The AQUA Approach to Querying Lists and Trees in Object-Oriented Databases*

Bharathi Subramanian
Brown University†

Theodore W. Leung
Brown University†

Scott L. Vandenberg
Siena College‡

Stanley B. Zdonik
Brown University†

Abstract

Relational database systems and most object-oriented database systems provide support for queries. Usually these queries represent retrievals over sets or multisets. Many new applications for databases, such as multimedia systems and digital libraries, need support for queries on complex bulk types such as lists and trees. In this paper we describe an object-oriented query algebra for lists and trees. The operators in the algebra preserve the ordering between the elements of a list or tree, even when the result list or tree contains an arbitrary set of nodes from the original tree. We also present predicate languages for lists and trees which allow order-sensitive queries because they use pattern matching to examine groups of list or tree nodes rather than individual nodes. The ability to decompose predicate patterns enables optimizations that make use of indices.

1 Introduction

Database query languages have primarily focused on sets (and sometimes multisets) while in this paper we enhance them to include lists and trees. Computer applications are becoming more sophisticated and require higher-level support for managing complex data. Many advanced application areas stress the boundaries of current database querying technology and require query support over bulk types more complex than sets and multisets. For example, multi-media systems must manage many different data types, such as text, video, and audio. Here, a document can be viewed as a tree of document components; a video is a sequence of frames; a sound-track is a sequence of tones. Similarly, scientific applications often deal with RNA-sequences [28] or with multi-dimensional arrays. Other applications that can benefit from a more comprehensive approach to queries and bulk types include geographic databases, electronic libraries, vision [26], molecular biology [28], program compilation [1], natural language processing, and hypermedia applications.

All of the applications mentioned above can benefit from support for queries; however, the lack of a query formalism that captures more complex bulk types forces application-specialists to invent query packages of their own that do not compose well with standard database query languages. They often involve operators that cannot easily be decomposed into more primitive components[1].

A better approach is to develop query languages and query processing techniques that can be applied to a wider universe of bulk types [2,3,20]. This universe includes lists, trees, arrays, and graphs as well as the more conventional sets and multisets. Query optimization strategies would apply more uniformly across all of these types. Moreover, queries on arbitrary compositions of these bulk types (e.g., set[tree]) could be handled more uniformly.

In [32], we introduced an object-oriented model and a query algebra (called AQUA) for sets and multisets. In this paper, we extend the AQUA query algebra by introducing operators and pattern matching primitives specific to lists and trees. These operators were designed based on several criteria: consistency with other AQUA operators, ability to express useful queries, amenability to transformation-based op-

*Partial support for this work was provided by the Advanced Research Projects Agency under contract N00014-91-J-4052 ARPA order 8225, and contract DAAB-07-91-C-Q518 under subcontract F41100.

† Dept. of Computer Science, Brown University, Providence, RI 02912-1910.

‡ Dept. of Computer Science, Siena College, Loudonville, NY 12211

[1] Such splitting is crucial for any kind of query optimization.

timization techniques, free composability with other algebra operators, and extensibility. We do not assume any particular user-level language, but we note that our extensions to AQUA can model the user-level language described in [35,36]. The AQUA list and tree algebras have a small number of primitive operators which can be used to build other useful operators.

Query operations frequently filter out some elements of a collection type. In an ordered structure, we want to ensure that such filtering preserves the order of the elements of a collection and does not lose information by segmenting a list or tree. Our operators are stable in that the relative orderings between all pairs of elements are preserved in the result.

Many database query operators take a predicate as one of their parameters. For sets of records, predicates based on boolean combinations of simple terms are adequate. Predicates for bulk types like lists or trees, must be sensitive to the inherent ordering among the elements. Traditionally, predicates operate on single elements of a bulk type. Our predicates account for order by using pattern matching to examine groups of elements from a bulk type object. These patterns define languages (sets of lists or trees) over lists and trees. We have based our pattern language on extensions to regular expressions, since the expressiveness and tractability of regular expressions is well known.

Query algebras should be designed to facilitate optimizations by providing opportunities for algebraic rewrites. Queries in our algebra can be rewritten by decomposing the predicates into smaller pieces. The resulting sub-pieces are frequently able to make good use of indexes. In another paper [31] we propose a framework for optimizing list and tree queries, and provide some guidelines and rules that can be used for query rewrites.

This work is done in the context of the AQUA [19] query algebra that has been developed jointly among Oregon Graduate Institute, University of Wisconsin, and Brown University as a standard input language for query optimizers.

The remainder of the paper is organized as follows. Section 2 provides background and overview of AQUA. Section 3 discusses the predicate language for list and tree operators. For ease of presentation, we first describe the tree operators in Section 4 followed by some examples in Section 5. In Section 6 we define list query operators and show how they can be viewed as special cases of tree operators. In Section 7 we discuss related work and finally in Section 8 we give a summary of the paper and a brief outline of future and ongoing work.

2 Background

In this section we provide a brief overview of the AQUA data model [19] with special emphasis on those aspects that apply to ordered bulk types. All entities in the AQUA model are objects, i.e. all entities have identity and provide a set of functions which define the protocol for the object. Since every object has identity, issues of equality [29] become critical. AQUA allows equality to be specified as a parameter to some of its operators (e.g., set **union**), thereby allowing queries to use various notions of equality.

The AQUA data model consists of the following type constructors: *Set*, *Multiset*, *Tuple*, *Union*, *Function*, *Abstraction*, *List*, *Tree*, and *Graph*. Each of these types provides a collection of algebraic operators [19], which form the basis for the algebra. One design criteria for the list and tree operators was to generalize existing operators for sets and multisets when possible. For example, AQUA's sets can be viewed as trees or lists with an empty edge set. The tree (list) operators map to the corresponding set operators, and relevant set operators behave the same way on trees (lists) with empty edge sets as they do on sets.

Lists and Trees One way to view trees is as nested list structures, but this puts the onus of maintaining the tree structure on the user. For example, in a tree structure, the user has to prevent two tree nodes from pointing to the same "child" list. Viewing trees as types in their own right allows us to use their specialized properties for query optimization, storage structures, and indices.

We would like to allow duplicate objects to appear in a list or tree, but the nodes of a list or tree are a set, which does not allow duplicates. For this reason we require the elements of a list or tree to be of type *Cell[T]*. A cell is an object whose only purpose is to contain the identity of another object of the list or tree's actual element type. This allows all the nodes to be unique, but to potentially reference the same object. We will use *List[T]* as a shorthand for *List[Cell[T]]*, and similarly for trees. Most of the query operators implicitly dereference the contents of the cell to get and manipulate the object that it contains.

A list or tree, then, is a parameterized type, *List[T]* or *Tree[T]*, and is defined to have a set of nodes, V and a set (for trees, a set of lists) of directed edges, E. Tree edges are directed away from the root, list edges are from left to right. "Fixed-arity" trees have constant out-degree, and "variable-arity" trees have non-constant out-degree. We assume that trees are ordered, that is, the children of a node appear in order

from left to right.

We adopt the following notation in the rest of the paper: L is a list, T is a tree, lp is a list predicate, and tp is a tree predicate. Predicates are defined in Section 3.

A *sublist* of L is an embedded list of contiguous elements. There are two kinds of substructures for trees that are of interest. We use *subgraph* to mean a connected subgraph of a tree. A *subtree* P of tree T is a subgraph of T where the following condition holds: For all nodes n in P, either all or none of n's children in T are in P.

We represent lists by writing the elements in sequence from left to right, surrounded by [ ]. So, a list containing a, b, and c would be written as $[abc]$. Trees are represented by a preorder-based notation in which a node is followed by a parenthesized list of its children. For example, the second tree in Figure 1 is represented by $b(d\,(f\;g)\,e)$.

3 Predicates, Patterns, and Results

Query algebras operate by retrieving database objects that satisfy a boolean predicate. The power of a query algebra is strongly affected by the power of the predicates that can be used.

In this section, we present a language for describing relationships between the elements of ordered data types like lists and trees. Instead of returning individual objects that satisfy a boolean predicate, our algebra returns pieces of ordered structures which match a particular pattern.

3.1 Predicates and pattern alphabets

Operators in the AQUA list and tree algebra use a pattern to describe the objects of interest. A pattern defines a language: a set of lists or trees. Pattern predicates are written in a language that is an extension of regular expressions.

The alphabet for the list and tree predicates is defined by a set of *alphabet-predicates*. An alphabet-predicate is a unary boolean function which is applied to an object. Each alphabet predicate is satisfied by a finite number of objects in the database. An object *matches* an alphabet-predicate if the object satisfies it.

Alphabet-predicates are written as parenthesized lambda expressions, so $(\lambda(Person)\ Person.age > 25)$ is an alphabet-predicate that should be applied to an object of type *Person*. In a pattern this predicate will match any *Person* object whose age is > 25.

In order to limit the complexity of list and tree queries, we only allow alphabet-predicates to be constructed from values of stored attributes of objects[2], constants, comparison operations, and the boolean operators AND, OR, and NOT. These constraints ensure that any alphabet-predicate can be evaluated in constant time.

3.2 List patterns

AQUA list patterns are derived from regular expressions, and provide the operations concatenation ($\circ$), Kleene closure (*), and disjunction (|). We define a list pattern lp inductively. In the base cases, lp is an alphabet-predicate or the metacharacter ? (which is always TRUE). The inductive cases follow. A list pattern can be defined as a union of two list patterns $(lp_1 \,|\, lp_2)$ or as a concatenation of two list patterns $(lp_1 \circ lp_2)$. In most cases, the $\circ$ symbol is omitted. Iterative self-concatenation (Kleene closure) is denoted as lp^* (zero or more times) or lp^+ (one or more times). A more formal description of a list pattern lp is:

lp ::= $[ilp]$ | $[\![lp]\!]$ ■

ilp ::= alphabet-predicate | ? | ilp^+ | ilp^*
| $[\![ilp]\!]$ | $lp \circ lp$ | lp '|' lp ■

We allow the use of $[\![\ ]\!]$ to show grouping and to improve readability. The metacharacters ^ and \$ (written as $\hat{}lp$ and $lp\$$) indicate that lp must match at the beginning or the end of the list respectively.

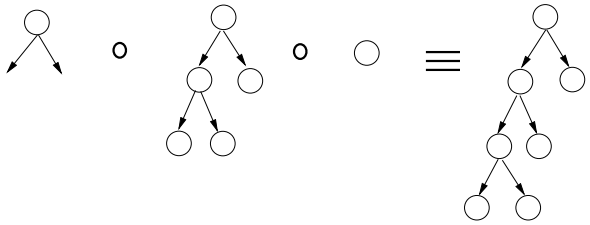

Figure 1: Using concatenation points in tree patterns

3.3 Tree patterns

We use a regular expression-like language for describing patterns for trees. Most of the operations for regular expressions generalize nicely to trees, with the exception of concatenation and operators derived from it (like Kleene closure). The difficulty arises because it is not clear where concatenation should take place. Our solution to this difficulty is to adopt the

[2]This cannot be determined by the user, since it would be a violation of encapsulation. However, the query optimizer can verify that the attributes involved are stored and not computed.

notion of concatenation points [6,33] which are used to specify where the concatenation should occur. We allow multiple concatenation points to appear in our patterns. A single concatenation point is usually denoted by the Greek letter α, and multiple concatenation points are denoted by subscripted versions of α, such as $\alpha_1, \alpha_2, ...\alpha_n$. We also require concatenation and its derived operators to be parameterized by a concatenation point (e.g. $\circ_\alpha$), so that the correct behavior is observed in the presence of multiple concatenation points.

Concatenating tree pattern tp_1 with tp_2 using concatenation point α_1 is written as $tp_1 \circ_{\alpha_1} tp_2$. For example, Figure 1 shows how the pattern $a(b(d(f\ g)e)c)$ can be written as the concatenation $[\![a(\alpha_1\ \alpha_2) \circ_{\alpha_1} [\![b(d(f\ g)e)]\!]]\!] \circ_{\alpha_2} c$.

Similarly, the iterative self concatenation operations [3] are subscripted, giving $tp^{*\alpha_1}$ and $tp^{+\alpha_1}$ for zero or more and one or more self concatenations respectively. If two trees are concatenated with a concatenation point α_1 and there is no α_1 in the first tree, the result is just the first tree. As with lists, the last iteration of an iterative self concatenation concatenates NULL to the appropriate concatenation points. As an example of iterative self concatenation, take the pattern $[\![a(b\ c\ \alpha)]\!]^{*\alpha}$. Four elements in the language defined by this pattern are shown in Figure 2.

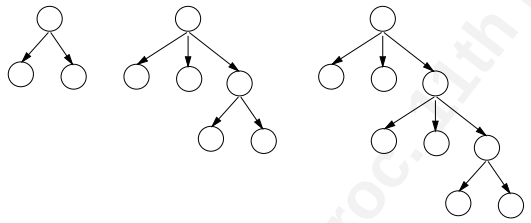

Figure 2: Self Concatenation in trees

Tree patterns provide the operations of concatenation ($\circ_\alpha$), Kleene closure ($*_\alpha$), and disjunction (|). As with lists, tree patterns can be defined inductively. In the base cases, a tree pattern tp is either a) a single node tree, represented by an alphabet-predicate, ?, or a concatenation point or b) the root of a tree, represented by an alphabet-predicate, or ?, followed by a parenthesized list of tree patterns for the list of children.

A more formal description of a tree pattern tp and a tree list pattern (list of children) tlp is:

tp ::= alphabet-predicate | ? | α | $[\![tp\ ]\!]$ | alphabet-predicate (tlp) | ? (tlp) | $tp \circ_\alpha tp$ | $tp^{+\alpha}$ | $tp^{*\alpha}$ | tp '|' tp ■

tlp ::= tp | $tlp \circ tlp$ | tlp^+ | tlp^* | tlp '|' tlp | $[\![tlp\ ]\!]$ ■

As an analog with list patterns, the tree pattern $\top tp$ matches only when tp is at the root of the tree and $tp\perp$ matches only when all leaves of tp match leaves of the tree. For example, both $\top\, b(d\, e)$ and $b(d\, e\perp)$ match the subtree $b(d\, e)$ in the second tree in Figure 1 and only $b(d\, e\perp)$ matches the subtree $b(d\, e)$ in the fourth tree. We also use $[\![\ ]\!]$ for grouping.

Since we use the list language to specify the children of any node, we can have trees in which nodes may have a nonuniform and arbitrary number of children.

3.4 Matching and Return Results

The result of a list or tree query operation is a new list or tree which contains the appropriate objects from the original list or tree. The use of patterns in list and tree queries can be viewed as a two step process. The first step is matching the pattern against the input, and the second step is determining what to return.

The alphabet for list and tree patterns is alphabet predicates, but the alphabet for lists and trees in the database is objects. In order to match list and tree patterns to database objects, the patterns and the objects they are matched against must have the same alphabet. We can transform a pattern P into P' as follows: eliminate the alphabet predicates from P by replacing each alphabet predicate ap in P with the regular expression $(x_1|x_2|\ldots|x_n)$, where $x_1,\ldots,x_n$ are the objects in the database that satisfy ap. Now the pattern and the match candidates have the same alphabet. Given an input list (or tree) L, a sublist (or subtree) I of L *matches* a pattern P if I is in the language described by P'. I is called an *instance* of P.

After a pattern is matched, the instance matching the pattern is "returned". In some situations, only a portion of the matching instance is of interest. So, we provide the ! metacharacter as a prefix to a subpattern P to specify that the largest subtree rooted at the node matching P's root be pruned from the result. For an example see the definition of **sub_select** in Section 4.

3.5 NULLs and Concatenation Points

If we view trees and lists as recursively defined types, then we recognize that all leaf nodes in a tree and the last list node have NULLs for children. We

[3] The inclusion of these operations means that some tree queries will be exponential. The performance of many such queries can be improved using our optimizations.

extend this notion (adapted from list and tree patterns) to allow concatenation points to appear in lists and trees. When a concatenation point symbol appears in a list or tree, it is treated as a labeled NULL. The only operation which is able to tell that such a NULL is present in a list or tree is the concatenation operator ($\circ$, $\circ_\alpha$), which examines the NULL's label to determine the applicability of the concatenation. This facilitates the use of our **split** operator.

4 Tree Operators

In this section we briefly describe some of the important tree query operators. AQUA also provides a range of other operators for purposes like navigating, updating, and providing structural information about a tree instance. These operators are not discussed in this paper.

Query operators The tree query operators can be classified into two sub-categories – those common to all bulk types (e.g. sets, bags) and operators that are specific to ordered bulk types. **Select** and **apply** belong to the former category while **sub_select**, **all_desc**, **all_anc**, and **split** belong to the latter category.

To illustrate the behavior of the operators we use a simple example of a family tree (Figure 3). Section 5 gives a more intricate example of trees and tree operators. Consider a family tree containing the descendants of a famous person. Each node represents a person object with a large number of attributes. However, in our example we only list the name, citizenship, eye color, and education attributes for each person in the tree. Each edge stands for the relationship "a child of" and a path in the tree stands for the relationship "a descendant of". In the rest of this section we describe the operators and how they work on the example tree.

Select and **apply** are derived from the respective set/bag operators in AQUA. However, since they operate on ordered types, the nodes of the result tree maintain the same relative order as the nodes in the input tree T. Assume T is of type *Type[S]*.

- **Select**$(p)(T)$: selects all nodes of T that satisfy the predicate $p : S \rightarrow Bool$. Here p is an alphabet-predicate which is applied to each node. If two nodes n_1 and n_2 satisfy p, n_1 is an ancestor of n_2 in the result tree(s) if and only if n_1 is an ancestor of n_2 in T. If no nodes in the path between n_1 and n_2 (excluding n_1, n_2) in T, satisfy p then there is an edge (n_1, n_2) in the result. So, **select** produces a set of trees with all the "predicate-satisfying" nodes and all the edges produced using the above guideline. The result set will contain only one tree if the root node of T satisfies p else it will contain a forest containing trees rooted at nodes r_1 ... r_n that satisfy p (and no ancestor of r_i in T satisfies p).

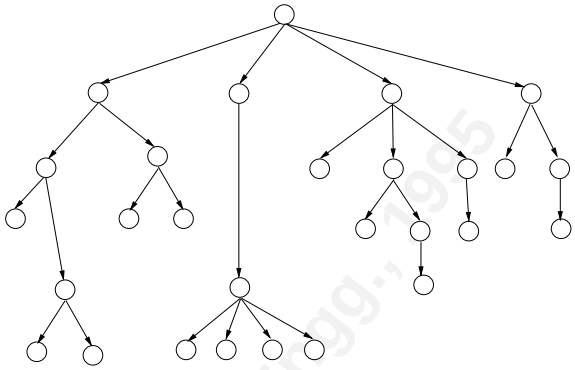

Figure 3: A Family Tree T

- **Apply**$(f)(T)$: applies function $f : S \rightarrow U$ to all the nodes of T, and constructs a tree of type *Tree[U]* isomorphic to T, containing the corresponding results.

The second category of operators also allows us to specify the relative order of the input nodes that satisfy the predicate. Most of the operators take a tree pattern tp and find the matching instances for tp in T. Figure 4 specifies a matching instance (or match) for tp = *"Mat" (? "Ed")* from tree T in Figure 3. The operator being used determines what is returned. For example, after matching, the **sub_select** operator returns the subgraphs of T that match tp whereas **all_anc** and **all_desc** return the matching subgraphs along with their ancestors and descendants respectively. These last two operators are very useful for describing where the match occurs in algebraic terms.

- **Split**$(tp, f)(T)$: For each match of tp in T, **split** creates three intermediate results: A tree corresponding to all ancestors of the match and their descendants (except the match itself); the match; and a list of all trees descended from the match. For each match, **split** applies a function f to this 3-tuple and returns a set containing the results of f for each match. To illustrate how **split** works consider the following example. Suppose we wish to split the tree T on the basis of the pattern "parent is Brazilian, one child is American". Using the

shorthand "Brazil" to stand for $\lambda(p)$ *p.citizen = "Brazil"*, "USA" for $\lambda(p)$ *p.citizen = "USA"*, and $\langle\ \rangle$ to indicate tuple formation, we can write the query as:

$$\textbf{split}(Brazil(!?^*\ USA\ !?^*), \lambda(x,y,z)\langle x,y,z\rangle)(T)$$

The result of this query is a set containing one tuple with three pieces as shown in Figure 4. The concatenation point α indicates where the match is attached to the rest of the tree (the ancestors). The α_1 and α_2 indicate where the match's descendants are attached. Note that α_1 corresponds to a subtree pruned using $!?^*$ and that α_2 corresponds to a subtree pruned because it was actually a descendant of the match.

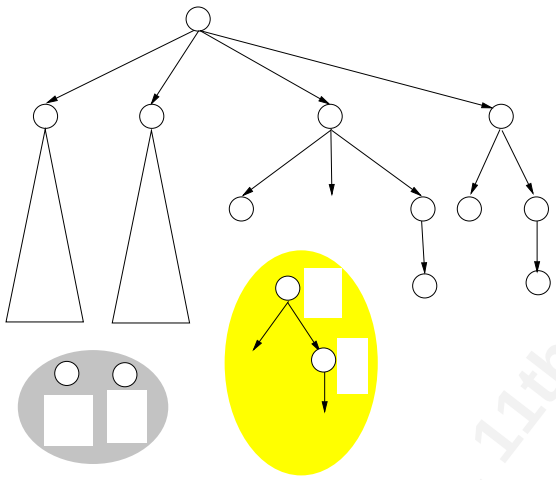

Figure 4: Result of **split**(*Brazil(* $!?^*$ *USA* $!?^*$*)*, $\lambda(x,y,z)\langle x,y,z\rangle)(T)$

Formally, **split** can be defined as:

$$\begin{aligned}\textbf{split}(tp,f)(T) = \{f(x,y,[t_1,t_2,...,t_n]) \mid\ & \\ (x \circ_\alpha y \circ_{\alpha_1} t_1 \circ_{\alpha_2} ... \circ_{\alpha_n} t_n = T)& \\ \wedge(x \circ_{\alpha_i} q = x, \text{for } i = 1,2..,n \text{ and any tree } q)& \\ \wedge(y \circ_{\alpha_1} nil \circ_{\alpha_2} nil \circ_{\alpha_3} ... \circ_{\alpha_n} nil \in \mathcal{L}(tp))& \\ \wedge(y \text{ contains all } \alpha_i, 1 \le i \le n)\}&\end{aligned}$$

The **split** operator is unique in that it allows us to break up a tree and put it back together later. Providing an operator to "break" up the tree around a specific pattern allows us to optimize some queries by transforming them into a **split** on a pattern that is inexpensive to find and to apply simpler queries to the pieces (an example is presented in Section 5).

- **Sub_select**$(tp)(T)$: returns the set of subgraphs of T that match pattern tp.

$$\begin{aligned}&\textbf{sub\_select}(tp)(T) = \\ &\quad \textbf{split}(tp, \lambda(a,b,c)\, b \circ_{\alpha_1,...,\alpha_n} [])(T)\end{aligned}$$

  The operation $b \circ_{\alpha_1,\alpha_2...\alpha_n} []$ is a shorthand for concatenating $\alpha_1, \alpha_2, ..., \alpha_n$ to NULL.

- **All_anc**$(tp,f)(T)$: can be expressed using **split** – it returns the result of function f applied to the match and all the ancestors of the match. Formally, **all_anc** is defined as:

$$\begin{aligned}&\textbf{all\_anc}(tp,f)(T) = \textbf{apply}(\lambda(a)\, f(1(a),2(a)))(A) \\ &\quad \text{where } A = \textbf{split}(tp, g(a,b,c))(T) \text{ and} \\ &\quad g(a,b,c) = \lambda(a,b,c)\langle a, (b \circ_{\alpha_1,\alpha_2...\alpha_n} [])\rangle\end{aligned}$$

- **All_desc**$(tp,f)(T)$: can also be defined in terms of **split**. It returns the result of the function f applied to the match and its descendants.

$$\begin{aligned}&\textbf{all\_desc}(tp,f)(T) = \textbf{apply}(\lambda(a)\, f(1(a),2(a)))(A) \\ &\quad \text{where } A = \textbf{split}(tp, g(a,b,c))(T) \\ &\quad \text{and } g(a,b,c) = \lambda(a,b,c)\langle b,c\rangle\end{aligned}$$

Why Split? **Split** is a very powerful operator. It can be used to construct all of the other matching operators. Additionally, **split** may be viewed as an order-preserving analog for **fold** [19] that is based on pattern matching. **Split** also allows us to preserve the context of the match. One might ask, though, what is the use of retaining the context. While there are cases in which users might want to see this in the result, we feel that the primary motivation is in optimization.

While a complete treatment of this topic is beyond the scope of this paper, the basic idea is simple. In relational optimization, a **select** with a complex conjunctive predicate might be rewritten as an intersection of two or more selects, each containing a different conjunct (or set of conjuncts) from the original. In this way a complex predicate is broken into simpler pieces some of which might be very cheap to process (e.g., by using an index).

We mirror this technique with **sub_select** in trees, by using the **split** operator to produce simpler **sub_select**s. Thus the query **sub_select**$(d(e(h\ i)j))(T)$ can be rewritten as:

$$\begin{aligned}&\textbf{apply}(\textbf{sub\_select}(\top d(e(h\ i)j))) \\ &\quad (\textbf{split}(d, \lambda(x,y,z)\, y \circ_{\alpha_1,\alpha_2} z)(T))\end{aligned}$$

Assume that we can use an index to efficiently locate all nodes in T that match d. The intuition is

that the **split** operator uses the index on d to pick all the subtrees of T that are rooted at d. This drastically narrows the search space for the subsequent **sub_select** since the pattern can occur only at the root of each tree in the result set of **split**. So, **apply**ing **sub_select** to the result of **split** produces the answer to our original query more efficiently.

As described above, AQUA has a large number of query operators which have been chosen for their usefulness and succinctness; however they can all be expressed in terms of a smaller subset of primitive operators. The primitive tree query operators are **apply** and **split** [30].

5 More Tree Queries

In order to demonstrate the use and power of the algebra, we consider a slightly more complex example. Consider a parse tree T of a database query. Each node stands for an algebra operator and the children of a node are the inputs to the operator (Figure 5). We can specify compile time optimizations on T using our tree operators. This suggests that our tree query language would be useful in constructing a rewrite based optimizer.

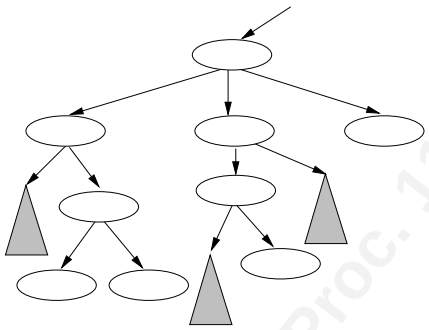

Figure 5: A parse tree

The parse tree is of type *Tree[Parse-tree-node]* where a *Parse-tree-node* supports the method *OpName* which returns a string that is the name of the query operator.

Suppose we wish to "optimize" the query represented by this parse tree T and the rule that we wish to use is:

$$\mathbf{select}(R, \mathbf{and}(p_1, p_2)) \equiv \mathbf{select}(\mathbf{select}(R, p_1), p_2)$$

The first step is to find all the places where the subtree $\mathbf{select}(R, \mathbf{and}(p_1, p_2))$ occurs with its context. The second step is an update operation where we need to replace this portion of the tree with $\mathbf{select}(\mathbf{select}(R, p_1), p_2)$ (which is why we need the context).

The first query is $\mathbf{split}(select(!?\ and), f)(T)$. Here we use "*select*" as a shorthand for the predicate $\lambda(pn)$ *pn.OpName = "select"* and "*and*" for the predicate $\lambda(pn)$ *pn.OpName = "and"*. This returns a set of matches where the pattern occurs. In order to modify the parse tree, we need to specify an appropriate function f which will create a new parse tree

$$\begin{aligned} f(x, y, z) \quad = \quad & x \circ_\alpha (\mathbf{tree}(\ A(A(B\ D)E)) \text{ where} \\ & y \stackrel{\circ}{=} A(B\ C(D\ E))) \circ_{\alpha_1, \alpha_2, \alpha_3} z \end{aligned}$$

The update function f is a three-place function that operates on the three pieces that **split** produces. The first two pieces are subtrees, while the third component (z) is a list of subtrees. The operation $y \circ_{\alpha_1, \alpha_2, \alpha_3} z$ is a shorthand for writing $y \circ_{\alpha_1} z_1 \circ_{\alpha_2} z_2 \circ_{\alpha_3} z_3$ where z_i refers to the ith element of the list z. Operator **tree** creates a new tree and $\stackrel{\circ}{=}$ is a shorthand for specifying that y is of the form $A(B\ C(D\ E))$ where $A...E$ are bound to nodes of the tree.

Now let us consider querying over trees with variable-arity nodes. An example of such a tree is a parse tree for a C program. One simple optimization over such a parse tree might be to find out if any **printf** (which is a variable-arity function) refers to a particular data structure *LargeData* at least twice so that we could cache the data structure appropriately. In this query we do not know a priori the arity of the **printf** node. So we need to use the notation for variable arity trees to express the query.

$$\mathbf{sub\_select}(printf(?^*\ LargeData\ ?^*\ LargeData\ ?^*))(T)$$

The query returns all occurrences of **printf** that refer to at least two occurrences of *LargeData*, along with all of its other parameters.

6 List Operators

Ignoring typing issues for the moment, we can view a list as a tree in which each tree-node has at most one child. Let's call such trees *list-like trees*. As a result, list operators translate to the corresponding tree operators applied to list-like trees.

We divide list query operators into two categories, similar to the categories for the tree query operators. Consider the first category of operators – **select** and **apply**. **Select** selects all the nodes that satisfy the predicate p from the tree T_L, where T_L is a list-like tree. This operation would return a singleton set with a tree containing nodes that satisfy p. Each of the nodes of the resulting tree will have at most one child. The result is the same as in the case where the type

of T_L is a list and the output type is also a list. We can easily see that list **apply** is the same as the tree operator **apply**.

The next category of operators - **sub_select**, **all_anc**, **all_desc**, and **split** take as input a list predicate. These operators are similar to the tree operators assuming we restrict their input to list-like trees. However, there are some differences in notation between list and tree predicates. So, before we demonstrate the mapping between list operators and their corresponding tree operators, we need to address the issue of different notations used for specifying patterns in lists and trees.

The list pattern $[abc]$ corresponds to the tree pattern $a(b(c))$. This difference in notation is due to the fact that trees normally have multiple children and the use of ( ) allows us to establish the hierarchy between the nodes of the tree. Now we show the mapping between the tree-predicate language and the list predicate language. The list and tree disjunction operators (|) are identical. The concatenation ($\circ_\alpha$) and the Kleene closure ($*_\alpha$) operators in trees are parameterized by *concatenation points*. However, list-like trees can have a concatenation point only at the leaf since the nodes can have at most one child. So, $[abc] \circ [cba]$ in list notation translates to $a(b(c(\alpha))) \circ_\alpha c(b(a))$ in tree notation, where α is a concatenation point symbol. Kleene closure over lists is slightly different since we can "pump" any node (or sublist) of a tree. As a result we have to view the list $[d[\![ac]\!]^*b]$ as a concatenation of three sublists, i.e. $[d] \circ [ac]^* \circ [b]$. Now translating this into tree notation, we can write the pattern as $d(\alpha_1) \circ_{\alpha_1} a(c(\alpha_2))^{*_{\alpha_2}} \circ_{\alpha_2} b$.

Using the above translation from list predicates to tree predicates, we can map all list operators onto tree operators. **Sub_select**$(lp)(L)$ returns the set of sublists of L that match pattern lp. Other operators like **all_anc**$(lp, f)(L)$, **all_desc**$(lp, f)(L)$, and **split**$(lp, f)(L)$ are also similar to their corresponding tree operators except for their input and return types (which are lists). Formal definitions of these operators are given in [30].

To illustrate the behavior of these operators let us consider a very simple example of a music database. The database consists of a large number of songs, where each song is represented as a list consisting of nodes that represent a note. Each note has a few properties like pitch (e.g., A, B, C, etc.) and duration. Now suppose we wish to find a simple melody (e.g. $[A??F]$) in a particular song L where A stands for $\lambda(n)$ *n.pitch = "A"*, F is $\lambda(pn)$ *n.pitch = "F"*, and ? is a note with any pitch. The corresponding query would be **sub_select**$([A??F])(L)$. This would return a set containing all such phrases in L. Now suppose we need to find the melody and the context in which it occurs - if we want the notes preceding the melody our query would be

$$\textbf{all\_anc}([A??F], \lambda(x,y)\langle x,y\rangle)(L)$$

This returns a set of tuples, one for each match found. The first field of the tuple returns the sublist from the beginning of the song up to the starting position of the melody, the second field returns the melody. **All_desc** and **split** would work similarly.

The primitive query operator for lists is **split** and all other query operators can be expressed in terms of it [30].

7 Related Work

In this section we discuss related work in the area of algebra and data model support for lists and trees, followed by a discussion of work in the area of predicate languages for these types.

7.1 Ordered Types in Other Data Models

Much of the previous work with ordering deals with order as in sequences or arrays. There are some papers that address other types like trees [2,3] and graphs [14]. However, most of the previous work deals with ordered types in a piecemeal fashion - our work is unique in that it takes a holistic approach to bulk types, including a richer notion of pattern matching.

The operations described in [2] are intended to be applicable to any bulk type, not just to lists and trees. As a result, operations like selecting a sublist are not provided.

MDM [24] presents a query algebra to support lists in an object-oriented data model. The salient feature of their algebra is the extension of the predicate language to allow position-dependent queries. However, their predicate language for lists is not as versatile as regular expressions since predicates are applied to each element of the list rather than the list as a whole.

The NST algebra [15] is specifically designed for structured office documents and is an extension of relational algebra. It tries to maintain the order of the input lists whenever possible, with a higher preference for the order of the first input list. As a result, most of the operators are not commutative. A tree-like structure in a document (paragraphs under sections) is handled by treating it as a nested sequence of sequences. The predicate language is limited in its power - for example, it cannot be used to extract sublists.

Rs-operations [11] are sequence operations that are based on pattern matching. Ginsburg and Wang define a set of powerful operations based on regular expressions that act as a kind of template for the operation, i.e., the regular expressions define what the operation should do "by example". However, they do not mention how these operations can be extended to trees or specify how these operations fit into a query optimization scheme.

The EXTRA/EXCESS system [34] contains an array type constructor and various operations on arrays. The elements of an array are accessed using their array indices. However, the system provides no support for lists (as opposed to arrays), trees or pattern-matching predicates.

Several authors [12,16] have addressed the issue of handling tree-structured data, especially in the form of hierarchical tree structures (specifically in text-dominated databases). The database is described by a schema expressed as a grammar and the operators manipulate data expressed in the form of "production rules". Each schema, represented by a tree, defines an "isa" relationship between the parent node and the child nodes. These papers, however, do not address the issue of more general trees.

An algebra for queries on sequences is presented in [27], but this algebra does not address pattern-matching (its predicates are applied to one node at a time). All of the operators result in a single sequence.

Recent work [35,36] on approximate tree matching discusses tree algebras targeted towards problems in vision and molecular biology. These papers propose various distance metrics for trees. These metrics are useful in answering queries such as "*give me all the subtrees of T which almost satisfy pattern P*". Such metrics are easily accommodated in our formalisms. The authors also present various optimizations for distance-based queries with associated indices that can be expressed by query transformations in the AQUA algebra. Approximate subsequence matching for lists is addressed in [9], but only fixed-length patterns are allowed (no regular expressions).

Many commercial systems (e.g. ObjectStore [23]) claim to support queries on lists, but these are simply the same queries that can be asked of sets; the ordering properties of lists are not taken into account in either the predicate language or in the result of the query.

7.2 Predicates for List and Tree Queries

A model and language for sequences of events is presented in [10]. Their pattern language is equivalent to regular expressions, but the result of any query is a single sequence, restricting the set of allowable queries. Query optimizations such as ours are not addressed.

A powerful pattern language for strings (but not trees) is described in [13]. This language is context-sensitive and thus more powerful than ours, but no attention is given to fitting it into an object-oriented context or into a query optimization scheme. It extends the relational algebra select by allowing filters based on multitape automata.

Our work on list and tree predicate languages is based on earlier work on regular expressions for lists and trees [4,6,21,25,33].

The predicate language for lists is based on regular expressions for strings. Our notation for trees is an amalgamation and extension of ideas first presented in [6,33]. We have augmented the tree language to handle trees with variable arity, to distinguish specific nodes like the leaves, etc. We have also integrated the behavior of these predicate languages with our operator domain, to allow users to specify sophisticated queries over lists and trees.

Another potential candidate for the predicate specification language was graph grammars [7,8] as they allow us to express "graph rewrites" in a succinct manner. However, the result of a graph grammar derivation is dependent on the order in which the productions are applied. The "order" dependence makes it very hard, if not impossible, to optimize queries by splitting or decomposing the pattern into "simpler" pieces, since the results may vary depending on the order in which the results of the simpler queries are combined. Another problem with graph grammars is the complexity of the notation. A predicate language based on regular expressions, however, is tractable and provides us with sufficient querying capabilities.

8 Summary and Work in Progress

We have described the list and tree algebras for the AQUA object-oriented query algebra. The operators in the algebra preserve the ordering of nodes in the original list or tree. The primitive query operator for lists is **split** and the primitive query operators for trees are **apply** and **split**. Our use of patterns based on extended regular expressions allows us to write queries that are sensitive to the order of the elements in the list or tree. Regular expressions are familiar, powerful, and well suited to a number of application domains. We briefly demonstrated how algebraic decomposition techniques can be used to optimize queries on lists and trees. A companion paper [31] treats the problem of optimization in more detail. There we present opti-

mization rules and access methods for ordered data types.

As part of our research on AQUA, we have developed a mapping for the ODMG set and bag algebra [5] to the AQUA set and multiset algebra. The array type in the ODMG specification is similar to our notion of list, and we believe that we will have little difficulty simulating the ODMG arrays with AQUA lists. Our view of predicates, however, is significantly more powerful.

We are currently developing a cost model and incorporating the list and tree algebras into the EPOQ extensible query optimizer [22] being developed at Brown University.

Acknowledgements

We would like to thank Catriel Beeri, Mitch Cherniack, Darryn Lavery, Gail Mitchell, Arnold Rosenberg, Dennis Shasha, Hagit Shatkay, Sairam Subramanian, Bennet Vance, and Jason Wang for useful discussions.

References

[1] A. V. Aho, M. Ganapathi, and S. W. K. Tjiang, "Code Generation using Tree Matching and Dynamic Programming," *ACM Trans. on Programming Languages and Systems* 11 (1989), 491–516.

[2] C. Beeri and Y. Kornatzky, "Algebraic Optimization of Object-Oriented Query Languages," *Proc. ICDT* (1990), 72–83.

[3] C. Beeri and P. Ta-Shma, "Bulk Data Types, A Theoretical Approach," *Proc. DBPL* (1993), 80–96.

[4] J. A. Brzozowski, "Derivatives of regular expressions," *J. ACM* 11 (1964), 481–494.

[5] R. G. G. Cattell, ed., *The Object Database Standard: ODMG-93, Release 1.1*, Morgan Kaufmann Publishers, San Francisco, 1994.

[6] J. Doner, "Tree Acceptors and Some of their Applications," *J. of Computer and System Sciences* 4 (1970), 406–451.

[7] H. Ehrig, H. -J. Kreowski, and G. Rozenberg, eds., *Graph Grammars and Their Applications to Computer Science* #532, Springer-Verlag, 1991.

[8] J. Engelfriet and G. Rozenberg, "Graph Grammars based on node rewriting: An Introduction to NLC Graph Grammars ," *Proc. 4th Intl. on Graph Grammars and Their Applications to Computer Science* 532 (1991), 12–23.

[9] C. Faloutsos, M. Ranganathan, and Y. Manolopoulos, "Fast Subsequence Matching in Time-Series Databases," *Proc. SIGMOD* (1994), 419–429.

[10] N. H. Gehani, H. V. Jagadish, and O. Shumueli, "Composite Event Specification in Active Databases," *Proc. VLDB* (1992), 327–338.

[11] S. Ginsburg and X. Wang, "Pattern Matching by Rs-Operations: Towards a Unified Approach to Querying Sequenced Data," *Proc. PODS* (1992), 293–300.

[12] G. H. Gonnet and F. W. Tompa, "Mind Your Grammar: a New Approach to Modelling Text," *Proc. VLDB* (1987), 339–346.

[13] G. Grahne, M. Nykänen, and E. Ukkonen, "Reasoning about Strings in Datbases," *Proc. PODS* (1994), 303–312.

[14] R. H. Güting, "GraphDB: A Data Model and Query Language for Graphs in Databases," *Proc. VLDB* (1994).

[15] R. H. Güting, R. Zicari, and D. M. Choy, "An Algebra for Structured Office Documents," *ACM Trans. on Office Info. Systems* 7 (1989), 123–157.

[16] M. Gyssens, J. Paredaens, and D. V. Gucht, "A Grammar-Based Approach Towards Unifying Hierarchical Data Models," *Proc. SIGMOD* (1989), 263–272.

[17] C. M. Hoffmann and M. J. O'Donnell, "Pattern Matching in Trees," *J. ACM* 29 (1982), 68–95.

[18] R. Karp, R. Miller, and A. Rosenberg, "Rapid Identification of Repeated Patterns in Strings, Trees, and Arrays," *Proc. STOC* (1972), 125–136.

[19] T. W. Leung, G. Mitchell, B. Subramanian, B. Vance, S. L. Vandenberg, and S. B. Zdonik, "The AQUA Data Model and Algebra," *Proc. DBPL* (1993), 157–175.

[20] D. Maier and B. Vance, "A Call to Order," *Proc. PODS* (1993), 1–16.

[21] R. McNaughton and H. Yamada, "Regular expressions and state graphs for automata," *IEEE Trans. on Electronic Computers* 9 (1960), 39–47.

[22] G. Mitchell, U. Dayal, and S. B. Zdonik, "Control of an Extensible Query Optimizer: A Planning-Based Approach ," *Proc. VLDB* (1993), 517–528.

[23] J. Orenstein, S. Haradhvala, B. Margulies, and D. Sakahara, "Query Processing in the ObjectStore Database System," *Proc. SIGMOD* (1992), 403–412.

[24] J. Richardson, "Supporting Lists in a Data Model (A Timely Approach)," *Proc. VLDB* (1992).

[25] A. Salomaa, "Two complete axiom systems for the algebra of regular events," *J. ACM* 13 (1966), 158–169.

[26] H. Samet, "Distance Transform for Images represented by Quadtrees," *IEEE Trans. on Pattern Analysis and Machine Intelligence* 4 (1982), 298–303.

[27] P. Seshadri, M. Livny, and R. Ramakrishnan, "Sequence Query Processing," *Proc. SIGMOD* (1994), 430–441.

[28] B. A. Shapiro and K. Zhang, "Comparing Multiple RNA Secondary Structures using Tree Comparisions," *Comput. Appl. Biosci.* 6 (1990), 309–318.

[29] G. M. Shaw and S. B. Zdonik, "Object-Oriented Queries: Equivalence and Optimization," *Proc. 1st Intl. Conf. on Deductive and Object-Oriented Databases* (1989), 264–278.

[30] B. Subramanian, T. W. Leung, S. L. Vandenberg, and S. B. Zdonik, "The AQUA Approach to Querying Lists and Trees in Object-Oriented Databases," Brown Univ., Dept. of CS, Tech. Report, Providence, RI 02912-1910, 1994.

[31] B. Subramanian, T. W. Leung, S. L. Vandenberg, and S. B. Zdonik, "Optimization of List and Tree Queries in AQUA," Brown Univ., Dept. of CS, Tech. Report, Providence, RI 02912-1910, 1994.

[32] B. Subramanian, S. B. Zdonik, T. W. Leung, and S. L. Vandenberg, "Ordered Types in the AQUA Data Model," *Proc. DBPL* (1993), 115–135.

[33] J. W. Thatcher and J. B. Wright, "Generalized Finite Automata Theory with an Application to a Decision Problem of Second-Order Logic," *Mathematical Systems Theory* 2 (1968), 57–81.

[34] S. L. Vandenberg and D. J. DeWitt, "Algebraic Support for Complex Objects with Arrays, Identity, and Inheritance," *Proc. SIGMOD* (1991), 158–167.

[35] T-L. Wang and D. Shasha, "Query Processing for Distance Metrics," *Proc. VLDB* (1990), 602–613.

[36] K. Zhang, D. Shasha, and J. T. L. Wang, "Approximate Tree Matching in the Presence of Variable Length Don't Cares," *J. of Algorithms* 15 (1993).