The Aqua Data Model And Algebra

Theodore W. Leung¹
Gail Mitchell²
Bharathi Subramanian³
Bennet Vance⁴
Scott L. Vandenberg⁵
Stanley B. Zdonik⁶

Technical Report No. CS-93-09
March 1993

¹ Brown University Box 1910, Providence, RI 02912
² Brown University Box 1910, Providence, RI 02912
³ Brown University Box 1910, Providence, RI 02912
⁴ Oregon Graduate Institute of Science and Technology
⁵ University of Massachusetts at Amherst
⁶ Brown University Box 1910, Providence, RI 02912
THE AQUA DATA MODEL AND ALGEBRA

Theodore W. Leung,† Gail Mitchell,*, Bharathi Subramanian,†
Bennet Vance,† Scott L. Vandenberge,† Stanley B. Zdonik*

Abstract

This paper describes a new object-oriented model and
query algebra that will be used as an input language
for the query optimizers that are being built as a part
of the EREQ project. The model adopts a uniform
view of objects and values and separates syntactic,
semantic, and implementation concerns. The algebra
addresses issues of type-defined equality and duplica-
tive elimination as well as extensions to bulk types
other than sets.

1 Introduction

Recently, a great deal of work has been done on the
topic of object-oriented query algebras [27, 22, 11]
and the modeling of bulk types [4, 25, 19]. These
proposals as well as those of other researchers on the
topic have explored some of the fundamental issues
and provided the starting point for the work reported
here. AQUA (A QUery Algebra) is the result of a
joint effort among researchers who have participated
in the design of previous algebras [26, 30, 31].

AQUA has been designed to address a number of
detailed modeling issues that we believe needed fur-
ther work, but the overarching goal for this work has
been the design of an algebra that would serve as the
input to a broad class of query optimizers. In this
way, it could be used as a de facto standard in the
construction of object-oriented query optimizers. It
would serve as the target language for user-level query
languages. Any user language for which there was a
translator to AQUA could then be processed by any
of the optimizers that are designed for AQUA. Thus,
AQUA is an intermediate language between the user’s
query and the query optimizer.

AQUA and the data model on which it is based are
strongly typed and are designed to deal correctly and
uniformly with abstract types. It should be kept in
mind that although we often talk about tuples in this
context, they are abstractions as well and are not the
stored representation of the objects.

AQUA is closed in the sense that all of its oper-
ators return objects that are defined in the model.
It should also be pointed out that although AQUA
supports updates through the use of methods, we do
not discuss this here.

Any query language or algebra must be embedded
in some data model. We have attempted to provide a
simple model that would be general enough to cover
the modeling concepts in other object-oriented mod-
els. We have adopted a model in which everything is
an object in the sense that it has a well-defined inter-
face and can be referenced from other objects, yet,
we also support value-based semantics as described
later in this paper.

AQUA has been designed to be very general. We
have purposely included a rich set of operators that
could directly simulate the operators in other pro-
posals. AQUA is not a minimal set of operators.
Redundancy allows us to more easily accommodate
many query languages, and, at the same time, allows
us to have a powerful and varied set of optimization strategies. Operator redundancy supports optimization by making query rewrites possible.

Another goal of our work 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 lists or arrays. 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. The presentation in this paper covers two bulk types, sets and multisets. The discussion of more complex, ordered types is beyond the scope of this paper [29].

Some of the operators that one would expect to find in any algebra appear in AQUA as well. They have, however, been generalized to deal with many data models and many possible bulk types. For example, our Join operator (see Table 4) takes the standard three arguments - the two input sets or multisets and a matching predicate. It also takes an extra argument which is a function of two inputs that combines every pair of objects from the two input sets that satisfy the matching predicate. This can cover Joins that produce sets of pairs or Joins that behave like Semi-joins, for example. If the two input sets were sets of lists, it could combine the two elements by concatenation, thereby producing a set of lists.

AQUA is currently being used as the input language for two prototype extensible optimizers, Epq at Brown University [21] and Revelation at OGI [10]. It appears to give us the power that we need for expressing complex queries while, at the same time, it seems to cover the functionality of all the query algebras of which we are aware.

2 Related Work

One of the primary goals of AQUA 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 (up to a point)). 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 [31], ENCORE/EQUAL [26], and Revelation [30].

AQUA is intended to support large numbers of bulk types and to do so in a flexible, uniform way, such that the addition of other bulk types later on will be straightforward. [6] proposes a meta-level algebra for collections of complex objects with identity and also includes some transformation rules for optimization. This algebra, however, does not correspond to a specific data model but rather to a higher-level notion of collections of objects. Its operations and rules are templates that are intended to be “instantiated” in actual systems according to certain parameters of the specific data model being implemented. Thus it take a different approach to generality than does AQUA. It also does not support several of the constructs of AQUA (including grouping and immutable semantics). EXTRA/EXCESS also attempts to support a large number of bulk types, but does not explicitly provide sets (they are provided only by eliminating duplicates from multisets). The inclusion of a union type is not new (see [17]), but we provide it with a clean algebraic interface using both tagcase and typecase constructs to be fully general. The impor-
tance of being flexible about the addition of new bulk types has been established (see [19]); the modularity of the AQUA approach facilitates this to an extent by following a rationale similar to that of Rozen and Shasha (see [25]) in several respects.

In attempting to support both values and objects, some systems (e.g. EXCESS [31]) choose to support only values in the type system and to model objects by using explicit identifiers. Other systems (e.g. Smalltalk and ORION [12, 5]) choose to support only objects in the type system and to model values as a special case of objects. IQL (see [1]) defines two separate languages, one enforcing object identity and one not supporting it at all. Our characterization of the distinction between “objects” and “values” as the difference between entities (objects) with mutable and immutable semantics provides a much cleaner formalism, and was partially inspired by systems such as Larch [13]. 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; see Section 3.1). Buneman and Ohori (see [8]) exhibits a similar philosophy, though, in its distinction between a kind and a type.

C++ [28] has a notion of “const” that is similar to our notion of “immutable”, but in C++ this notion is part of the type system, and thus causes a variety of problems that motivated us to separate type and semantics. Eiffel [20] makes a distinction between reference and copy semantics, but not between mutable and immutable semantics. Unlike ILOG (see [14]) 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 (see Section 3.3).

It has been pointed out by Atkinson et al (see [4]) that in object-oriented systems, a type may supply its own method for testing equality. This capability, however, introduces problems such as what is the meaning 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 seems to be without precedent in that it avoids any notion of equality other than object identity but still allows the (non-deterministic) creation of sets whose members are determined by an arbitrary equivalence relation. Many models (e.g. MDM [23]) do not have this flexibility.

Most “pure” object-oriented models ([12, 18], and others) provide and enforce encapsulation of data types. In AQUA our 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 to be described using a single uniform type system. This is similar to the ADT concept provided by Postgres [24], but 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 [9], 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 is in AQUA). AQUA can, of course, emulate these features of Postgres and EXCESS. 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 [3], but in their 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.

Several of our operators resemble operators of ENCORE/EQUAL, EXTRA/EXCESS, and Revelation. See Section 5 for detailed descriptions of the operators. Fold comes directly from Revelation, but we have adapted it for use with both sets and multisets, as it is basically a form of structural recursion ([7]). Apply is similar to mapping-style operators of other models; it is closely related to the SET-APPLY operation of EXCESS. Our nest and unnest operations are generalizations of those defined in ENCORE/EQUAL, and the group and set operations are also found in EXCESS. Dup_elim for both sets and multisets (as we’ve defined it) and convert appear to be original to this model, with dup_elim being by far the more interesting. AQUA’s dup_elim can be thought of as a generalization of other duplicate elimination operators (e.g. that of ENCORE/EQUAL [26]). Our binary join operation is similar to the n-ary Image operator of MDM [23], but differs from it in that we separate the join predicate from the function to be applied to matching pairs; the idea of this is to enhance optimization by making certain queries (e.g. equijoins) easier to recognize. A set-theoretic choose operator appears in the algebras of Osborne and MDM (see [22, 23]). Non-determinism is also present in [2], 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. Also, the decision to make the boolean operators (and, or, and not) full-fledged algebra operators, rather than constructs available only in certain parts of the language (e.g. predicates), as in the relational algebra, EXCESS, and Straube’s algebra (see [31, 27]), adds to the flexibility of the algebra. Finally, the type parameter to union, difference, and intersection is similar to that used in EXCESS.

3 The AQUA Data Model

The AQUA data model is founded on the notions of strong typing and abstract data types. In the AQUA data model, an object has a type, and its state is accessed and modified via a well defined set of interface functions. The state of an object is a mapping from AQUA objects to mathematical values. 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.

Everything that is stored in an AQUA database is an object. Database objects can have global names which facilitate access to the corresponding object. This approach allows the model to have a uniform flavor. The expressive power of the model is not crippled by this uniformity, as we shall see below.

The uniformity of the data model allows us to easily and consistently define algebraic operators over
collections of objects and values.

3.1 A Three Tiered Object Model

A common view of types is that they are a description of various aspects of the meaning or behavior of that type. In the AQUA type system, we have separated the notion of “syntactic” type from the notion of “semantic” type. Types are a syntactic property of names. Each type has an associated non-empty set of semantics, each of which provides a description of meaning and behavior that objects of the type can have. Every type provides a set of interface functions, since everything in AQUA is an object. The interface presented by the type is a set of signatures for the interface functions. The meaning and implementation of the interface functions is determined by the semantics and implemention of a given instance of the type.

An AQUA type is defined recursively:

\[ B = \{ \text{integer, float, boolean, string} \} \]

(the set of base types)

\[ (\text{name, hier, } C(m_1 : t_1, ..., m_n : t_n)) \]

\( m_i \) is a name, \( t_1, ..., t_n \) are types or objects, and \( C \) is a type constructor.

The hier symbol represents the set of immediate supertypes of the type being defined. The data model supports subtyping via the notion of substitutability. Since our types are syntactic, substitutability is also syntactic. We use the symbols \( \sqsubset \) and \( \sqsupset \) to indicate subtype and supertype relationships between types. Functions have types, although our notation (described below) only allows the instantiation of particular functions. Type equivalence is by name.

The semantics of a type might loosely be thought of as a Larch [13] specification, which axiomatically describes properties of the operations on a type. The particular language used for describing semantics is a topic of our current research.

Specifying semantics separately from types allows different instances of the same type to have different behaviors. For example, making this distinction aids us in our goal of viewing all entities as abstract data types by moving the traditional distinction between objects and values into the semantics layer. This is discussed in more detail below. At present, we define two possible semantics, mutable and immutable. Objects with mutable semantics may update their state, while objects with immutable semantics may not.

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 commutativity for deques might be written as \( \text{notEmpty}(q) \Rightarrow \text{enqueue}(\text{deque}(q), x) = \text{deque}(\text{enqueue}(q, x)) \). Commutativity axioms for sets might include: \( \text{select}(p2)(\text{select}(p1)(s)) = \text{select}(p1)(\text{select}(p2)(s)) \) or \( \text{join}(p, f, A, B) = \text{join}(p, f, B, A) \) (see Section 5.1 for a complete description of the AQUA join operator).

Each semantics of a type may have multiple implementations, resulting in the third tier of our three tiered type system. An implementation is a description of how an instance of a type with a particular semantics is to be implemented (in terms of data structures and algorithms). The specification of implementation allows the database or a 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. A result of our approach is that type can be determined statically (at compile-time), but semantics and implementation can only be determined at run time.
3.2 Constructing Types

A type constructor is a meta type which defines a family of types. The Set type constructor defines the family of types that includes Set[Person], Set[Department], etc. New types are created by instantiating the meta type with a specific type, like Person or Department.

The interface of a new type is determined by the particular type constructor and its parameters. AQUA types with user-defined interfaces may also be defined using a special "abstraction" type constructor, defined below. All types defined using the abstraction constructor must provide their own new operation for creating objects of that type. The operators of the AQUA algebra are the methods on the AQUA type constructors.

We define the following standard type constructors; Set[T] – a collection of unique entities of the same type; Multiset[T] – a bag of entities of the same type (i.e. an entity may appear more than once in a multiset); Tuple[l1 : T1, ..., ln : Tn] – a fixed-length list of labeled entities of (possibly) different types; Union[l1 : T1, ..., ln : Tn] – a tagged union, similar to that found in many programming languages; Function[T1, ..., Tn, T] – a named list of types, the last of which is the output type of the function (the others being the input types); Abs[T1, ..., Tn, f1 : T1, ..., fn : Tn] – abstraction. An abstraction is a new abstract data type and has a collection of named functions, which form the interface of the abstraction. (We leave the representation of the abstraction to the implementation and do not discuss it here); N-dimensional array[T](n) – an n-dimensional collection of entities of the same type; individual entities can be accessed directly. Only one of the dimensions may be of variable length; List[T] – a sequence of entities of the same type; Tree[T] – a tree whose nodes all contain entities entities of the same specified type; and Graph[T] – a graph whose nodes all contain entities of the same specified type. Lists, trees, and graphs are not discussed further here; see [29] for a complete description of them. N-dimensional arrays are a subject of our future research.

Abstractions, tuples and unions are subtypable. Functions are subtypable using the standard contravariance rule. Sets and multisets are not subtypable – according to substitutability, if Set[Dogs] ⊆ Set[Animals] then we could insert the elephant Dumbo, an animal, into a set of Dogs, which is incorrect.1 A similar argument shows that multisets are not subtypable.

3.3 Our Approach to Values

Mutable and immutable semantics are the key to incorporating the traditional notions of object and value into the AQUA model. In the discussion that follows, we use object to refer to the traditional notion of object, and use AQUA object to refer to objects as they appear in AQUA. There are (so far) two possible semantics for AQUA objects, mutable and immutable. Other semantics can be defined, and AQUA objects may have any semantics that has been defined in the system and is applicable to that type.

AQUA supports the traditional notion of values via immutable semantics for objects. If an object is immutable, its contents can never change, and it becomes impossible to detect whether or not it is being shared — it takes on the role of a value. The AQUA base types only support immutable semantics. Traditionally, objects are used to achieve sharing, and values are used to prevent sharing.

The type system will allow an immutable AQUA object to be assigned to a variable containing a mu-

1 This is known as proof by pachyderm
table AQUA object. In general, when assigning an AQUA object to a variable, the semantics of the new object becomes the semantics of the variable. For example, semantics allow us to create a single type, say Person, which has mutable and immutable semantics. We can then create a mutable Person and an immutable Person, and use either one where a Person is required. The behavior of the program depends on whether or not the Person is mutable or immutable at the time that the code is executed. As discussed in the next section, this allows us to have type compatible Person “objects” and Person “values”.

3.4 Examples

We will use the following schema definitions to show the use of the abstraction type constructor, and to set the stage for some sample queries described later. A Company is an abstract data type which uses a tuple as its representation, and which supplies a set of methods to access and update its name and address. Person is similar, but it also has an Employer field which is of type Company. We also have a set of Company objects, Companies = Set[Company] and a set of Person objects, Persons = Set[Person]. No semantics is presented since our language for defining semantics is incomplete.

```haskell
type Company =
  abs(Tuple[name : string, address : string, 
          employer : Company] 
      name(Company) -> String; 
      change_name(Company, String); 
      address(Company) -> String; 
      change_address(Company, String); 
  )

abs(Tuple[name : string, address : string, 
          employer : Company] 
    name(Person) -> String; 
    address(Person) -> String; 
    employer(Person) -> Company; 
  )
```

4 The AQUA Approach to Algebra Design

In this section, we discuss some of the design issues related to the AQUA algebra, including the syntax of terms, the use of type parameters, and the treatment of equality.

4.1 Syntax

Expressions in the algebra are represented by terms. A term is either: a variable, constant or function symbol, a lambda abstraction of the form \( \lambda(x_1 : T_1, x_2 : T_2, \ldots, x_n : T_n) : R \), or an application, \( t_0(t_1 : T_1, \ldots, t_k : T_k)(t_{k+1} : T_{k+1}, \ldots, t_n : T_n) : R \), where \( t_0, t_1, \ldots, t_n \) are terms and \( t_0 \) must have a function type. A lambda abstraction can be a given a name. For example, the term \( \text{Name} = \text{apply}(\lambda(p) \text{select_field}(\text{name})(p))(\text{Persons}) \) is a named term returning a set containing the names of each person in a set of Persons. Apply (defined in table 1), select_field and name are function symbols, \( p \) is a variable, and \( \lambda(p) \text{select_field}(\text{name})(p) \) is a lambda abstraction.

We note that predicates must be functions with boolean return type. Predicates are composed using AQUA’s built-in operators and its term language, which is based on lambda calculus. They are normally passed as parameters to operators like select, join, exists, and forall. All queries result in the creation of a new AQUA object as the result. For example, the query discussed in the previous para-
graph would result in a new set containing the names of each person in a set of Persons.

Some of the AQUA operations are parameterized by an equality, a type, a function, a name, or some combination of these. In the next subsections we describe type parameters and equalities, before actually describing the operations in Section 5.

We adopt some conventions and notations for defining the operators. $A$ and $B$ are used to refer to the input sets or multisets; $R$ is used to denote the output set/multiset or the result set/multiset; $a$ is used to represent an element of the input set or multiset $A$; $f$, $g$ and $h$ are used to represent functions; $id$ represents the identity function; and $p$ represents a predicate. $T$ indicates the result type of an operator. Tuples are represented by $\langle \rangle$, $L$ is a tuple field name, and $a/L$ means the tuple value $a$ minus the field labeled $L$. Other notations will be defined as needed.

### 4.2 Type Parameters

The parameterized type constructors, and subtyping requirements for types, are designed to support static type-checking. One choice that was made to assist in this support is to explicitly give a result type as a parameter to some of the algebraic operations.

Many of the operations in our algebra construct instances of new types as their result. In such cases, inferring the result type is not easy in an algebra that allows multiple supertypes and union types. In order to resolve this difficulty and provide flexibility, the algebra takes the result type as an input parameter for operators in which we may not always have a unique supertype when combining inputs of compatible (but not identical) types. For example, consider the union of a set of oranges and a set of lemons. The type of the result can be either a set of “fruits” or a set of “good sources of Vitamin C” (Figure 1). To resolve this, the user has to specify the type of the result.

![Figure 1: Result type for union with multiple supertypes](image)

### 4.3 Equality

Some types may have more than one useful notion of equality. We require every type to have 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, she should also define a default notion of equality for that type. Meaningful user-defined equalities should induce an equivalence relation over all the instance of that type.

Equality is essential to the definition of some operators, like set union. A set union operation needs to eliminate duplicates in the result. A result that is a set of Persons 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 having the same state.

Identity is the default equality for set elements. If a user wishes to impose another equality 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
occurs 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.

**Dup elim** can be defined in terms of two other operators, group and choose. If \( eq \) is a binary predicate, and \( S \) is a set,

\[
\text{dup elim}(eq)(A) = \text{apply(choose o snd)}(\text{group}(\lambda(a)\text{select}(\lambda(b)eq(a,b))(A))(A))
\]

**Group** \((f)(S)\) groups the elements of set \( S \) into equivalence classes by using the \( f \) parameter, returning a set of tuples whose first field is a member of the an equivalence class and whose second field is the set of instances in the equivalence class. **Choose** take a set and nondeterministically selects one element of the set as its result. The function ‘snd’ is composed with choose and returns the second element of a tuple, in this case the set of instances in the equivalence class. The **dup elim**ed set has no duplicates with respect to the new equality, and it also has no duplicates according to identity. See Tables 1 and 2 for definitions of apply, choose, group, and select.

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.

Consider the union of two sets \( A = \{(1, a), (2, b)\} \) and \( B = \{(2, a), (2, b)\} \). Assume we want \( A \) union \( B \) 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. The result will then have three elements: \( \{(1, a), (2, b), (2, a)\} \).

Now suppose we want \( A \) union \( B \) 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 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. \((5, a)\) could legally be in the result, but is disallowed by our definition of **dup elim** – it was not in the set before **dup elim** was applied.

### Table 1: Unary Set Iterators

<table>
<thead>
<tr>
<th>Definitions</th>
</tr>
</thead>
<tbody>
<tr>
<td>apply((f)(A)) = ({f(a)</td>
</tr>
<tr>
<td>select((p)(A)) = ({a</td>
</tr>
<tr>
<td>exists((p)(A)) = (\exists a \in A. p(a))</td>
</tr>
<tr>
<td>forall((p)(A)) = (\forall a \in A. p(a))</td>
</tr>
<tr>
<td>mem((a)(A)) = (a \in A)</td>
</tr>
<tr>
<td>fold((u, f, \oplus)(A)) = (\begin{cases} u, &amp; A = \emptyset \ \bigoplus f(a), &amp; A \neq \emptyset \end{cases} )</td>
</tr>
</tbody>
</table>

5 The Operators

In this section we describe the operations defined on the different types. Some of these operators can be expressed in terms of some others, leading to many redundancies in the operator set. They have been retained partly because they permit some expressions to be written with greater conciseness and clarity than would otherwise be possible, and partly because they lend themselves to specialized implementations and optimizations that can be more efficient than those of a more general operator.

#### 5.1 Set Operators

This subsection describes the set operators in our algebra. Most of these operators are derived from sim-
**Definitions**

<table>
<thead>
<tr>
<th>Operation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>set(a)</code></td>
<td><code>{a}</code></td>
</tr>
<tr>
<td><code>choose(A)</code></td>
<td>some <code>a ∈ A</code></td>
</tr>
<tr>
<td><code>group(f)(A)</code></td>
<td>`{(f(a), eqclass(a))</td>
</tr>
<tr>
<td><code>dup-elim(eq)(A)</code></td>
<td><code>R ⊆ A s.t. ∀x, y ∈ R, -(eq(x, y))</code>, and <code>∀x ∈ A, ∃y ∈ R s.t. eq(x, y)</code></td>
</tr>
<tr>
<td><code>nest(L)(A)</code></td>
<td>`{tup-concat(a</td>
</tr>
<tr>
<td><code>unnest(L)(A)</code></td>
<td>`{tup-concat(a</td>
</tr>
<tr>
<td><code>convert(A)</code></td>
<td><code>A</code> as Multiset</td>
</tr>
</tbody>
</table>

Table 4: Set Restructuring Operators

<table>
<thead>
<tr>
<th>Operation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>join(p, f)(A, B)</code></td>
<td>`{f(a, b)</td>
</tr>
<tr>
<td><code>tup_join(p)(A, B)</code></td>
<td><code>join(p, tup_concat)(A, B)</code></td>
</tr>
<tr>
<td><code>outer_join(p, f, g, h, T)(A, B)</code></td>
<td>`{f(a, b)</td>
</tr>
</tbody>
</table>

Table 5: Join Operators

<table>
<thead>
<tr>
<th>Operation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>union(T)(A, B)</code></td>
<td>`{x</td>
</tr>
<tr>
<td><code>intersect(T)(A, B)</code></td>
<td>`{x</td>
</tr>
<tr>
<td><code>diff(T)(A, B)</code></td>
<td>`{x</td>
</tr>
</tbody>
</table>

Table 2: Binary Set Operators

<table>
<thead>
<tr>
<th>Operation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>LFP(T, f)(A)</code></td>
<td><code>\bigcup_{i=0}^{∞}(f^i(A))</code>, <code>where f^0(A) = ∅</code></td>
</tr>
</tbody>
</table>

Table 3: Least Fixed Point Operator

In the rest of the subsection, we expand upon issues about some of the operators of the algebra.

The **fold** operator is a powerful operator – `fold(u, f, ⊕)(A)` reduces set `A` to a single value by applying `f` to each element and iteratively combining the results with a dyadic operator `⊕`. `u` is the result of `fold` on the empty set. For example, `set collapse` can be implemented using `fold`, the identity function, `id` and the `union` operator.

**fold**({}, `id`, `union`)([{1, 2}, {2, 3, 4}, {5}]) = {1, 2, 3, 4, 5}

Operators **exists**, **forall** and **mem** return a **boolean** value and can be used as predicate formers. `Nest` and `Unnest` have been defined in table 4 using a single tuple field name `L`. However, this definition can be easily extended to a list of field names. In such a case, `a/L` refers to the tuple value `a` minus the fields in the list `L` and `a.L` is the concatenation of all

operators, and table 5 lists the various join operators.

A list of all the operators for sets and a brief definition for each of them, is given in tables 1 through 2. Table 1 lists the unary set operators, table 2 deals with the binary set operators, table 3 defines the **LFP** operator, table 4 describes the set restructuring operators, and table 5 lists the various join operators.
the values of the fields in list $L$.

The binary set operators, **union**, **intersection** and **difference** are the familiar set-theoretic operations; however our definitions are complicated by considerations of typing. When combining two sets with a binary set operator, it is not necessary that they have the same type. It is sufficient that their elements have at least one common supertype, as the default equality of this supertype is used for comparison. So as in EXCESS [9], these operators take an extra argument that specifies the type of the result, as discussed in subsection 4.2. The result type of **union** has to be a supertype of the types of the input sets. However, in **intersection** the result type can either be the supertype of both input types or be one of the input types. In the case of **difference**, the result type has to either be the type of the first input set or its supertype.

To briefly illustrate some of the set operators, consider a query that finds all the people who live in the same city that they work in and groups them based on the name of this city. This is done by using the employer field of a Person object. We use $A.B$ as a shorthand for **invoke** ($A, B$), which invokes method B on object A.

$\begin{align*}
\text{LiveWhereWorkPeople} &= \\
\quad \text{select}(\lambda(x).x.\text{address} = x.\text{employer}.\text{address}) \\
\quad (\text{Persons})
\end{align*}$

Next, we use **group** to group the people in $\text{LiveWhereWorkPeople}$ by the city that they live in, by applying **group** to $\text{LiveWhereWorkPeople}$.

$\begin{align*}
\text{group}(\lambda(x).x.\text{address})(\text{LiveWhereWorkPeople})
\end{align*}$

The result consists of a set of ordered pairs

<table>
<thead>
<tr>
<th>Definitions</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\text{multiset}(g)$</td>
</tr>
<tr>
<td>$\text{convert}(A)$</td>
</tr>
</tbody>
</table>

Table 6: Multiset Restructuring Operators

($city, people$) where $city$ is the name of a city in which at least one person both lives and works and $people$ is a set of $Person$ objects all of whom live and work in $city$.

The various **join** operators deserve special mention due to their generality. **Join** takes a function as a parameter, thus allowing the user to define a “combining” function. The other **join** operators are similar generalizations involving a predicate and a function. Note that the union used in the definition of **outer join** is the set **union**. The resultant type $T$ of the **outer join** must be a supertype of the result types of functions $f$, $g$ and $h$, to allow unioning the results of the functions. Left and right outer joins can be expressed in terms of **outer join** with the appropriate interpretation of null values. Familiar join operators like **natural join**, **equijoin**, **semijoin** and **antijoin** are not primitives in the algebra, but they can be expressed easily in terms of the included join operators. As an example of the AQUA **join** operator, consider an implementation of a left outer join in terms of the **join** operator.

$\begin{align*}
\text{join}(true, \\
\quad \lambda(x, y)(\text{if } (x.\text{address} = y.\text{address}) \text{ then} \\
\quad \text{tup.concat}(x, y) \\
\quad \text{else} \text{tup.concat}(x, \text{null}))))) \\
\quad (\text{Persons, Companies})
\end{align*}$

The **LFP** operator is defined in table 3. The function $f$ is of type $T \rightarrow T$, where $T$ is the type of set $A$. Also, $f$ must be monotonic.
### Definitions

<table>
<thead>
<tr>
<th>Operator</th>
<th>Description</th>
<th>Conditions</th>
</tr>
</thead>
</table>
| `union(T)(A, B)` | Union of multisets | \( R \text{ s.t. } \forall x \in R, |R[x]| = \max(|A[x]|, |B[x]|) \)  
Also, \( \forall y \text{ s.t. } ((y \in A) \text{ or } (y \in B)), y \in R \) |
| `additive_union(T)(A, B)` | Additive union of multisets | \( R \text{ s.t. } \forall x \in R, |R[x]| = |A[x]| + |B[x]| \)  
Also, \( \forall y \text{ s.t. } ((y \in A) \text{ or } (y \in B)), y \in R \) |
| `intersect(T)(A, B)` | Intersection of multisets | \( R \text{ s.t. } \forall x \in R, |R[x]| = \min(|A[x]|, |B[x]|) \)  
Also, \( \forall y \text{ s.t. } ((y \in A) \text{ and } (y \in B)), y \in R \) |
| `diff(T)(A, B)` | Difference of multisets | \( R \text{ s.t. } \forall x \in R, |R[x]| = \max(0, |A[x]| - |B[x]|) \)  
Also, \( \forall y \text{ s.t. } ((y \in A) \text{ and } (y \in B)), y \in R \) |

#### Table 7: Binary Multiset Operators

5.2 Multiset Operators

Multisets support nearly all the same operations as sets, with very similar semantics in most cases. The difference between a multiset and a set is that a multiset may contain multiple occurrences of the same element. The notation used to denote multisets is \( \{\ast e_1, e_2, \ldots, e_n\ast\} \), where \( e_i \) are the elements of the multiset. We define the cardinality of an element of a multiset as the number of occurrences of that element in the multiset. The notation \( |A|_a \) means “the cardinality of \( a \) in multiset \( A \)”. We will also speak of the cardinality of a multiset \( |A| \), meaning its total element count, tallying duplicates as many times as they occur.

Most multiset operators are quite similar to the corresponding set operators, except for the fact that the input and output types are multisets instead of sets. Most of the formal definitions in tables 1, 3, 4 and 5 hold for multisets too.

The exceptions are multiset, convert, defined in table 6; and `union`, `additive_union`, `intersection` and `difference` which are defined in table 7.

In the binary operators on multisets, we find the greatest departure from the corresponding set operators. All the binary operators are based on the cardinality of the elements in the two input sets (table 7). For example, `union` in a multiset is

\[
\text{union}(\text{Int})(\{\ast 1, 1, 2\ast\}, \{\ast 1, 2, 2\ast\}) = \{\ast 1, 1, 2, 2\ast\}
\]

However, regarding the typing of the arguments and the result, binary multiset operators are similar to the set operators. We also define `additive_union` for multisets.

5.3 Other Type Operators

Besides sets and multisets, the algebra supports a host of other types. The union type along with its constructor allows creation of discriminated unions. The operations defined for the type are `union`, `tag_case` and `typecase`. `Union(U, tag, e)` creates an instance of union type \( U \) and initializes its contents to be entity \( e \) with tag \( tag \). Both `tag_case(e)` and `typecase(e)` selectively execute a set of terms based either on the tag or the type of the union instance \( e \).

Function types represent functions, which take some number of typed parameters and return a single typed result. There is no explicit type constructor for function types. Instead, instances of function types are created by the use of typed lambda expressions.

Tuples are records with named fields, with the familiar operators for instance creation (tuple), concatenation (\texttt{tup-concat}), and field selection (\texttt{select-field} or infix “.”). Sufficient care is taken to avoid duplicate field names in \texttt{tup-concat}.

Boolean is actually a type rather than a constructor. Booleans are used to represent truth values to be
tested by conditionals and provided as the result of comparisons and quantifiers (the set operators \textbf{exists} and \textbf{forall}). Operations on boolean are \textbf{and}, \textbf{or} and \textbf{not}.

Abstract data types are composite types whose elements are accessed only via a set of functions, which are called the interface. The functions are accessed via the \texttt{invoke(I, f)} operator which invokes \textit{f} on instance \textit{I}.

6 Conclusions

This paper has briefly summarized the AQUA data model and algebra. It is proposed as the 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. As a result, the set of operators is purposefully not minimal. We have illustrated its use with a few simple examples.

The AQUA data model embodies a uniform approach to objects and values. Values are simply immutable objects. They are objects in all other respects. They have an abstract interface, and they possess an identity that can be used to refer to them.

A type describes syntactic properties of objects and their methods. Semantic properties of a type are supplied by an axiomatic specification, called its \textit{semantics}, that is separated from the type definition (i.e., syntax). Immutability is an example of something that would be specified in the semantics. A given type can be associated with multiple semantics, and each of these semantics can be implemented in many ways. Currently we provide a default mechanism for determining the semantics of objects that are results of algebraic queries and we provide a mechanism for overriding this default.

This paper has discussed algebraic operators for the \textit{Set} and the \textit{Multiset} types. We also propose an extension to AQUA to include algebraic operators for other bulk types such as \textit{List}, \textit{Tree}, and \textit{Graph} [29].

Acknowledgements

Thanks to: Catriel Beeri, DARPA, Leo Fegaras, David Maier, Scott Meyers, and Hagit Shatikay.

References


