Querying Faceted Databases

Columbia University Technical Report CUCS-013-03

Kenneth Ross*
Columbia University
kar@cs.columbia.edu

Angel Janevski
Columbia University
aj311@cs.columbia.edu

May 29, 2003

Abstract

Faceted classification allows one to model applications with complex classification hierarchies using orthogonal dimensions. Recent work has examined the use of faceted classification for browsing and search. In this paper, we go further by developing a general query language, called the entity algebra, for hierarchically classified data. The entity algebra is compositional, with query inputs and outputs being sets of entities. Our language has linear data complexity in terms of space and quadratic data complexity in terms of time. We compare the entity algebra with the relational algebra in terms of expressiveness. We also describe an implementation of the language in the context of two application domains, one for an archeological database, and another for a human anatomy database.

*This research was supported by NSF grant IIS-0121239.

1 Introduction

A number of application domains require the modeling of complex entities within classification hierarchies. For many of these domains, the hierarchy is where the main complexity of the domain is concentrated, with other features of the domain, such as relationships between entities, being relatively simple. We aim to develop a data model and a query language appropriate for such domains.

A *monolithic* concept hierarchy is one in which a single large classification tree is used to represent the application domain. Monolithic hierarchies have been criticised for "rigid hierarchical and excessively enumerative subdivision that resulted in the assignment of fixed 'pigeonholes' for subjects that happened to be known or were foreseen when a system was designed but often left no room for future developments and made no provision for the expression of complex relationships and their subsequent retrieval." [16]

A *faceted* classification, on the other hand, "does not assign fixed slots to subjects in sequence, but uses clearly defined, mutually exclusive, and collectively exhaustive aspects, properties, or characteristics of a class or specific subject. Such aspects, properties, or characteristics are called facets of a class or subject, a term introduced into classification theory and given this new meaning by the Indian librarian and classificationist S.R. Ranganathan and first used in his Colon Classification in the early 1930s." [16]

Computers can make faceted classifications work for search [4, 5]. Once a domain has been classified into a number of orthogonal facets, users can select values for one of more facets independently. As the search progresses, the candidate set of answers shrinks. The computer can give feedback to the user on the current size of the candidate answer set, and can update the search so that categories with no answer candidates in them are not displayed. The user is relieved of knowing the exact classification system used, and can find an object by describing its properties. Systems implementing document search for such data models include Flamenco [3] and FacetMap [2]. A user study of Flamenco is presented in [9].

Our aim is to go beyond a simple search facility for faceted hierarchies, and to provide a *query language* for the formulation of more sophisticated queries.

Relational query languages do not provide built-in facilities for manipulating hierarchies. Hierarchies must be simulated. For example, if C_1 is a subclass of C_2, one could store C_1 and C_2 as separate relations R_1 and R_2, but then one needs to keep extra information somewhere (perhaps in a view) to indicate that members of C_2 are also members of C_1. Alternatively, R_1 could store all information about members of C_1 and C_2 defined for class C_1, and R_2 could store extra attributes for entities also in C_2. But then queries asking for members of C_2 need to perform a join. The complexity increases as the depth of the hierarchy increases. Alternative approaches are also possible, but are cumbersome in other ways. In a sense, the relational model uses one construct, i.e., the relation, to represent both relationships of entities to one another, as well as the structure of the entities themselves. In domains where the entity structure is the dominant source of complexity, it is natural to make a different design choice, namely to make the "set of entities" the basic data structure. Related formalisms that also focus on sets of entities are described in Section 2.10.

Our Approach

We start with faceted classification as our basis. A domain expert provides the schema, i.e., a collection of orthogonal classifications of the application domain into moderately-sized hierarchies. Our fundamental notion is the "entity set," a collection of (possibly heterogeneous) entities from various classes in the hierarchy.

A query in our "entity algebra" takes entity-sets as input, and produces an entity-set as output. We thus achieve *compositionality*, meaning that the inputs to a query and the output from a query are of the same type, so that complex queries can be build by composing simpler pieces. Since entities of different classes may coexist in such an entity set, the system must determine, from a query expression and from the schema (but not from the data; see Section 2.10), which attributes are available in all entities in the result of a query expression.

We are aiming for a language that, while allowing most queries typical of our target domain, possesses *low data complexity*. A benefit of our approach is that we guarantee linear space complexity and quadratic time complexity for all expressible queries. In contrast, the relational model admits queries that can take polynomial time and space, where the exponent of the polynomial can be proportional to the number of operators in the query.

The capacity of our system to write queries whose answers represent general relationships is limited. This is a deliberate choice. Our primary goal is to make the data model and query language conceptually simple and understandable to users. Being able to represent complex relationships as well as complex entity hierarchies would create a much higher conceptual burden on users, as well as a higher data complexity.

The system informs the user of all attributes that are available for querying. This can require some calculation in a faceted hierarchy, because (a) attributes are inherited from multiple sources, and (b) constraints may imply membership in a more specific class whose attributes then become available. From the user's point of view, this process is transparent: the user is presented with the set of available attributes for each query or subquery.

We compare the expressive power of the entity algebra with the relational algebra. In general, the expressiveness of the two algebras is incomparable. If we focus on "flat" schemas and relational queries that return just entity-IDs, we can quantify exactly what kinds of relational queries we are forgoing in order to get our complexity results. The answer (projections, and joins with cyclic hypergraphs) is reassuring, since such constructs are typically not crucial for queries on complex hierarchies.

Our design has been implemented in two prototype systems. One system supports an archeological database of finds that are organized into a variety of categories. A second system supports a database of human anatomy, that is classified into hierarchies in various ways. Both systems share a common infrastructure corresponding to the model described here. They differ in the definition of the hierarchies (i.e., the schema) and in the actual data stored. Additional domains could easily be incorporated given a schema and the corresponding data.

In Section 2, we describe our framework, introduce the entity algebra, and assess its complexity and expressiveness. In Section 3 we describe an implementation of our framework. We conclude in Section 4.

2 Framework

2.1 Domain Model

The units of operation for our query language are *sets of entities*. Each query operates on one or more sets of entities and always returns a set of entities. In the archaeology domain, for example, all excavation finds are entities in the database. Each find has many attributes and one of the attributes is the entity type, which can be *object*, i.e., an artifact, or *context*, i.e., a characteristic region of the excavation site.

Entity sets that have explicitly stored entities in them are called *classes*. A schema defines a finite set of classes. Classes have *attributes* associated with them. An attribute has a name and a data type. Each entity in a class must have a *value* of the appropriate type for each attribute. An entity may belong to multiple classes. For example, an object can belong to the class "Pots" and the class "My-Favorite-Objects" simultaneously. Such an object provides values for all attributes of all classes it belongs to. Note that we do not require the creation of a subclass "My-Favorite-Pots" to store favorite objects that happen to be pots. This modeling style is what makes faceted classification different from traditional object-oriented models of hierarchies. If we did require such classes, there would be too many of them, as each class could be intersected with an arbitrary set of other classes. Figure 1 shows a class hierarchy based on our archeology application. Attributes are shown in square brackets.

Classes may also have *constraints* attached to them. For example, the class "Big-Pots" might have a constraint on the *capacity* attribute of the pots which can belong to that class. Note that

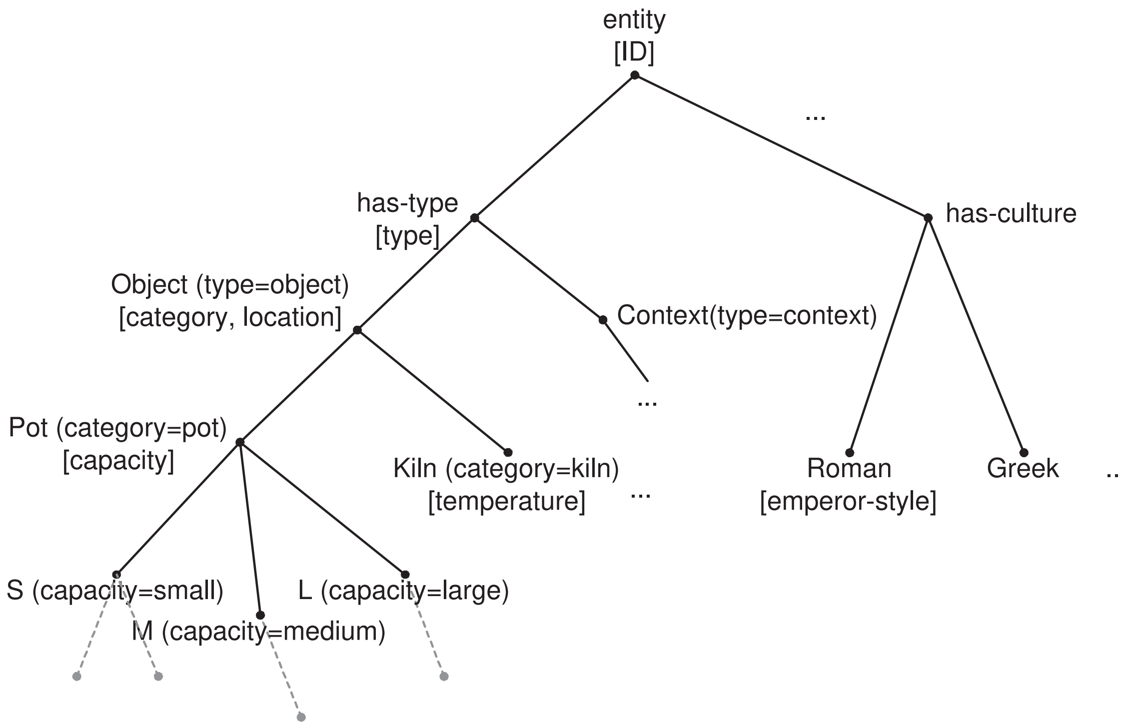

Figure 1: A Partial Archeology Schema

these are integrity constraints in the traditional sense, and not view definitions. There may be large pots in the database that, for some reason, do not belong to the "Big-Pots" class. Additional examples of contraints appear in round brackets in Figure 1. The constraints imply that Pots and Kilns are disjoint, while an entity may have both Greek and Roman culture.

Classes are organized into a *hierarchy*. We write $C_1 < C_2$ to mean that C_1 is a subclass of C_2. This is graphically represented by drawing a line with C_2 above C_1. The transitive closure $\preceq$ of the subclass relationship is a partial order with a single maximal element E, which denotes the class of all entities. If $C_1 \preceq C_2$ then all attributes of C_2 are also attributes of C_1. Similarly, all constraints on entities in C_2 also apply to entities in C_1. The maximal class E has a single attribute called "ID". All values of the ID attribute are unique. If an entity in class "Pots" has ID 123, and an entity in class "Roman objects" has ID 123, then they refer to the same real-world artifact, namely a Roman pot.

Since different classes may use the same name for attributes, we disambiguate attributes by providing as a prefix the name of the class in the hierarchy from which a class inherited the attribute. So, if both C_1 and C_2 have an attribute *style*, and $C_3 \preceq C_1$ and $C_3 \preceq C_2$, then C_3 has two attributes C_1::*style* and C_2::*style*. In principle, C_3 could also define its own version C_3::*style*. There is no overriding of attributes. Also, an attribute that is inherited on multiple paths is not replicated.

While we have not explicitly represented relationships, we note that general relationships can be simulated by thinking of tuples as entities. This is the dual of the relational model, in which entities are modeled as relations.

2.2 Constraints

We assume that a constraint language CL is given. A typical constraint language may allow equalities and inequalities over integers, reals, and strings. Formulas in CL may use as free variables expressions of the form $S.A$ where S is an entity set, and A is an attribute of S. The domain of $S.A$ corresponds to the type of A in S. We assume that CL

includes logical conjunction "$\wedge$" and disjunction "$\vee$".

As mentioned above, integrity constraints from CL may be placed on classes. We use the same constraint language to define operators such as selection; see Section 2.3.

We will say that a constraint language CL is *decidable* if the satisfiability of sentences in CL is decidable. Constraint language implementations may benefit from the use of a constraint solving system [11].

2.3 Operators and Queries

A query is formed by applying operators to entity sets to form new entity sets. The user starts with a collection of entity sets defined by the classes in the schema. During a query session, the user can refer to a previously defined entity set as a subexpression. The language defined by the operators below is called the *entity algebra*.

If C is a class, then the query expression C denotes all entities that are members of a class C' where $C' \preceq C$. We allow the following operators where E and E' are entity sets, θ is a constraint with free variables ranging over attributes of E, and θ' is a constraint from CL with free variables ranging over attributes of E and E'.

- $\sigma_\theta(E)$ returns all entities from E that satisfy the condition θ.
- $E \rtimes_{\theta'} E'$ returns all entities e from E for which there is some entity e' in E' such that (e, e') satisfies θ'.
- $E \cup E'$ returns all entities that are in either E or E'; duplicates are omitted.
- $E \cap E'$ returns all entities that are in both E and E'.
- $E - E'$ returns all entities that are in E but not in E'.

This definition of our operators is not quite complete. If E is a class, then it is clear which attributes are available for the conditions θ and θ' above. However, if E is itself an expression, we have not yet explained how to determine the attributes available from E. For example, we need to know how to determine which attributes are available from the expression $C_1 \cup C_2$ which admits entities belonging to two different classes. This issue is addressed in Section 2.4.

We remark that having entities from different classes poses no structural problem in our model. A set of entities can contain entities of many types, and each entity can have its own set of defined attributes. When one wants to display the entities in the result of a query, each entity can be displayed in a way that is appropriate to its type(s). For our application domains, this kind of result structure is much more convenient than a relation. In order to show all attributes of all result entities, a relation would need to have an attribute for each possible attribute of any entity in the result set, with most attribute values being null.

2.4 Expression Types

The determination of which attributes are available from query expressions is not trivial. We can state a semantic correctness criterion informally as follows: An attribute A is *correct* for a query expression E if and only if, for every possible database instance, every entity in the result of E possesses attribute A. This criterion needs to be slightly refined to allow for the possibility that a query expression is not well-formed. As a result, we formulate a recursive formal definition.

Definition 2.1 *If an entity set E is a class, then the correct set of attributes for E is the set of attributes defined for that class in the schema.*

Let F be an operator on entity-sets $E_1, \ldots, E_n$, and suppose that the correct set of attributes for $E_1, \ldots, E_n$ has been determined. Suppose that F is well-formed, i.e., that conditions in F refer only to attributes that are correct for $E_1, \ldots, E_n$. Then an attribute A is correct *for the query expression $F(E_1, \ldots, E_n)$ if and only if, for every possible database instance, every entity in the result of the query possesses attribute A.*

Given this semantic correctness criterion, we wish to determine syntactic methods for obtaining the correct set of attributes. We emphasize that it is up to the system, and not the user, to determine the correct set of attributes. As the user formulates each subquery, the system gives the user feedback about which attributes are available. We illustrate some of the subtleties of determining the correct set of attributes in the examples below.

Example 2.1 *If E is an expression such as $C-C$ or $\sigma_{false}(C)$ that is guaranteed to be empty, then all attributes are correct for E. Thus, in order to determine the correct attributes for $\sigma_\theta(C)$ we need to know whether θ is satisfiable. Similarly, to determine the correct attributes for $C - \sigma_\theta(C)$ we need to know whether θ is a tautology. If class C has an integrity constraint ϕ, then the above statements apply to $\theta \wedge \phi$ rather than just θ.*

Example 2.2 *If C_1 and C_2 are classes, then $C_1 \cap C_2$ should include all attributes from both C_1 and C_2. On the other hand, $C_1 \cup C_2$ should include only attributes that are common to both C_1 and C_2, i.e., attributes that are inherited from a common ancestor in the hierarchy. Note that there may be more than one "least" ancestor, because the hierarchy is not necessarily a tree. A common ancestor is guaranteed by the presence of the class E.*

Example 2.3 *In this example we show that correct attribute sets cannot be computed for each subexpression separately, and unioned or intersected incrementally.*

Consider three classes S, M, and L representing "small," "medium" and "large" pots, respectively. Suppose that each such class is a subclass of the class Pot, which has an attribute "capacity". Each subclass has a constraint on capacity. For example, class S would have the constraint capacity=small. For the sake of argument, suppose that each of S, M, and L has its own additional attributes.

Consider the expression $(S \cup M) \cap (M \cup L)$. The correct attributes of $(S \cup M)$ would be the attributes of class Pot. The same reasoning applies to $(M \cup L)$. So it would seem that the attributes of Pot are precisely the correct attributes of the whole expression. This reasoning is fallacious. To see why, let us rewrite the original expression as the equivalent expression $(S \cap M) \cup (S \cap L) \cup (M \cap M) \cup (M \cap L)$. The constraints on each subclass mean that the only nonempty term in the union is $(M \cap M) = M$. Thus, the correct set of attributes are those of M, which is a strict superset of those belonging to class Pot.

Example 2.3 shows that we cannot compute the attribute sets via a function g with $g(X \cap Y) = g(X) \cup g(Y)$.

We now describe our initial typing algorithm for queries involving selections, unions and intersections.

Algorithm 2.1 *We are given an entity algebra query Q, using just selections, intersections and unions. Compute an equivalent query T by (a) pushing the selection conditions down to classes, using the fact that selections distribute over unions and intersections, and then (b) rewriting the result in disjunctive normal form so that T is a union of conjunctive queries. Replace instances of $\sigma_\theta(\sigma_\phi(E))$ by $\sigma_{\theta\wedge\phi}(E)$. Suppose that $T = T_1 \cup \ldots \cup T_n$, where each T_i is a conjunctive query.*

For each T_i, do the following. Suppose that $T_i = \sigma_{\theta_1} C_1 \wedge \ldots \wedge \sigma_{\theta_m} C_m$, where each C_j is a class and each θ_j is a (possibly trivial) condition. If the constraints on the respective classes are ϕ_1, …, ϕ_m, then determine whether $\phi_1 \wedge \ldots \wedge \phi_m \wedge \theta_1 \wedge \ldots \theta_m$ is satisfiable. If so, compute the attribute set A_i as the union of all attributes in $C_1, \ldots, C_m$.

Return the intersection of all computed attribute sets A_i. If there were no such sets computed, return the universal set of all attributes.

Lemma 2.1 *Suppose that the constraint language is decidable. Then Algorithm 2.1 terminates, and computes exactly the correct set of attributes for query Q.*

Proof. *Given the decidability of the constraint language, all steps of the algorithm terminate. To show that the algorithm is sound, suppose that attribute A is output by the algorithm. Then attribute A is possessed by some class in each term T_i that is satisfiable. Thus, every entity satisfying Q has attribute A. To show completeness, suppose that some correct attribute A was not output by the algorithm. Then for some satisfiable term T_i, no class in T_i has attribute A. Since T_i is satisfiable, there exists a database instance in which there is* *an entity belonging to all classes in T_i and satisfying the selection conditions of T_i, thus satisfying Q. However, this entity does not possess attribute A, contradicting the assumption that A was correct for Q.*

We can extend the algorithm and the correctness result to queries with semijoins.

Definition 2.2 *Consider a query $E_1 \ltimes_\theta E_2$ where E_1 and E_2 contain just selections, unions and intersections. Using the construction of Algorithm 2.1, we can obtain a query Q_2 equivalent to E_2 in disjunctive normal form. We abstract Q_2 into a logical formula by forming a logical disjunction of terms, one per conjunctive term in Q_2. Each term consists of the conjunction of the θ and ϕ expressions described in the construction. Let us call the complete formula F_2. We can then "abstract" the semijoin, treating it as if it were a selection $\sigma_{\theta'}(E_1)$, where θ' is defined as $\theta \wedge F_2$. In this formula, free variables from E_2 are assumed to be existentially quantified.*

The abstracted semijoin removes the requirement that matching tuples *actually* exist in E_2, and replaces it with the broader criterion of whether matching tuples *could possibly* exist in E_2. The transformation may introduce extra conjunctions, disjunctions, and free variables, but the decidability of satisfiability in the constraint language is not compromised.

Example 2.4 *Let class C_1 have an attribute X, and suppose classes C_2 and C_3 both have attributes Y and Z. Suppose that C_2 has an integrity constraint stating that $Y = Z$. Then*

$$C_1 \ltimes_{X=Y} (C_2 \cap \sigma_{Z<3}(C_3))$$

can be abstracted as $\sigma_\theta(C_1)$, where θ is

$$\exists Y, Z : (X = Y) \wedge (Y = Z) \wedge (Z < 3)$$

which can be simplified to $X < 3$.

Lemma 2.2 *A semijoin query is satisfiable if and only if its abstracted semijoin query is satisfiable. Further, the same assignments of values to the variables of each query lead to satisfiability.*

Proof. *Suppose the semijoin query $E_1 \ltimes_\theta E_2$ is satisfied by tuples e_1 and e_2 in E_1 and E_2 respectively in some database instance. Then e_1 satisfies the abstracted query, with e_2 providing the satisfying values for the existentially quantified variables. Conversely, suppose that the abstracted query is satisfiable with tuple e_1 being a satisfying assignment for the variables of E_1. Then construct a tuple e_2 in E_2 by using a satisfying assignment for the existentially quantified variable to generate values for the corresponding attributes in E_2. The construction ensures that e_2 satisfies the integrity constraints of E_2. Thus, for some database instance, the original semijoin query is satisfiable.*

The extension to Algorithm 2.1 involves first applying the transformation of Definition 2.2 to each semijoin in the query in a bottom-up order. The transformed query contains only unions, intersections and selections, and can be processed through Algorithm 2.1 as before. The correctness argument is a simple extension of Lemma 2.1 using Lemma 2.2, and the fact that semijoins are monotonic in their inputs.

Subtraction seems intrinsically harder, due to its nonmonotonicity. A corresponding abstraction process requires a constraint language CL that is closed under negation and universal quantification. Further, we cannot analyze subexpressions of a query independently, because one subexpression might require the *absence* of a certain tuple for satisfiability, while another might require its *presence*.

For subtraction we use a sound, but not necessarily complete method for determining the attribute set. For a query Q that includes subtraction, we form a query Q' by eliminating all subtractions from Q. Every subexpression of the form $E_1 - E_2$ in Q is replaced simply by E_1 in Q'. We then compute the attributes of Q' as above.

The query complexity of Algorithm 2.1 is at least exponential in the size of the query, since it has to perform a transformation into disjunctive normal form. The complexity of satisfiablility checking in CL also has obvious implications for the complexity of Algorithm 2.1.[1] Nevertheless,

[1]In the event that CL is not decidable, then we are forced to settle for sound but incomplete satisfiability testing in

we expect queries to be short, and Algorithm 2.1 to be useful in practice. In Section 2.5 we show that the language has low data complexity.

Example 2.5 *Consider the schema of Figure 1 and suppose we wish to find all kilns located within a certain distance t of any medium-sized Roman pot. This kind of query cannot be answered by using a conventional search facility; a query language is required. In the entity algebra, we could express this query as*

$$Kiln \ltimes_{\theta} \left(\sigma_{capacity=medium}(Pot \cap Roman)\right)$$

where θ is "$d(Kiln.location, Pot.location) < t$." All attributes of both Pot and Roman are available for use in the selection and semijoin conditions.

2.5 Data Complexity

One of our initial goals was to choose a language with low data complexity. In this section we demonstrate that all entity algebra queries can be answered in linear space complexity (with constant of proportionality 1), and quadratic time complexity.

Lemma 2.3 *Entity algebra queries generate output that is no larger than the total size of the union of the input classes.*

Proof. *By induction, the output must be a subset of the union of all inputs.*

Lemma 2.4 *Union-free entity-algebra queries generate output that is a subset of at least one of the input classes.*

Proof. *By induction; this is a property of all operators other than union.*

Lemma 2.5 *All entity algebra queries can be computed in time at most quadratic in the total size of the input.*

Proof. *Selection can be computed in linear time. Union, intersection and difference can be computed in $O(n \log n)$ time, where n is the total size of the inputs. Semijoins can be computed in $O(n^2)$ time by simply comparing all pairs of tuples. Given that the size of the output of a subexpression is bounded by the size of its inputs (Lemma 2.3), the whole query takes at most quadratic time.*

2.6 Language Extensions

Because one of our initial goals was to obtain low data complexity, we do not consider desirable language extensions that increase the data complexity. Similarly, our model is centered around the notion of always returning a set of entities in response to a query. An extension that broadened the types of results, such as to return pairs of entities, would *weaken* the model. We believe that the uniformity and simplicity of input and outputs makes the conceptualization task easier for the user.

Note that the model allows the representation of relationships. These relationships can be modeled via foreign keys. In the archeology domain, we could have an entity called "discovery" with references to both the artifact discovered and to the person who made the discovery. Our emphasis on entities over relationships means that relationships are represented as entities. In a sense we make the opposite choice from the relational model, which represents entities as relations.

We discuss two language extensions that retain the spirit of the entity algebra. The first is the capacity to define new attributes as *views*. For example, suppose that each member of class object has a recorded (x, y, z) position at which it was discovered, in a local coordinate system. We could define new global position attributes (gx, gy, gz) derived from (x, y, z) and the reference point entity coordinates. (Formally, this feature would entail a generalization of the semijoin operator.) These new attributes would be available for all members of class object, including members of its subclasses. If the view was registered in the database schema, then the set of available attributes for entities in each class would be extended appropriately.

The second extension is a form of aggregation. The idea is to allow a limited form of aggregation that corresponds (in relational terms) to grouping by the entity-ID. Thus we could define, for each person working on the site, the number of discoveries made by that person. The result would be

Algorithm 2.1.

represented as a view attribute on class person. To achieve this functionality, we again extend the semijoin operation to allow an optional aggregate computation over the records of the second entity set matching each entity in the first entity set. Neither of these extensions change the asymptotic space or time complexity of the language. They also preserve the central theme of inputs and outputs being entity sets.

2.7 Expressive Power

The expressive power of the entity algebra is incomparable with relational algebra. Relational algebra is capable of expressing queries that return tuples of entities, which the entity algebra cannot. Its space complexity and time complexity are polynomial, as opposed to the linear space and quadratic time complexity of the entity algebra. On the other hand, relational algebra (without nulls) is not capable of expressing a query analogous to Example 2.3 in which the attributes of class M are available in the result.

Nevertheless, we can compare the expressive power of the two languages in the context of a *flat* hierarchy. Imagine each class as a relation, and consider a query expressed in relational algebra over those flat relations. For comparability, suppose we limit ourselves to relational queries that return a single column of entity-IDs. Under what circumstances can such a query be expressed in our language? The answer to this question will give us a sense of what kinds of relational queries we are giving up in order to obtain our more limited language.

Lemma 2.6 *Let S be a relational schema in which every relation has a column named ID that is known to be a key. Let Q be a relational algebra query that involves only joins, and suppose $R.ID$ is a column of the output of Q, where R is a relation in S. Then $\pi_{R.ID}(Q)$ is expressible in the entity algebra if the join hypergraph [15] for Q is acyclic.*

Proof. *This result uses a result of Yannakakis [17] (see also [15]). The joins can be ordered so that "ears" [15] of the join hypergraph are removed one by one, ending with R. Because of the special form of the projection (one attribute from relation R), no attributes from an inner subexpression are needed in an outer subexpression, and joins can be replaced with semijoins.*

Lemma 2.6 suggests that the entity algebra cannot express cyclic joins. The intuition is given in Example 2.6. Since queries with cyclic hypergraphs are rare, this loss of power does not seem like a major sacrifice.

Example 2.6 *Consider the relational query*

$$\pi_{R.ID}(R \bowtie_{(R.A=S.B)\wedge(R.C>T.D)} (S \bowtie_{S.F=T.G} T)).$$

The join hypergraph is cyclic. There is no way to express this query using only semijoins, because no matter which pair of relations we semijoin first, we need attributes from both in the remainder of the query. If we include two semijoins, (e.g., $S \rtimes T$ and $T \rtimes S$) then we lose the association between the S and T tuples.

Theorem 2.7 *The entity algebra can express any relational query that can be written as a combination, via the set operations union, intersection, and difference, and via local selections, of queries satisfying the conditions of Lemma 2.6.*

Proof. *Local selections can be pushed down to base relations. Each component query can then be expressed via semijoins as shown in Lemma 2.6. The set operations operate on just IDs, and can be simulated by corresponding set operations in the entity algebra.*

Since set operations distribute over joins, the class of queries that can be written as described in Theorem 2.7 is fairly broad. Conspicuously absent from Theorem 2.7 is the projection operator. Example 2.7 shows an example where the entity algebra cannot express a relational query involving projection.

Example 2.7 *Consider the relational query*

$$\pi_{R.ID}(R \bowtie_{(R.C>F)} (\pi_F S - \pi_F T))$$

where attribute F (belonging to S and T) is distinct from ID. The entity algebra does not provide facilities for projection, and difference can only be applied to entity sets including an ID attribute.

Thus we cannot write a subexpression corresponding to $(\pi_F S - \pi_F T)$. Such an expression would not even be an entity set. Further, since $R \rhd\!\!\!< _\theta (S - T)$ is not, in general, equivalent to $(R \rhd\!\!\!< _\theta S) - (R \rhd\!\!\!< _\theta T)$, we cannot write this expression as the difference of expressions that include an ID attribute.

The lack of a projection operator means that all operations apply to entities "as a whole" and not to arbitrary subsets of attributes. This is a reasonable choice in our context, in which entities are the central concept, and manipulations of attributes without reference to their corresponding entities is unlikely to be common.

2.8 Virtual Classes

Consider Example 2.3, and suppose that we wish to insist that a pot must be classified as either small, medium, or large. If we could represent such information, then we should be able to infer that the expression

$$\sigma_{capacity=medium}(Pot)$$

has type M. Without the extra information, there may be a pot with medium capacity in class Pot (and not in its subclasses), meaning that the type of the expression above would be *Pot* rather than M.

The intuitive way to specify this extra information would be to formulate a sentence in the constraint language CL stating that any member of class *Pot* must be in $S \cup M \cup L$. Because such a constraint relates more than one class, it places additional requirements on CL beyond those we have assumed so far. Further, an explicit constraint relating *Pot* with $S \cup M \cup L$ is vulnerable to schema changes. If another category "extra-large pots" was to be added as a subclass of *Pot*, then the constraint on *Pot* would also need to be changed.

Rather than requiring an extended constraint language, we propose a simpler solution to represent the kind of constraint mentioned above. A non-leaf class may be declared as *virtual*, which means that it has no explicit members beyond those of its subclasses. In order to achieve the correct type for a query expression Q, we rewrite Q. A virtual class C mentioned in Q is replaced by the expression $C_1 \cup \ldots \cup C_k$, where the C_i are the subclasses of C. Subclasses that are themselves virtual are recursively rewritten. The resulting query Q' is equivalent to the original query Q on instances in which virtual classes contain no members beyond those of their subclasses. We then type Q' as described in Section 2.4.

Example 2.8 *Consider the query Q given by*

$$\sigma_{capacity=medium}(Pot)$$

on the schema of Example 2.3, but in which we declare class Pot as virtual. We rewrite Q as Q', i.e.,

$$\sigma_{capacity=medium}(S \cup M \cup L).$$

According to Algorithm 2.1, the type of Q' is M.

2.9 Presentation Layer

While writing queries using the entity algebra allows one to define entity sets in a compositional way, users may like to *display* an answer set using a more elaborate language. Entities should be viewed in ways appropriate to their types. For example, entities with image attributes could have those images displayed. Entities with foreign keys to other entities may have the referenced entity displayed as a component of the original entity. Entities belonging to multiple classes should have the individual displays concatenated in some meaningful way. Entities in an entity set may be heterogeneous; each entity in the set may be displayed differently.

In principle, the presentation language may be more expressive (and have higher complexity) than the entity algebra. We are willing to accept this dichotomy because (a) the presentation language does not have to be compositional, (b) the purpose of the presentation language is different from the query language, and (c) the fundamental constructs of the language may be different. A familiar example of such separation is the "order by" clause in SQL, which can only be applied at the top-level of a query. A relation is fundamentally an unordered structure. Yet, for the purposes of presentation, users benefit from getting their answers in a particular order. Geographical Information Systems provide another example, where the rendering of the query results is (largely) independent of the definition of the query.

The presentation layer can be developed separately from the query language. Custom presentations of entity sets can be applied at each point in a sequence of intermediate queries, but they will not affect the outcome of subsequent query operations applied to these intermediate queries.

In Section 3 we describe an implementation that makes particular choices about how entities are presented. However, alternative presentation language designs are possible.

2.10 Related Work

Our work is orthogonal to work that look at how to model domain hierarchies using XML or some other standard interchange format. In principle, our query system could use any kind of hierarchy representation, although it is likely to work best for one that has a faceted organization.

In systems like Flamenco, there is no formal schema. Entities are tagged with metadata describing their attributes. After a partial search that results in some entity set S, each attribute mentioned by some entity in S is available for further querying. (When a user uses such an attribute, he or she is implicitly limiting the result set to entities having that attribute.) This kind of approach is typical of Information Retrieval applications in which one does not have control over the underlying data. It is also typical of semistructured data models and query languages, although see [13, 14] for ways to infer an approximate schema from semistructured data.

In contrast, we take an approach more typical of conventional structured databases, in which there is a formal schema, and the integrity of the data with respect to the schema can be ensured. For us, an attribute cannot be accessed unless we know that all entities in the underlying entity set possess the attribute. Advantages of our approach include:

- The correctness of a query statement can be ensured at compile-time, without running intermediate queries. A single overall plan for the final query can be generated, rather than forcing a subexpression-by-subexpression evaluation.
- The structure of the output of a query does not change in response to data updates. This is particularly important for the correctness of view definitions.
- Schema conflicts can be resolved. For example, a schema-less system would have difficulty disambiguating metadata tags that happened to share the same attribute name.

Note that we could simulate the Flamenco-style approach by showing all attributes of all entities as part of the presentation language; to process a selection on an attribute A present in just some members of an entity set S, the system can first intersect S with the class defining attribute A.

Object-oriented models [12] organize the data hierarchically, and make "objects" the central concept. However, object-oriented models are usually extensions of object-oriented programming languages, in which an object has a single type. The only way to obtain objects with the characteristics of multiple types is to define new classes that inherit from multiple parent classes (multiple inheritance). In general, such an approach requires a combinatorial number of classes, corresponding to all semantically possible combinations of classes.

Our work can be viewed as an algebraic formulation of a limited description logic [6, 7, 8], with roles being representable by the constraint language. The algebraic formulation allows us to explcitly compare the entity algebra with the relational algebra, and to directly use database engines that implement relational operations. Our representation of hierarchies is similar to that of description logics and conventional semantic data models [10].

3 Implementation

We now give a brief overview of our implementation to demonstrate how it supports the entity algebra. A more complete description of our implementation will be presented elsewhere. We have implemented two applications, one based on human anatomy and one based on an archeological excavation. For brevity, we describe just the archeology application, which is being used for a real archeological excavation [1].

Our system stores its underlying information in a special format using a commercial relational database system. A query engine interacts with the underlying database to implement the entity algebra operations. A lightweight client, implemented using Java Servlets, provides a user interface that interacts with the query engine over the Internet through a browser. Data within our system cannot be directly updated; it may be periodically refreshed from the external source database(s).

The query engine takes a query formulated in the entity algebra, expands all subexpressions, and converts the entire query into an SQL query over the stored data. The results of the query are returned to the user interface. The current implementation uses a very simple constraint language: a basic constraint is an equality between an attribute value and a constant. Distinct constants are not equal. Basic constraints can be combined using conjunction and disjunction.

The user interface uses text to express query operations rather than explicitly presenting the algebra, so that users familiar with the application domain (but not with the algebra) can use the system effectively. The interface is designed so that complex queries can be assembled from simpler pieces, where each piece corresponds to a subexpression in an entity algebra query. Users have access to past query results when formulating subsequent queries. The user interface also supports shortcuts, so that frequently accessed classes or subexpressions are pre-loaded into the list of past query results. Commonly used relationships are also directly expressed. For example, if selecting objects based on the distance between the object's location and some other location is a commonly used operation, the distance function on points is made available for use within semijoin operations.

The presentation layer is implemented through code plug-ins. As the client application is implemented using Java servlets, the details and style of the presentation can include formatted text, images, audio, and video.

Figure 2 shows a screenshot of the current user interface for the query system.

4 Conclusions

We have described the entity algebra, a query language designed for posing queries over complex faceted hierarchies. We have examined its complexity and expressive power. It achieves linear space and quadratic time data complexity. Yet it retains most of the expressive power of the relational algebra for queries returning sets of entities; only projections and joins with cyclic hypergraphs are "excluded." An implementation of the language is described, with particular focus on an application in archeology.

References

[1] The Amheida project. `http://www.mcah.columbia.edu/amheida`.

[2] The FacetMap project. `http://facetmap.com`.

[3] The Flamenco project. `http://bailando.sims.berkeley.edu/flamenco.html`.

[4] M. J. Bates. How to use controlled vocabularies more effectively in online searching. *Online*, 12(6):45–56, 1988.

[5] M. J. Bates. Indexing and access for digital libraries and the internet: Human, database, and domain factors. *Journal of the American Society for Information Science*, 49(13):1185–1205, 1998.

[6] A. Borgida. Description logics in data management. *IEEE Transactions on Knowledge and Data Engineering*, 7(5):671–682, 1995.

[7] A. Borgida, M. Lenzerini, and R. Rosati. Description logics for databases. In *The Description Logic Handbook*, pages 472–494. Cambridge University Press, 2002.

[8] D. Calvanese, G. De Giacomo, and M. Lenzerini. Description logics: Foundations for class-based knowledge representation. In

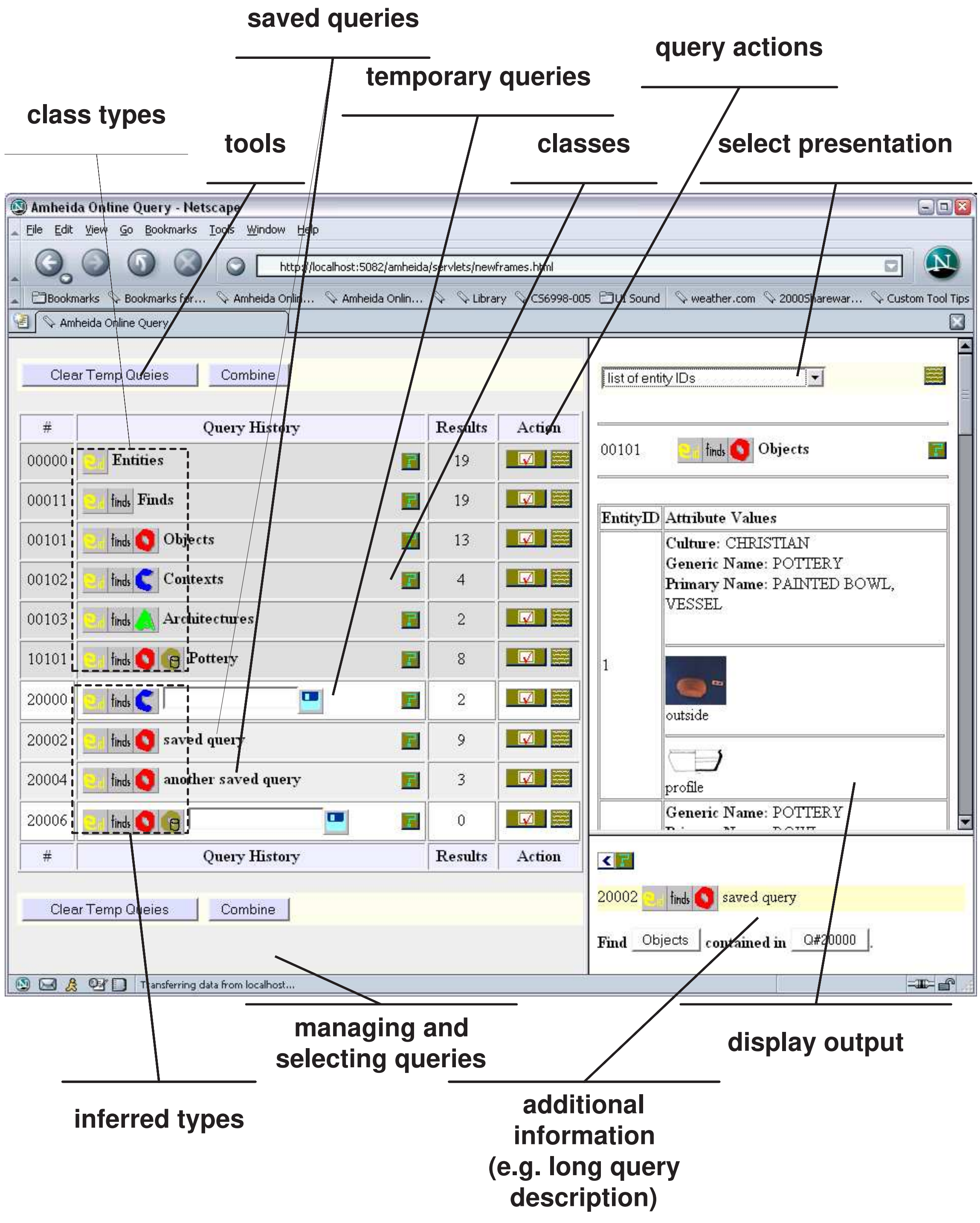

Figure 2: User interface screenshot

Proc. of the 17th IEEE Sym. on Logic in Computer Science, pages 359–370, 2002.

[9] J. English et al. Flexible search and navigation using faceted metadata. Submitted for publication; available from `http://bailando.sims.berkeley.edu/flamenco.html`, 2002.

[10] R. Hull and R. King. Semantic database modeling: Survey, applications, and research issues. *ACM Computing Surveys*, 19(3):201–260, 1987.

[11] J. Jaffar, S. Michaylov, P. J. Stuckey, and R. H. C. Yap. The CLP(R) language and system. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 14(3):339–395, 1992.

[12] A. Kemper and G. Moerkotte. *Object-Oriented Database Management*. Prentice Hall, 1994.

[13] S. Nestorov, S. Abiteboul, and R. Motwani. Infering structure in semistructured data. *SIGMOD Record*, 26(4):39–43, 1997.

[14] S. Nestorov, S. Abiteboul, and R. Motwani. Extracting schema from semistructured data. In *Proceedings of the ACM SIGMOD conference*, pages 295–306, 1998.

[15] J. D. Ullman. *Principles of Database and Knowledge Base Systems*. Computer Science Press, Rockville, MD, 1989. (Two volumes).

[16] B. S. Wynar. *Introduction to Cataloging and Classification*. Libraries Unlimited, Inc., 8th edition, 1992.

[17] M. Yannakakis. Algorithms for acyclic database schemes. In *Proceedings of the VLDB conference*, pages 82–94, 1984.