing support for type filtering and horizontal and vertical navigation. User queries are optimized by generating integration views locally or by pushing queries to the external source. The *Tree* operation is a sequence of *Group*, *Sort* and *Map* operations which are Object algebra operators. This XML algebra is better than the IBM Algebra in the sense that it is more optimizable. The drawback is that the authors have developed it in the context of an integration system using their own data model and type system and not provided a well-defined list of operators and explicit optimization rules.

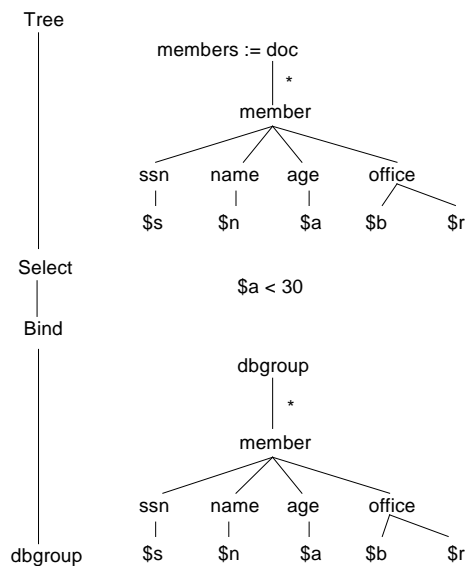


Figure 7. Query Q1 as evaluated by YATL algebra

2.3.4 Lore Algebra

The Lore system [14] at Stanford is a complete DBMS designed specifically for semi structured data. The main focus of Lore's query processor is on cost-based query optimization, specially the efficient evaluation of path expressions. Each query is transformed into a logical query plan using logical operators such as *Select*, *Project*, *Discover*, *Name*, etc. which can be considered algebra operators. Each logical query plan can give rise to more than one physical query plan and a cost based approach is used to select the best physical plan. The logical query plan and one of the several possible physical query plans for query Q1 are shown in Figure 8.

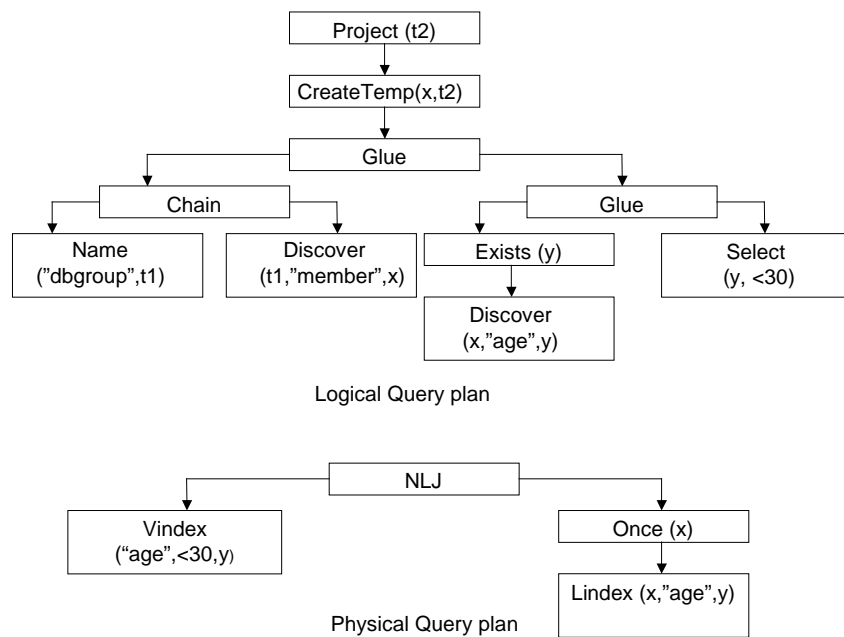


Figure 8. Lore query plans for query Q1

Each simple path expression in the query is represented as a *Discover* node, which indicates that in some fashion information is discovered from the database. Simple path expressions are grouped together into a single path expression as a tree of *Discover* nodes connected via *Chain* nodes. It is the responsibility of the *Chain* operator to optimize the

entire path expression represented in its left and right subplans. Places where independent components meet are called *rotation points* since during the creation of physical query plans the order between independent components can be rotated to get vastly different physical query plans. Each rotation point is represented by a *Glue* node. The *CreateTemp* and *Project* nodes at the top of the plan are responsible for gathering the satisfying evaluations and returning the appropriate objects to the user. One of the physical operators is $Vindex(l, Op, Value, x)$ which accepts a label l , an operator Op and a *Value* and places into x all atomic objects that satisfy the “*Op Value*” condition and have an incoming edge labeled l . The $Lindex(x, l, y)$ operator places into x all objects that are parents of y via an edge labeled l . The $Once(x)$ operator only allows an evaluation to be passed to its parent node if the object bound in x has never been seen before. NLJ is the nested-loop join operator which denotes a dependent join where the left subplan passes bound variables to the right subplan. Thus, we see that the main physical operators are index operators, which are used to represent very specific types of indexes maintained in the Lore system. The main drawback of this algebra is that since the logical operators have been developed keeping in mind the physical index operators, they become very specific to the Lore system.

2.3.5 AT&T Algebra

This algebra [10] has been inspired by SQL, OQL and Nested Relational Algebra. The authors have also implemented a type checker and an interpreter for the algebra in OCaml, which is a functional programming language. No storage structures have been discussed. In this algebra most of the operations are based on the iteration operation. The iteration operation is a *for* expression that iterates over elements in a document given the path expression. The selection operation is a *for* expression with a *where* clause and the join operation is nested *for* expressions. The operator for path expressions(/) or the Project operation is also built from *for* expressions, the *children* function and the *match* expression. For example, the expression:

dbgroup/member

is equivalent to the expression

```

for c in children(dbgroup) do
  match c
    case a : member[UrType] do a
    else()

```

where UrType is the type of the element. Using this algebra, query Q1 is expressed as follows:

```

Q1:   for m in dbgroup/member do
        where m/age/data() < 30 do
        m

```

Several optimization rules have been specified for this algebra, which simplify the queries by eliminating unnecessary *for* expressions or by reordering or distributing the computations.

3 Analysis

3.1 Cross Comparison

Most XML algebras have operators which operate on a model similar to the XML query data model [30] as shown in Figure 5. However, as we have seen in the previous section, the algebras are quite different from each other. While the IBM [3], AT&T [10] and Niagara [12] algebras were proposed as stand alone XML query algebras, Lore [14] and YATL [7] algebras were developed for the Lore database system and YAT integration system, respectively. Thus, the algebra operators proposed by Lore could be used to generate query plans that could be efficiently executed using the physical index operators of the Lore database system. Similarly, since YATL was developed for an XML based integration system, the optimization strategies are focused towards efficiently querying distributed data. The IBM algebra, though not system specific like Lore, suffers from some drawbacks such as complex query structures with a large number of variable bindings and a lack of optimization rules. The Niagara algebra, which is the most recently proposed algebra (it was proposed after the W3C had selected the AT&T algebra as the proposed standard), has operators similar to the IBM algebra which in turn has operators similar to object algebra operators. Niagara was developed to overcome the drawbacks of IBM algebra. The authors have proposed a new technique of operating on sets of annotated bags which reduces the complexity of the query. Query rewrite rules

for optimization are also discussed. Finally, we have the AT&T algebra [10] which was selected by the W3C as the proposed standard XML algebra [29]. Although the W3C chose to propose this algebra as the standard, yet the fact remains that unlike other algebras, the expressions here resemble a high-level query language. In fact, in a July 2001 revision [31] to the original algebra document by the W3C, there have been some syntactical changes and now the document is titled as the proposed semantics of an XML query language XQuery.

3.2 Experimentation with a chosen algebra

Since none of the XML algebras proposed had discussed implementation issues in detail, we decided to build an experimental prototype XML database system which would process queries written in an algebraic form. We were interested in implementation issues such as generation, optimization and evaluation of query plans. Implementing a database system also involves storage, indexing, transaction support, etc. which are not the focus of this paper. The indexing, storage and query evaluation techniques for our prototype system are described in [18]. We did not choose Lore or YATL algebras since these are already part of existing systems. Niagara algebra had not been proposed when we started our work. Left with the IBM and AT&T algebras, we opted to implement the latter since the W3C had chosen this algebra as the proposed standard for XML queries.

4. Experimental Prototype

4.1 Architecture of MMXDB

The overall architecture of our prototype XML database system is shown in Figure 9. It is a simple main memory database system which is in the initial stages of development and does not yet have support for transactions, concurrency control, security and recovery. We have a thin client which passes user queries in an algebraic form to the database system. The parser parses this query to produce a syntax tree. This syntax tree is passed to the query plan generator which modifies the syntax tree to form the query evaluation tree. This query evaluation tree is input to the query evaluator which interacts with the storage manager and evaluates the query.

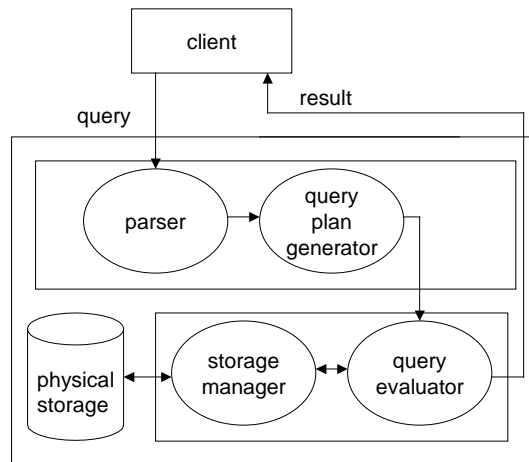


Figure 9. Architecture of MMXDB

4.2 Query Processing

4.2.1 AT&T algebra syntax and modifications

Figure 10 is the syntax of the core algebra that we implemented. The algebra also has a specific syntax to define the data and schema of algebra documents. Consider the XML document of Figure 4. The data definition statements for this document are as shown below:

```
type Dbgroup =
  dbgroup [ Member{1, *} ]
```

```
type Member =
  member [
    @ssn [ Integer ],
    name [ String ],
    age [ Integer ],
    office [ Office ]
  ]
```

```
type Office =
  office [
    building [ String ],
    room [ Integer ]
  ]
```

name	Name	
variable	Var	
constant	Const	
binary operator	BinaryOp ::= + - and or = != < <= >= >	
unary operator	UnaryOp ::= + - not	
expression	Exp ::= Const	atomic constant
	Name	element name
	Var	variable
	@Name	attribute
	Exp, Exp	sequence
	Exp \wedge Exp	intersection
	()	empty sequence
	if Exp then Exp else Exp	conditional
	let Var = Exp do Exp	local binding
	FuncName (Exp)	function application
	Exp BinaryOp Exp	binary operator
	UnaryOp Exp	unary operator
	for Var in Exp do Exp	iteration
	sort Var in Exp by Exp	sort
	(Exp)	bracket
	FuncName ::= sum avg min max Var	
query item	Query ::= query Exp	query expression

Figure 10. Core syntax of AT&T algebra

The data population statements for this document are as shown:

```

let dbgrp0 : Dbgroup =
  dbgroup [
    member [
      @ssn [ 12345678 ],
      name [ "Smith" ],
      age [ 28 ],
      office [
        building [ "A" ],
        room [ 6 ]
      ]
    ],
    member [
      @ssn [ 101112131 ],
      name [ "Clark" ],
      age [ 35 ],
      office [
        building [ "A" ],
        room [ 7 ]
      ]
    ]
  ]
]

```

The path navigation operator (/) is not a part of the core algebra syntax. If we wanted to find all the members in dbgrp0 the query syntax would be

```
Q1:      query dbgrp0/member
```

This would be rewritten in core algebra syntax as

```
Q1:      query for e in dbgrp0 do
           for x in nodes(e) do
             match x
               case m : member[AnyComplexType] do m
             else ()
```

In the above expression, nodes(Exp) is a built in function of the algebra which returns children of the input node. The algebra has other similar built-in functions defined over the data model which are known as Data Model Constructor and Accessor functions. We have not implemented the nodes function nor the match-case expression. This is because we believe that we can rewrite the original query in a simpler form¹ as shown below:

```
Q1:      query for m in member(dbgrp0) do m
```

The above expression is also in the core algebra syntax except we now allow the element name (in this case – member) to be specified as a function name. The only information we lose is that book is of type AnyComplexType, which indicates that book is an element. If we had to get an attribute of book (e.g. ssn) then the type would have been specified as AnySimpleType. To avoid this loss of information, we always prefix the name with an underscore if it is an attribute name. So the query Q2 to find the ssn of all members of dbgrp0 would be:

```
Q2:      query for m in member(dbgrp0) do
           for s in _ssn(m) do s
which is equivalent to the path expression dbgrp0/member/ssn
```

4.2.2 Parsing

We used JavaCC [19] which is a parser generator tool in order to generate a parser for the AT&T algebra². The input to the tool is a grammar specification. As shown in Figure

¹ This was partially motivated by earlier work on Active KDL [20, 22]

² An alternative parser generator tool written in Java that works with left recursive grammars is CUP[33]

10, the AT&T algebra document had the grammar production rules specified and so generating the parser was not complicated. The only problem we encountered was that the JavaCC tool does not work with left recursive grammars. AT&T algebra is left recursive. A grammar is said to be left recursive if it has a nonterminal A such that $A ::= Ax$ for some string x . The AT&T algebra grammar has production rules such as $\text{Exp} ::= \text{Exp BinaryOp Exp}$ which make it left recursive. The standard approach to eliminating left recursion, which is to introduce a new nonterminal and production rule had to be used repeatedly.

4.2.3 Syntax Tree Generation

The first step towards generating a query tree was to generate a syntax tree for the query or algebra expression. For this we used the JJTree tool which comes with JavaCC³. JJTree constructs a syntax tree for each expression after it has been successfully parsed. The default structure of the tree is that when a production rule R generates a tree, that tree's children are the trees generated by the rules called on R 's right-hand side. Tree nodes are Java classes which have the same name as the rule's left-hand side. Figure 11 shows the syntax tree for query Q2 of section 4.2.1 All the nodes have the same name as the left-hand side of the production rule which generated the node.

4.2.4 Tree Transformation for Optimization and Evaluation

Just by looking at the syntax tree of Figure 11, it is not very clear which production rule generated the tree node. This is especially true for the production rules that have the term Exp on the left-hand side. So the first step towards generating a query evaluation tree was to annotate the tree nodes to make the syntax tree more meaningful. After annotating the tree nodes, the syntax tree of Figure 11 is now as shown in Figure 12.

³ An alternative syntax tree generator is JTB[34]

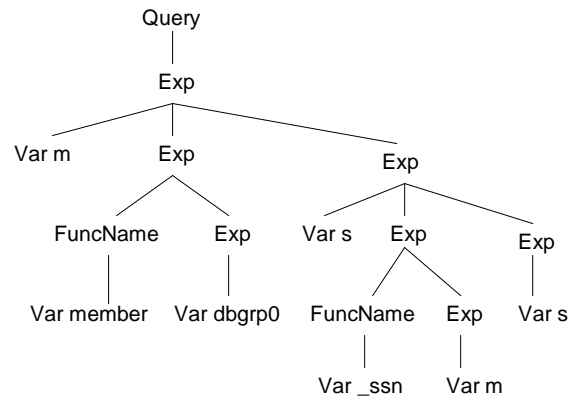


Figure 11. Syntax tree for query Q2

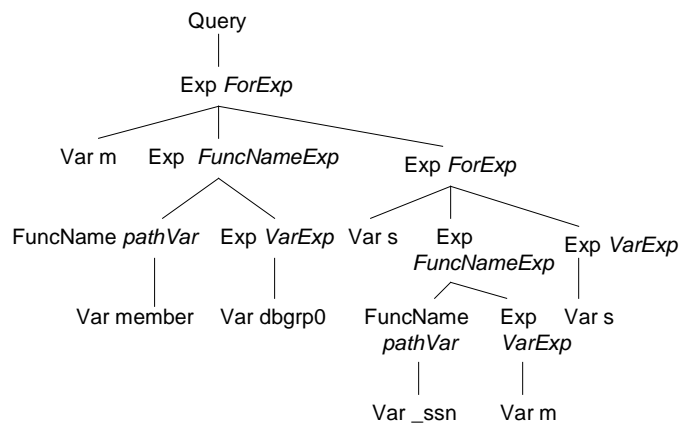


Figure 12. Annotated syntax tree for query Q2.

Other slight modifications had to be made to some syntax trees in order to reorder or remove nodes which were generated due to the nonterminals we added to the grammar to remove left recursion. For instance Figure 13 shows part of a syntax tree in which PrimaryExp node was added to remove left recursion in expressions of the form $Exp ::= Exp \text{ BinaryOp } Exp$. Figure 14 is the tree after removing the PrimaryExp node.

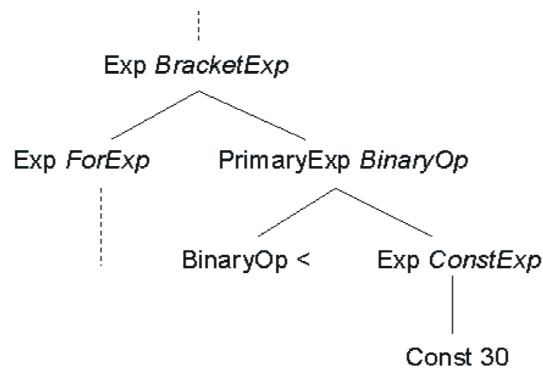


Figure 13. Syntax tree with PrimaryExp node.

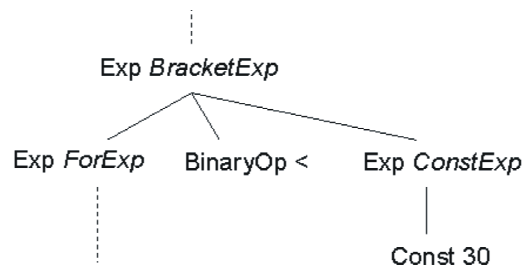


Figure14. Syntax tree with PrimaryExp node removed

On observing the syntax trees thus generated we noticed that all the leaf nodes were the data nodes; in other words, nodes that specified the data to be operated on. Thus data flow would always be bottom up. We decided that the annotated syntax tree, with some more slight modifications would suffice as a query tree. The modification for the syntax tree of Figure 12 is shown in Figure 15. Here we have removed the FuncName node and annotated the variable associated with it as a path variable. We defined an evaluate function for each type of internal node. The query would be evaluated by recursively

calling the evaluate function of each node starting from the root node. The details of this strategy and the various evaluate functions are described in detail in [18].

4.3 Observations and Findings

Once the query tree was generated and the evaluate functions of the various types of tree nodes were defined, evaluating any kind of tree was very simple since all it involved was a call to the evaluate function of the root node. This we felt was one of the advantages of

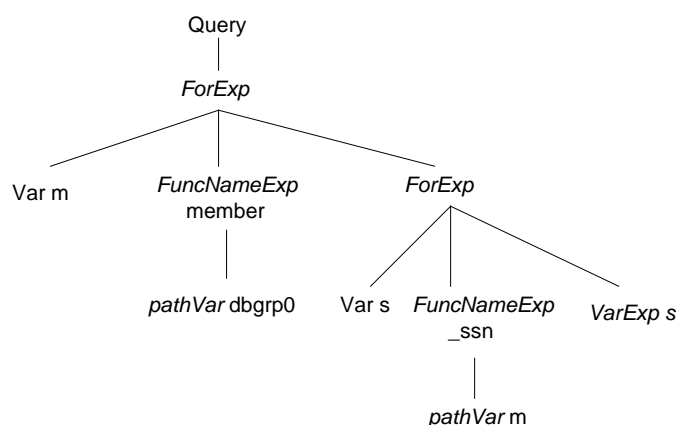


Figure 15. Query tree for query Q2

this algebra. However there were a few disadvantages. One of the disadvantages of the recursive evaluation technique was that the query plan was difficult to optimize. There was little room to reorder operators such as joins (which in this algebra would be nested for loops) for efficient query evaluation. Thus, creative storage and indexing techniques are needed in order to efficiently execute queries using this algebra. It should, however, be mentioned that the authors have specified rewrite rules for the algebra which would optimize the user expressions by rewriting the queries to say, eliminate unnecessary for expressions, etc. Our system accepts queries in the algebraic form and so we did not translate any XML query language to this algebra. However, we feel that the syntax of

the algebra is such that it would be difficult to translate queries written in any XML query language to this algebra, unless that particular query language, like XQuery, was syntactically similar to the algebra.

5. Conclusions and Future Work

In this paper we first gave an overview of the development of query algebras from Codd's first relational algebra [6] to algebras for extended-relational, object-oriented and semi-structured or XML data models. We showed that all algebras had operators based on Codd's original relational algebra operators and although a common set of algebra operators appeared for extended-relational models, this was not the case for object-oriented or XML data models. However, many of the object-oriented algebras had what are called "higher order" operators, i.e., operators that take other operators as parameters. Use of these higher order operators emphasized the view of an algebra as a functional language. As the focus of this paper was on XML algebras, we then discussed in detail and compared the various XML algebras proposed. We discussed stand-alone XML algebras such as IBM [3], AT&T [10] and Niagara [12] as well as Lore algebra [14] which was developed for the Lore database management system and YATL algebra [7] that was developed for YAT, an XML-based integration system. We reviewed these XML algebras from the point of view of choosing one of them to express user queries for MMXDB, which is a main memory XML database system we are developing. We decided to implement the AT&T algebra [10] which was chosen as the proposed standard for XML algebras by the W3C. We made a few changes to the algebra in order to simplify query rewrites. The AT&T algebra could be considered a functional language and we used the recursive nature of the algebra expressions to generate a query plan that could be recursively evaluated. In order to generate the query plan, we had to first study the algebra semantics and rewrite some expressions for simplicity. We also had to modify the grammar to make it non-left recursive and build a parser and syntax tree generator. Our findings show that one of the major disadvantages is that a query plan based on this algebra, once generated, is difficult to optimize. In order to efficiently execute any query plan, we also need to address storage and indexing issues. These however are not the focus of this paper and details of these issues for MMXDB are

discussed in [18]. One of the reasons we could not optimize the query plan may be due to the strategy of recursive evaluation that we used. Hence a possible direction for future work would be to try out a different evaluation strategy that would allow us to reorder operators for efficiency. Another direction would be to develop physical operators that take advantage of physical storage structures and generate a mapping from the query plan to a physical plan.

REFERENCES

- [1] S. Abiteboul, N. Bidoit, *Non first normal form relations: An algebra allowing data restructuring*, Journal of Computer and System Sciences, 33(3) (1986) pp. 361-393.
- [2] S. Abiteboul, C. Beeri, *On the power of languages for the manipulation of complex objects*. Technical Report No. 846, INRIA, May 1988.
- [3] D. Beech, A. Malhotra, M. Rys, *A formal data model and algebra for XML*, Communication to the W3C, September 1999.
- [4] P. Buneman, R.E. Frankel, *FQL-A Functional Query Language*, Proc. Of the ACM Conf. On Management of Data, 1979.
- [5] M. Chinwala, R. Malhotra, J.A. Miller, *Progress Towards Standards for XML Databases*, Proceedings of the 39th Annual ACM Southeast Conference, 2001, pp. 227-284.
- [6] E. F. Codd, *A relational model of data for large shared data banks*, Comm. of the ACM, 13(6) (June 1970) pp. 377-387.
- [7] V. Christophides, S. Cluet, J. Simeon, *On wrapping query languages and efficient XML integration*, ACM SIGMOD Conference on Management of Data, 2000.
- [8] Dadam P., et al., *A DBMS Prototype to Support Extended NF 2 Relations: An Integrated View of Flat Tables and Hierarchies*, Proc. 1986 ACM-SIGMOD Conference on Management of Data, Washington, DC, 1986.
- [9] A. Deshpande, D. Van Gucht, *An implementation for Nested Relational Database*, Proceedings of the 14th International Conference on Very Large Data Bases, April 1988, pp. 266-274.
- [10] M. Fernanadez, J. Simeon, P. Wadler, *A semi-monad for semi-structured data*, International Conference on Database Theory, London, January 2001.
- [11] P. C. Fischer, S. J. Thomas, *Operators for non-first-normal-form relations*, In Proceedings of IEEE Computer Software and Application Conference, 1983, pp. 464-475.
- [12] L. Galanis, et al., *Following the Paths of XML Data: An Algebraic Framework for XML Query Evaluation*, Technical Report, University of Wisconsin, Madison 2001.

- [13] M. Gyssens, D. Van Gucht, *The Powerset Algebra as a Result of Adding Programming Constructs to the Nested Relational Algebra*, Proc. ACM SIGMOD, 1988, pp.225-232.
- [14] J. McHugh, J. Widom, *Query Optimization for XML*, Proceedings of International Conference on Very Large Databases (VLDB), Edinburgh, Scotland, August 1999.
- [15] S.L. Peyton, *The Implementation of Functional Programming Languages*. Prentice-Hall, 1987.
- [16] G. Kuper, *The Logical Data Model: A New Approach to Database Logic*, PhD thesis, Stanford University, 1985.
- [17] T.W. Leung, et al., *The AQUA Data Model and Algebra*, Technical Report No. CS-93-09, Brown University, March 1993.
- [18] R. Malhotra, J.A. Miller, *XML Database Engines*, Submitted to Knowledge and Information Systems, July 2001.
- [19] Metamata, Sun Microsystems, *JavaCC-The Java Parser Generator Version 2.0*. http://www.webgain.com/products/metamata/java_doc.html
- [20] John A. Miller, Walter D. Potter, Krys J. Kochut, Ali A. Keskin, Ender Ucar, *The Active KDL Object-Oriented Database System and Its Application to Simulation Support*, Journal of Object-Oriented Programming, Special Issue on Databases, Vol. 4, No. 4 (July-August 1991) pp. 30-45.
- [21] Osborn, S.L., *Identity, Equality and Query Optimization*, in: K.R. Dittrich (Ed.), *Advances in Object-Oriented Database Systems*, Lecture Notes in Computer Science 334, Springer-Verlag, 1988, pp. 346-351.
- [22] W. D. Potter, K. J. Kochut, J. A. Miller, V. P. Gandham, R. V. Polamraju, *The Evolution of the Knowledge/Data Model*, in: F. Petry, L. Delcambre (Eds.), *Advances in Databases and Artificial Intelligence*, Vol. 1, 1995, pp. 263-310.
- [23] M. Roth, H. Korth, A. Silberschatz, *Extended Algebra and Calculus for \neg INF Relational Databases*, ACM TODS, 13(4), December 1988.
- [24] I. Savnik, Z. Tari, T. Mohoric, *QAL: A Query Algebra of Complex Objects*, Data & Knowledge Eng. Journal, 1998.
- [25] M.H. Scholl, *Theoretical foundations of algebraic optimization utilizing unnormalized relations*, in: G. Ausiello, P. Atzeni (Eds.), *ICDT'86*, Volume 243 of Lecture Notes in Computer Science, SpringerVerlag, 1986, pp. 409-420.

- [26] G. M. Shaw, S. B. Zdonik, *A Query Algebra for Object Oriented Databases*, Proceedings of the 6th International Conference on Data Engineering, Los Angeles, California, February 1990.
- [27] B. Vance, *Towards an Object-Oriented Query Algebra*, Technical Report CS/E 91-008, Oregon Graduate Institute, U.S.A., January 1992
- [28] S.L. Vandenberg, *Algebras for Object-Oriented Query Languages*, PhD thesis, University of Wisconsin - Madison, U.S.A., 1993.
- [29] World-Wide Web Consortium, *The XML Query Algebra*, Working Draft, December 2000. <http://www.w3.org/TR/2000/WD-query-algebra-20001204>
- [30] World-Wide Web Consortium, *XML Query Data Model*, Working Draft, May 2000. <http://www.w3.org/TR/query-datamodel>
- [31] World-Wide Web Consortium, *XQuery 1.0 Formal Semantics*, Working Draft, June 2001. <http://www.w3.org/TR/2001/WD-query-semantics-20010607>
- [32] S. C. Sheth, *QT4XML: A Query Tool for XML Documents and Databases*, Masters Thesis, University of Georgia, July 1999.
- [33] S.E. Hudson, CUP Parser Generator for Java Version 0.10j, <http://www.cs.princeton.edu/~appel/modem/java/CUP/#LICENSE>.
- [34] JTB Java Tree Builder Version 1.2.2, <http://www.cs.perdue.edu/jtb/index.html>.
- [35] A. Makinouchi, *A consideration on normal form of not necessarily normalized relations*, Third International Conf. on Very Large Databases, Tokyo, 1977, pp. 447-453.
- [36] E. F. Codd, *Relational Completeness of Data Base Languages*, Data Base Systems, Courant Computer Symposia Series, PrenticeHall, 6, 1972, pp. 65-98.

APPENDIX A

Non-invertible unnesting

The most important property of the nest and unnest operators of extended-relational algebras is that while unnest is always the inverse of nest, it is not always the case that $\text{nest}(\text{unnest}(r)) = r$. 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 as Partitioned Normal Form (PNF). The relation r_1 (see Figure 1) is in PNF since the dependency $\text{dept} \rightarrow \text{projects}$ holds. However this is not the case with the relation $r_1 \cup r_2$ of Figure 2. Several of the NF^2 algebras [23, 1, 9] impose the PNF restriction. The AIM algebra has the keying (χ) operator which is introduced to eliminate the problem of non-invertible unnesting. This operator appends a key column to a relation before it is unnested and then re-nested, and this ensures that nesting after an unnest will result in the original NF^2 relation. It has been shown in [28] that algebras which impose the PNF restriction are in reality no more powerful than the relational algebra. 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.