Ordered Types in the Aqua Data Model

Bharathi Subramanian[1]
Stanley B. Zdonik[2]
Theodore W. Leung[3]
Scott L. Vandenberg[4]

Technical Report No. CS-93-10

March 1993

[1]Brown University Box 1910, Providence, RI 02912
[2]brown University Box 1910, Providence, RI 02912
[3]Brown University Box 1910, Provodence, RI 02912
[4]University of Massachusetts at Amherst

Ordered Types in the AQUA Data Model*

Bharathi Subramanian Stanley B. Zdonik Theodore W. Leung
Dept. of Computer Science, Brown University

Scott L. Vandenberg
Dept. of Computer Science, University of Massachusetts at Amherst

Abstract

We present a query algebra that supports ordering among the data elements. Order is defined as a relationship between various data elements of an instance. This relationship can be a total or partial order among the elements or among equivalence classes where each equivalence class consists of one or more elements. In terms of data structures, ordered types can be viewed as graphs, trees, or lists.

Lately there has been a lot of interest in bulk types like lists, trees, and graphs that are not supported by traditional data models and query algebras. This interest is fueled by the fact that much of the data in the scientific domain is inherently ordered. Therefore, scientific applications that involve genome sequences, satellite data, scientific data, etc. require database support for ordered data structures like lists, trees, and graphs. In this paper, we discuss the AQUA query algebra and its associated data model with special emphasis on ordered types and their operators.

*Research supported in part by the by the Advanced Research Projects Agency under ARPA order No. 18 and administered by U.S. Army Research Laboratory under contract DAAB-07-91-C-Q518.

1 Introduction

There are many applications [2,3,6] in which ordered types are required. Scientific applications have a need to store ordered types such as time-series data and genome sequences, and textual databases store information that is structured as a tree. These applications store huge volumes of data and must locate information from these structures very efficiently.

Query languages and algebras support declarative retrieval from a database. They are based on a set of high-level operations over collections of objects. These operations hide the looping structure that would be present in an algorithm that executes them. By and large, these operations have been confined to manipulations of sets. While there has been some recent work on extending query languages to other bulk types like sequences [4,8], additional research is needed.

This paper presents an extension to the AQUA (A QUery Algebra) query algebra [7] to include ordered types like graphs, lists, and trees. N-dimensional arrays are a topic for future work. We begin by defining algebraic operations over graphs. Graphs are used as the fundamental building block out of which the operations for the other types are derived. Sequences and trees are viewed as specialized graphs. Duplicates are introduced into these graph structures through a notion of a *cell* type.

We could argue that graphs and trees could be viewed as nested list structures, but the onus of maintaining the structure is placed on the user. For example in a tree structure, the user has to prevent two nodes from pointing to the same "child" list. Also, viewing trees and graphs as types in their own right allows us to utilize their specialized properties for query optimization and gives us more flexibility in defining operations over them. This distinction might also help in other related areas like specialized storage structures to speed access and specialized index structures for querying.

This paper focuses on an algebraic approach to queries over ordered types. A graph $G = (N, E)$ with node set N and edge set E can be used to express an order over the node set N. To do this, we say that for any two nodes $a, b \in N, a \leq b$ *iff* $(a, b) \in T$, where T is the set of edges in the transitive closure of the graph G. Such an order can be restricted further to produce a tree or a list (i.e., a total order).

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

In several cases, an operation on an ordered type will not return an object of the starting type. We do not view this as a violation of the closure property. In our view, closure should require that an operation will return an object of a type within the model. For example, a **select** over a tree is not guaranteed to produce another tree; however, it will always produce a list of trees which is a perfectly good type in the model. Such a result can be composed with other operators for that type.

This paper first briefly introduces the AQUA data model. Next, it examines some related work, and then describes our approach to ordered types. This is followed by a discussion of the specific operators that we support for graphs, trees, and lists. We close with a few examples of how these operators are used and some suggestions for future research.

2 AQUA Model

The AQUA query algebra [7] is based on an object-oriented data model. All objects have identity, and these identities allow us to distinguish between objects using identity-based equalities.

The type-system in AQUA is a 3-tiered system. New types are created using the existing type constructors and types. A type is defined recursively as $(name, hier, C(m_1 : t_1, ..., m_n : t_n))$ where m_i is a name, $t_1, ..., t_n$ are previously defined types and C belongs to the set of type constructors defined in the model. *Integer, boolean, float,* and *string* are the base types provided. *Hier* is the set of immediate supertypes. Type equivalence is by name. Also, every type provides a set of interface functions, since everything in AQUA is an object.

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

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

Equality is essential to the definition of operators like union, intersection and other comparison-based operators. For ordered types, the default equality is identity, similar to that for unordered bulk types like sets and multisets. In the case of sets and multisets, other notions of equality are handled by providing the special operators **group** which creates equivalence classes based on a given equality and **choose** which picks a member of each class nondeterministically. For ordered types, the notion of other equalities is simulated by using these equality-specific set operators on the node and the edge sets.

2.1 Constructing Types

One of the primary goals of the algebra and the model has been to support a large number of bulk types in a uniform manner. A type constructor is a metatype which defines a family of types. The *Set* type constructor defines the family of types that includes *Set[Int], Set[Department]*, etc. New types are created by instantiating the metatype *Set[T : Type]* with a specific type, like *Int* or *Department.*

AQUA provides the following type constructors: *sets, multisets, tuples, unions, functions, cells, lists, graphs,* and *trees.* The algebra also supports the *abs* constructor that allows creation of new abstract types. The *operators* of the AQUA algebra are the methods on the AQUA type constructors.

2.2 Duplicates

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

3 Related Work

Much of the previous work with ordering deals with order as in sequences or arrays. Beeri and Kornatzky [1] discuss trees in their paper. However, there is no known work with directed acyclic graphs or graphs.

Beeri and Kornatzky [1] propose an object-oriented query processing paradigm where the objects are built of primitive objects, an explicit object identity type constructor, and bulk type constructors. Then operations and optimizations are presented, which apply to any bulk type constructor definable in their paradigm. In this approach, lists, arrays, and trees can all be defined, and a subset of the useful operations on such structures is described in the paper. These operations include a "pump" function, which is similar to AQUA's **fold** operation. Since the operations described in [1] are intended to be applicable to any bulk type, not just to lists and trees, they are too general for our purposes – we wish to distinguish between ordered and unordered types, and provide a richer set of operations. Furthermore, many of the operations listed in [1] are not described precisely, and their existence is assumed. Here we remove that assumption. Graphs are not discussed in [1], and it is not clear how (or if) they would fit into the paradigm presented there.

MDM [8] talks about a query algebra to support lists in an object-oriented data model. Operators from a discrete, linear-time temporal logic provide the basis for the algebra. The salient feature of the algebra is the extension of the predicate language to allow position-dependent queries, which adds a lot more

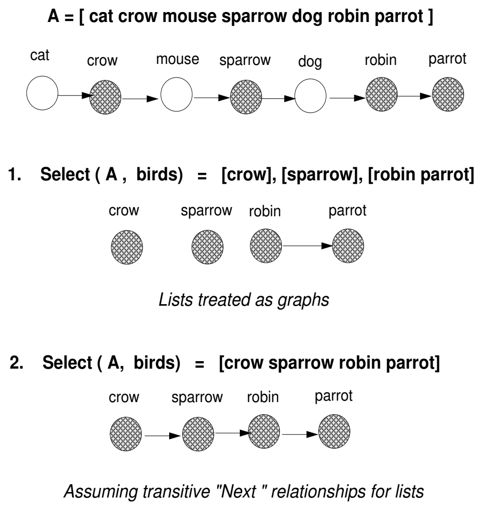

Figure 1: Select for Lists

flexibility to the kind of queries that can be posed to the database. Union and difference are similar to the corresponding operators in EXTRA/EXCESS [9]. However, the MDM algebra does not provide for operations on trees or graphs.

The NST algebra [5] is specifically designed for structured office documents and is an extension of relational algebra. The data model is based on nested sequences of tuples. It tries to maintain the order of the input lists whenever possible, with a higher preference for the order of the first input list. For example, in union, elements are concatenated and duplicates from the second list are eliminated. As a result, most of the operators are not commutative. Duplicates are allowed in lists, however union and intersection eliminate duplicates from the result set. A tree-like structure in a document (paragraphs under sections) is handled by treating it as a nested sequence of sequences.

Rs-operations [4] are sequence operations that are based on pattern matching. Along with these operations, sequence logic (SL), which is a first-order logic, is also introduced. Ginsburg and Wang define a set of powerful operations based on regular expressions, which act as a kind of template for the operation. However, the manipulation of the input lists seems to be more at a physical level rather than at a logical level.

The EXTRA/EXCESS system [9] contains an array type constructor; arrays can be fixed- or variable-length and can contain entities of any EXTRA type. The elements of an array are accessed using their array indices, but there is no ability to traverse from one element to another in these arrays. Operators are provided for extraction of elements and subarrays, for creating and concatenating arrays, and for applying a function to all elements of an array. The system provides no support for lists, trees, or graphs, and multidimensional arrays are constructed as arrays of arrays. Thus the arrays of EXTRA/EXCESS bear more resemblance to AQUA's N-dimensional arrays (which will be discussed in a future paper) than to the structures described in this paper.

4 Ordered Types

In this section, we describe *ordered* bulk types and the operations on them. Order in our setting is a mechanism to specify a "precedes" or "follows" relationship between pairs of elements. In its most general form, this kind of relationship can be represented as a graph, where elements are represented as nodes of the graph and edges between elements represents explicit precedence relationships. "Precede" is an antisymmetric relation, i.e. if a precedes b and b precedes a, then a and b are equivalent. As a parallel, strongly-connected components in graphs (a strongly connected component is one in which all nodes are reachable from each other) could be viewed as an equivalence class. In a sense, all nodes in the strongly-connected component are reachable from the same set of nodes in the graph and are equivalent for reachability queries.

Graphs form the basis for our definition of ordered types. Lists and trees are specialized forms of graphs. Besides the obvious restriction that the underlying structure be a tree (or a list), we impose an additional constraint of transitivity. In other words, if there is a directed path $a_0, a_1, \cdots, a_n$ in the tree (list), we assume that there is an implicit edge between a_0 and a_n. Note that these implicit edges are not actually present in the tree structure. To see why this transitivity assumption is *natural* when viewing lists, consider selecting birds from a list of animals A = [cat crow mouse sparrow dog robin parrot] (Figure 1). Treating list A as a graph would give us a set of graphs instead of a list [1]. Most of us would expect the select to return [crow sparrow robin parrot].

However, note that the edges between crow and sparrow, sparrow and robin did not exist in the original list though they are there in the *expected* result. The implicit assumption here is that the relationship between the elements is transitive. A similar example can be used to show that transitivity is a natural assumption for trees as well. Note that the *transitivity* property ensures that the resultant type is the same as the input type (Lists $\mapsto$ Lists and Trees $\mapsto$ Trees or a set of Trees). As a result of this property, the behavior of the operators for graphs is slightly different than that for lists and trees.

Sets in the AQUA model are at the other end of the spectrum; they can be viewed as the most *unordered* type of graphs, as sets can be represented as graphs with empty edge sets. We explore this connection in greater detail in section 5.6.

[1] Assuming we ignore the typing issues for the moment.

4.1 Graphs

Graphs are defined as a set V of nodes, and a set E of directed edges between the nodes. An edge is defined as a pair of nodes and the direction of the edge is from the first node to the second[2].

The type constructor for graphs is defined as *Graph[T]*, which constructs a graph type consisting of objects of type T as the nodes. Edges in the graph are tuples consisting of a pairs of nodes of type T. As mentioned earlier, this definition does not allow duplicate objects as nodes. Duplicates are handled by using a graph of type *Graph[Cell[T]]*. Since this is used later, while defining type conversion operators on trees and lists, we use the term "cell-graphs" to refer to such graphs. Graphs, like sets and multisets, do not participate in sub-typing.

4.2 Trees and Lists

Trees and lists are also defined as a set V of nodes, and a set (or a list in the case of unordered trees) E of directed edges between the nodes. However, the underlying structure of a tree (list) instance must be a tree (or a list). Assuming edges are directed away from the root, this implies that a tree must have one node with no incoming edges and all other nodes must have a single parent (or incoming edge). For a list, there must be one node with no incoming edge and another with no outgoing edge. All other nodes in a list must have one incoming edge and one outgoing edge.

The AQUA model supports two kinds of trees, *ordered-* and *unordered-trees*. Ordered-trees are trees where there is an order between the children of a node and unordered-trees assume that there is no explicit order between the children. Ordered-trees are defined as a set V of nodes and a list E of directed edges (as opposed to a set of directed edges for unordered-trees). The relative ordering among the edges from the parent node to the child node in the list E determines the order of the children nodes.

A tree (or list) of type T consists of nodes of type *Cell[T]* and the edges between these nodes. The node typing is different from that of graphs, where the node are of type T. We do this since it allows us to handle duplicates in a consistent manner. Duplicates in trees and lists present a problem when dealing with operators that could possibly map two or more nodes of the original tree onto the same object. In such a scenario, preserving all the associated edges might violate the tree (or list) structure. As a result, we adopt the "cell" structure to avoid duplicate nodes.

The type constructor for trees is defined as *Tree[T]*, which is a tree type consisting of nodes of the same type *Cell[T]* and edges between these nodes. The type constructor for lists is similar, *List[T]* is a list type consisting of nodes of type *Cell[T]* and their associated edges. Lists and trees do not participate in subtyping.

5 Operators

In this section, we describe in detail the various operations on graphs, trees and lists. The functionality of most operators is similar across all the ordered types. The syntax of the operations is similar to the operations defined in [7], but for the sake of completeness we give a brief overview here. All expressions in the algebra are represented by *terms*. A term is either:

- a variable, constant or a function symbol
- a lambda abstraction of the form $\lambda(x_1 : T_1, ..., x_n : T_n)t : R$, where $x_1, ..., x_n$ are variables, t is a term, $T_1, ..., T_n$ are variable types. A

[2] This definition does not explicitly allow for edge-attributes but they can be simulated by placing a "dummy node" on each edge with the relevant attributes.

lambda abstraction can be given a name.

- an *application* of the form $t_0(t_1 : T_1, ..., t_k : T_k)(t_{k+1} : T_{k+1}, ..., t_n : T_n) : R$, where $T_1, ..., T_n$ are types, $t_0, ..., t_n$ are terms and t_0 has a function type.

Predicates are functions with *boolean* return type, and are composed using AQUA's built-in operators and its term language (which is based on lambda calculus). Predicates are passed as parameters to operators like **select**.

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

5.1 Graphs

This subsection defines the algebraic operators on graphs.

- **Graph**(x) creates a graph consisting of the node x.
- **Nodes**(G) returns a set of all the nodes of graph G.
- **Edges**(G) returns a set of all the edges of graph G.
- **Sources**(G) returns a set of nodes of the graph G, that have no incoming edges.
- **Sinks**(G) returns a set of nodes of the graph G, that have no outgoing edges.
- **Im_ancestor**(G, x) returns a set of nodes that have edges leading into the node x in graph G. This could be used for graph traversal.
- **Im_descendent**(G, x) returns a set of all the nodes that have edges from node x in graph G. This operator can also be used for graph traversal.
- **Union**(G, H) unions two graphs G and H. The resultant node set is the union of the node sets of graphs G and H. The edge set of the resultant graph is the union of the two input edge-sets of graphs G and H. These unions are based on the default equality.
- **Intersect**(G, H) is similar to Union. This operation finds the intersection of the two graphs G and H to return the common sub-graph. The resultant graph consists of all the nodes and edges that are present in both G and H, i.e. the intersection of the node sets and the edge sets of G and H.
- **Select**(G, p) selects a sub-graph of graph G, based on the nodes that satisfy predicate p. All the edges between selected nodes, from the input graph, are present in the resultant graph. Predicates are discussed in greater detail in section 5.5.
- **Apply**(G, f) applies the function f to each node of the graph to transform the existing object into a new object. The edge-set remains the same unless two or more nodes from the old graph map to the same node in the new graph. In such a case, the edges of all the "combined" nodes are unioned to obtain the edges for the new node.
- **T_closure**(G) returns the transitive closure of the input graph G. So in effect, the relationship between the nodes is assumed to be transitive and the new edges created as a result of this property are added.

5.2 Update Operations

These are operations that are used for updating or modifying the graph. Due to the restriction that graphs must not have duplicate nodes, any duplicate nodes obtained in the course of an operation will be "merged" in the resultant graph along with the related edges, as in the **apply** operation.

- **Append**(G, H, *from_node*, *to_node*) appends graph G to graph H, by adding an edge between the *from_node* and the *to_node*. Since the second graph can be attached anywhere in the first graph, the *from_node* can be any node in graph G and the *to_node* is a node in graph H.

- **Add_node**(G, x) adds a new node x to the graph G.

- **Replace_node**(G, x, y) replaces the node x in graph G with the new node y.

- **Delete_node**(G, x) deletes the node x and the edges associated with it from the graph G.

- **Add_edges**(G, S) adds all the edges in set S to graph G. Note that no duplicate edges will be added. Furthermore, any edge without valid endpoints will not be added to the graph, i.e. the edge is added only if both its endpoints are in the node-set of G.

- **Delete_edges**(G, S) deletes all the edges in set S from the graph G.

- **Convert_to_tree**(G) converts graph G to a tree type. However, the graph must already possess a tree-structure. The operation just converts the type of the object, and all the nodes are converted to "cells".

- **Convert_to_list**(G) is similar to the **convert_to_tree** operation. Graph G is converted to a list type, and all the nodes of the graph are transformed into cells. The graph must have a list structure before applying **convert_to_list**.

5.3 Trees

In this section, we briefly describe the operators on trees. Most of the tree operators have been derived from the corresponding graph operators, so in the following paragraphs we shall highlight the differences between the corresponding graph and tree operators. We discuss the operators for ordered-trees below; operators for unordered-trees follow logically from the ordered-tree operators. Therefore, for the sake of simplicity we use the short-form "tree" to refer to ordered-trees. Also, in all our examples we assume that all edges are directed away from the root of the tree.

The basic tree operators in AQUA are: **sources** (T), **sinks** (T), **nodes** (T), **edges** (T), **e_edges** (T), **im_ancestor** (T, x), **im_descendent** (T, x), **select** (T, p), **apply** (T, f), **tree** (x), and **find_path** (T, x). The update operators are: **append** (T_1, T_2, *from_node*, *sibling*), **add_node** (T, x, *parent*, *sibling*), **replace_node** (T, x, y), **convert_to_graph** (T), **convert_to_iso_graph** (T), and **convert_to_list** (T). **Convert_to_list** and **replace_node** are similar to the graph operators.

Note that some functions are defined only on graphs: **union**, **intersect**, **t_closure**, **add_edges** and **delete_edges**. This is mainly because such operations are not always useful in trees, and in most cases the result will not be a tree. For example, **union** on two trees with common nodes might produce a graph due to unioning the edge sets of the common nodes from the two input trees. This operation however, can be performed by "converting" trees to graphs.

The main differences between the other tree operators and their corresponding graph operators are:

- The children of a node in a tree are ordered. As a result, the operators return a list of nodes or sub-trees instead of a set. For example,

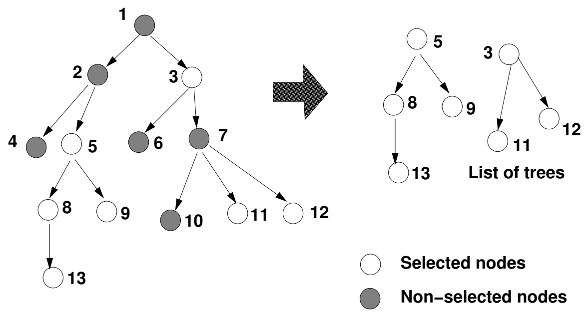

Figure 2: Select on a tree

sources(T), **sinks**(T), **im_ancestor**(T, x), and **im_descendent**(T, x) are similar to the corresponding graph operators, except that the result is a list of nodes. So, **source** returns a singleton list consisting of the root of the tree, **sinks** returns a list consisting of all the leaves of the tree, **im_ancestor** returns a list containing the parent of the given node in the tree and **im_descendent** returns a list of all the children of the given node, in order.

- The edges of a tree are transitive. Therefore, **edges** operator returns a list containing all the edges of the tree – this includes the edges that were explicitly added to the tree and the edges that were "created" due to the transitive property of the edges. The **e_edges** operator, in contrast, returns a list containing only the edges that were explicitly added to the tree. For example, **edges** on the tree rooted at 5 (figure 2) would return $[\{5, 8\}, \{5, 9\}, \{8, 13\}, \{5, 13\}]$ and **e_edges** would return $[\{5, 8\}, \{5, 9\}, \{8, 13\}]$. This property also influences any operator that "deletes" nodes (**select** and **delete_node**). Deletion of a node causes an implicit edge, i.e. an edge created due to transitivity, to become an explicit edge which can be loosely thought of as the edges used to draw the tree. So, if we just look at explicit edges, a deletion causes addition of new edges (from the list of all edges) between the the parent and the children nodes of the deleted node (Figure 3).

  Select(T, p) selects a list of sub-trees of tree T, based on the nodes that satisfy predicate p. All the edges between selected nodes from the input tree are present in the resultant graph. Any new edges "created" due to the transitivity relationship between the nodes are also added in the resultant tree (Figure 2). The ordering in the resultant list is based on the *relative* ordering of the roots of each tree in the list. For e.g., in figure 2, the tree with node 5 as the root comes "before" the tree rooted at node 3, inspite of the difference in levels. The ordering is mainly based on position – if sub-tree A is to the left of sub-tree B (assuming ordering is from left to right), then A is followed by B in the resultant list. It is a kind of depth-first ordering of the The sub-trees in the list are ordered based on the relative order in which the respective roots of the sub-trees are visited in a depth-first traversal.

- For certain operators, there are certain constraints on their behavior, as the result has to be a tree. **Add_node** in graphs just adds a node to the input graph. However for trees,

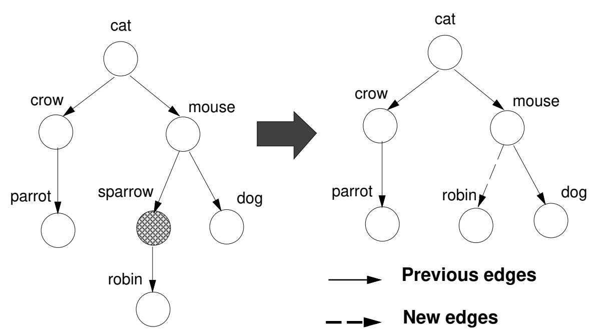

Figure 3: **Delete** ($Tree$, "sparrow")

add_node also adds an edge connecting the new node and the input tree, keeping the tree connected. **Add_node**(T, x, *parent*, *sibling*) adds a new node x into tree T, as a child of node *parent*. As the *parent* might have other children, the order of node x is determined by an extra argument *sibling*, which is the sibling node immediately preceding x. If *sibling* is not specified, x is added as the first child of *parent*. Furthermore, if the node *parent* is not specified, x becomes the root of the tree with T as its subtree.

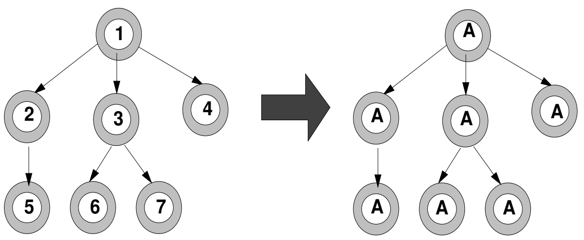

Figure 4: **Apply**(T, $\lambda(x)$ A)

In the case of **append** (T_1, T_2, *from_node*, *sibling*) the *to_node* is always the root of the second tree, hence it is not a parameter to the **append** operator. **Append** (T_1, T_2, *from_node*, *sibling*) appends tree T_1 to tree T_2, as a child of the *from_node*, after the sub-tree rooted at *sibling* (which is also a child of the *from_node*). If *sibling* is not specified, the tree T_2 is added as the first child of the *from_node*.

- Trees of type T are composed of cells that contain objects of type T. As a result, most function applications deal with the contained-object instead of the cell. So, as in the case of **apply** on a graph, **apply** on trees transforms the "contained-object" based on the parameter function f.

  Apply(T, f) applies the function f to the "content" of each cell (node) of the tree; to transform the existing object into a new object. The edge-set remains the same. This ensures that the basic structure of the tree is not modified. The resultant tree is built of new cells that contain the transformed objects (Figure 4).

Other operators that are specific to trees are: **Tree**(x) creates a tree that consists of node x. **Find_path**(T, x) returns a list of nodes encountered on the path from the root of the tree T to the node x. The last node of the resultant list is x, and the first node is the root of the tree. **Convert_to_graph**(T) converts a tree T to a cell-graph. All the edges, both implicit and explicit edges are present in the resultant graph. Note that the nodes of the new graph are composed of cells, as the tree nodes are cells. However, if we desire to have a graph with objects instead of cells at the nodes (and unioning the edge-sets of identical nodes), we could use the **apply** operator to remove the cells. **Convert_to_iso_graph**(T) is similar to **convert_to_graph**, except for the edges in the new graph. The new graph contains only the explicit edges. This operator would allow for transformations from tree to graph without any change in structure.

Convert_to_list, **replace_node**, and **nodes** are similar to the corresponding operators in graphs. The only difference is due to typing of the input and the output. For example, **nodes** on a tree returns a set containing all the nodes of the tree, similar to **nodes** for graphs. However, the tree-nodes are of cells unlike graph-nodes that are objects.

5.4 Lists

In this section, we discuss operations on lists. These operators are almost identical to the corresponding operators on trees, except for the input and output types which are lists instead of trees, and the absence of the "sibling" parameter. Any kind of add operation in trees requires a *sibling* parameter, that specifies the node after which the new node/sub-tree must be added. This is needed for trees as the children of a node are ordered. In the case of a list however, since there is only one child for every node, this parameter is unnecessary.

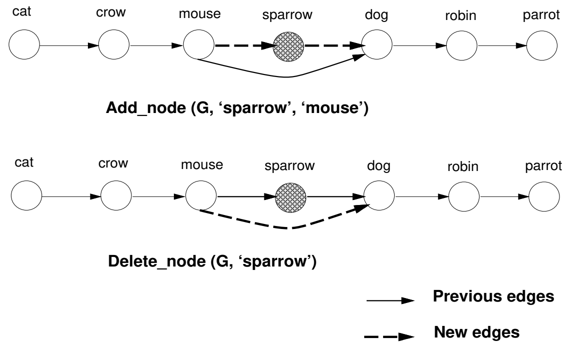

Figure 5: Add_node and Delete_node on a list

Nodes (L), **edges** (L), **e_edges** (L), **select** (L, p), **apply** (L, f), **add_node** (L, x, y), **replace_node** (L, x, y), **delete_node** (L, x), **convert_to_graph**(L), and **convert_to_iso_graph**(L) are similar to the corresponding unordered tree operators. **Add_node** in a tree does not involve deletion of any existing edges, however, in the case of lists adding a node in the middle of the list might result in deletion of an edge and addition of up to two edges (Figure 5). **Source**(L), **sink**(L), **im_ancestor** (L, x), **im_descendent**(L, x) are also similar to their equivalent graph operators but they return a single node for lists instead of a list of nodes as in trees. **Append**(L_1, L_2) is similar to **append** in trees and graphs, but the *from_node* is always the last node of list L_1 and the *to_node* is the first node of list L_2. This results in a concatenation of the two input lists.

There are list operators that do not have corresponding tree operators. **List**(x) creates a list with a single element x. **Sort**(L, f) sorts list L based on the comparator function f. f is a transitive function that given any two elements (a, b) from the list L, returns *less-than* if a should appear before b in the result list, *greater-than* if a should appear after b in the result list and *equal-to* if a and b are "equal" based on the function f. **Convert_to_tree**(L) converts list L to a tree type.

5.5 Predicates

Predicates for ordered types are similar to those for the other bulk types. This would allow for position dependent functions like **im_ancestor**(G, a) as a predicate. So, a **select** with such a predicate would return all nodes of graph G which satisfy the condition, i.e. nodes that are **im_ancestor** of node a.

$$\mathbf{select}(G,\ \lambda p.p = \mathbf{im\_ancestor}(G, a))$$

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

$$\mathbf{select}(Set\ of\ genomes, AU^{+}A)$$

A limited form of matching is possible with the current AQUA operators, and the above query can be specified with these operators albeit in a complicated fashion. However, a more succinct predicate language for sequence matching would facilitate specialized optimizations. This notion could be extended to include matching predicates for trees or graphs.

5.6 Sets as graphs

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

6 Examples

In this section we give some examples to illustrate some of the operators for graphs, lists, and trees.

6.1 Graphs

Consider an airline company, Airline1. They have a large number of airports out of which they operate, with flights connecting them (Figure 6). In our graph, the nodes represent the airports and the edges represent the flights between the airports. The nodes are assumed to have more details about the airports, besides their city names.

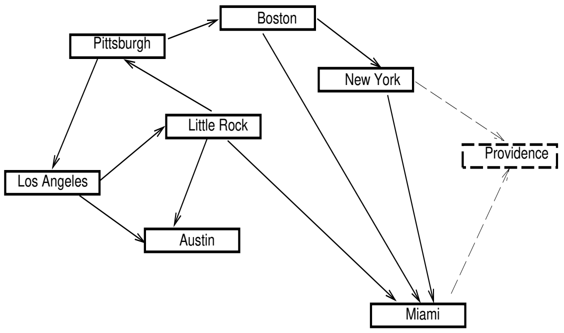

Figure 6: Flight graph for $Airline1$

type $Airport = abs(Tuple[city : String, ...]$

$city(Airport) \rightarrow String;$

$\vdots$

)

type $Airline = Graph[Airport]$

Our query is to find all the places that have a direct flight to Boston, either by $Airline1$ or $Airline2$. The basic query is to get all the **im_ancestor**s of the node $Boston$ in the combined airline map, $TwoAirlines$. The combined airline map is gotten by unioning the maps of $Airline1$ and $Airline2$.

TwoAirlines = **Union**($Airline1$, $Airline2$)

DirectToBoston =
Im_ancestor($TwoAirlines$,
choose(**nodes**(**select**($TwoAirlines$,
$TwoAirlines.nodes.city = Boston$))))

6.2 Trees and Lists

For trees, we do a search to find the word *query* in a dictionary tree (Figure 7). Since some operations on trees return lists, the example also illustrates operations on lists. *NodeWithWord* is a recursive query that exits with the word *query* as soon it is found. The query initially checks if the word is in the root of the tree. If it is not at the root, the same query is applied to all the sub-trees of the current root. For the sake of simplicity, we have assumed that boundary conditions in the query are taken care of.

NodeWithWord = QueryTheRoot($DictionaryTree$)

QueryTheRoot = $\lambda(X)$
if (**cell_content**(**source**(X)).$word =$"*query*")
exit(**source**(X))

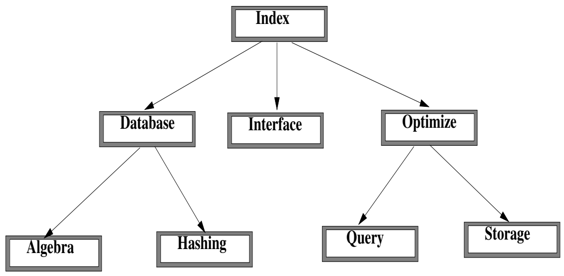

Figure 7: Searching for a specific word in a dictionary tree

$$\textit{else}\ (\mathbf{apply}(\mathbf{select}(X, \lambda(p)\, p \neq \mathbf{source}(X)),$$
$$\textit{QueryTheRoot}))$$

7 Conclusions and Future Work

This paper has described the support for ordered bulk data types provided by the AQUA data model and algebra. The primary ordered bulk type is a graph, from which we derive trees and lists by imposing constraints on the edge set. Uniqueness of tree and list nodes is enforced using the *Cell* type constructor. Important aspects of AQUA's ordered bulk type support are the consistency of operators and semantics among the various ordered types and the close relationship between graphs and sets, resulting in a very uniform data model.

Current and future research includes investigation of additional operators on ordered bulk types (e.g. LFP, as described in [7]) and predicate specification sub-languages for ordered bulk types. For example, we can specify regular expression-like predicates on ordered objects, but more powerful mechanisms (such as context-free grammars) need to be examined. Indexable ordered types in AQUA (such as N-dimensional arrays) will be discussed in a future paper.

Acknowledgments

We would like to thank Catriel Beeri, Gail Mitchell, and Sairam Subramanian for useful discussions.

References

[1] Catriel Beeri and Yoram Kornatzky, "Algebraic Optimization of Object-Oriented Query Languages," *Proceedings of the International Conference on Database Theory* (1990), 72–83.

[2] James C. French, Anita K. Jones, and John L. Pfaltz, "Summary of the Final Report of the NSF Workshop on Scientific Database Mgmt.," *SIGMOD Record* 19 (1990), 32–40.

[3] Karen A. Frenkel, "The Human Genome Project and Informatics," *Communications of the ACM* 34 (1991), 41–51.

[4] Seymour Ginsburg and Xiaoyang Wang, "Pattern Matching by Rs-Operations: Towards a Unified Approach to Querying Sequenced Data," *Proceedings of the 11th ACM Principles of Database Systems* (1992), 293–300.

[5] Ralf Harmut Güting, Roberto Zicari, and David M. Choy, "An Algebra for Structured Office Documents," *ACM Transactions on Office Information Systems* 7 (1989), 123–157.

[6] Eric S. Lander, Robert Langridge, and Damien M. Saccocio, "Mapping and Interpreting Biological Information," *Communications of the ACM* 34 (1991), 33–39.

[7] T. Leung, G. Mitchell, B. Subramanian, B. Vance, S. Vandenberg, and S. Zdonik, "The AQUA Data Model and Algebra," *Tech. Rpt. CS-93-09, Submitted to the Fourth Intl. Workshop on Database Programming Languages* (1993).

[8] Joel Richardson, "Supporting Lists in a Data Model (A Timely Approach)," *Proceedings of the 18th VLDB Conference* (1992).

[9] Scott L. Vandenberg and David J. DeWitt, "Algebraic Support for Complex Objects with Arrays, Identity, and Inheritance," *SIGMOD Proceedings* (1991), 158–167.