A COMPARISON BETWEEN DEDUCTIVE AND OBJECT-ORIENTED DATABASE SYSTEMS

Jeffrey D. Ullman
Stanford University

ABSTRACT

The author will not pretend to be unbiased in the matter of comparing the deductive and object-oriented approaches to new database systems; he believes that despite some important concepts originating with the object-oriented approach, the deductive family of systems will ultimately dominate. We shall explore some of the reasons why the two approaches do not mix well. Fortunately, most of the features of object-oriented database systems can be incorporated in deductive systems (but not vice-versa). Then we argue that declarativeness is important for at least some of the new applications, and that declarative languages cannot be object-oriented in a nontrivial way. We next contrast the approaches with regard to the way they classify data elements. We close with an examination of the prospects for "classless" data, which is motivated by the prospect of databases where the number of distinct classes would be too large to name systematically. There are a number of challenges to implementing such facilities in either paradigm, and we propose some appropriate research questions.

1. INTRODUCTION

Throughout this paper, we assume the reader is familiar with the basic notions of relational database systems. We also assume acquaintance with datalog, logical rules, and the least-fixed-point semantics of such rules. We cheerfully recommend Ullman [1988, 1989] for the necessary definitions and concepts.

Deductive Systems

Research in deductive database systems is fairly regarded as beginning with the book by Gallaire and Minker [1978], although we could argue that it wasn't until the mid-1980's that research in the field began to focus on the issue of optimization, which is essential if such systems are to become realistic prospects. These systems are based on predicate logic as a data model, often following a Prolog style of using Horn-clause, or if-then, rules to define predicates or relations.

Example 1: The following, classical example of the "same generation" problem will illustrate the style.

```
(1) sg(X,Y) :- flat(X,Y).
(2) sg(X,Y) :- up(X,Xpar) & sg(Xpar,Ypar) & down(Ypar,Y).
```

In this example, there are three relations or predicates, *flat*, *up*, and *down* that are presumed to be stored physically in the database (*EDB*, or "extensional database," predicates). The fourth predicate, *sg*, is defined implicitly by the rules (it is an *IDB*, or

Work supported partially by NSF grant IRI–90–16358, Air Force grant AFOSR–90–0066, and ARO grant DAAL03–91–G–0177.

"intensional database," predicate).

Rule (1) says that individuals X and Y are of the same generation if they are related by the $flat$ predicate, which we may think of as "sibling." Rule (2) says that X and Y are at the same generation if we can find individuals $Xpar$ and $Ypar$, which we may think of as the "parents" of X and Y, respectively, such that X and $Xpar$ are related by *up* (so *up* represents "parent of"), $Ypar$ and Y are related by $down$ (so $down$ represents "child of," or the inverse of "parent of,"), and $Xpar$ and $Ypar$ are relatives at the same generation.

The combination of rules (1) and (2) is a logic program. Its least fixed point, or "meaning," is the relation for predicate sg as a function of the EDB relations for *flat*, *up*, and $down$. We may think of the program as computing generalized cousins (including siblings). □

Object-Oriented Systems

Work on object-oriented database systems has emphasized system development and enunciation of principles, as in Atkinson et al. [1989]. Roughly, an object-oriented database system is one that supports the classical features of database systems, which were perhaps best summarized by Bancilhon [1988] as systems that support "large amounts of persistent, reliable, shared data." When we add to these requirements the object-oriented programming paradigm found in languages such as Smalltalk and C++, we have an object oriented database system.

But what are the important features of an object-oriented programming language? We can identify three such.

1. *Object-identity.* "Objects" are postulated to exist and have meaning independent of their properties. For example, two students with the same name and all other properties identical as far as the database is concerned are still recognized as distinct objects if they were created separately in an object-oriented system.

2. *Classes.* An object-oriented system has an associated type system, and this system usually is quite powerful, allowing at least the construction of new types by record-formation, set-formation, and reference. For example, the type of the class $Student$ might be a record consisting of a name, ID number, and a set of courses. The members of the course set would themselves be references to $Course$ objects in another class. Usually associated with classes is *encapsulation*, the notion that operations on objects in a class are limited to certain *methods* defined for that class. Internal representation for a class is not accessible to the user.

3. *Inheritance.* Classes fall into hierarchies, and the methods for a class are inherited by its subclasses, unless superceded.

4. *Late Binding.* Methods may be applied to data of various classes, and the interpretation of a method name cannot be made until run time, when we see the class of the object to which it is applied.

2. DEMAND-PULL BY NEW DATABASE APPLICATIONS

Both deductive and object-oriented approaches to database systems are motivated by new applications of database technology. While the earlier generation of database systems grew in response to a class of important, record-oriented needs, such as banking or airline reservation systems, there is a large population of new applications that seem to require some database technology but are not served well by the earlier systems. We can cite design and concurrent engineering applications, integration of heterogeneous, independent databases, applications involving multimedia data, and "data dredging" applications, for example. These and their requirements are summarized in Silberschatz et al. [1990]. Often, the data for such applications has much more complex structure than is found in the earlier record-oriented applications, and we find the need for more complex operations and multiple levels of abstraction as well.

Extending the Relational Model

Both the object-oriented (hereafter OO) and deductive-database (hereafter DD) approaches attempt to extend the relational model in some way. In the OO approach, the notion of "class" replaces the relation scheme, and the extension is primarily through

1. The introduction of arbitrary, user-defined functions, or "methods," and
2. The extension of the simple data structure used by relations (sets of tuples) to a rich set of user-definable types.

Baldwin [1991] surveys commercial OO database systems, and Bancilhon and Kim [1990] survey some of the more experimental systems, including O_2 (Deux et al. [1990]) and Orion (Kim, Garza, Ballou, and Woelk [1990]).

The DD approach, on the other hand, has been more conservative about its extensions. Often, there is the ability to create more general data types, through the logical tool of function symbols and terms as arguments. Sets are sometimes supported as an independent concept, and lists appear naturally through the mechanism of function symbols. However, the major additional power in DD systems comes from the use of a more powerful form of logic. While relational systems are built on first-order logical expressions as the query language, DD systems use collections of first-order expressions. These are usually written as Horn clauses, or if-then rules, and have a semantics based on fixed points, or some more complicated ideas for disambiguating collections of rules without a unique least fixed point, such as stratification (Van Gelder [1986], Apt, Blair, and Walker [1988], Naqvi [1986]), or well-founded semantics (Van Gelder, Ross, and Schlipf [1991], Ross [1991]).

The systems of this class are generally less advanced toward commercialization, but there are a number of experimental systems under development. LDL (Naqvi and Tsur [1989]) is the first, and most advanced of this class, and can be considered ancestral to all. Other projects of this type include

- CORAL (Ramakrishnan [1990]) at the Univ. of Wisconsin,
- Aditi (Vaghani et al. [1990]) at the Univ. of Melbourne,
- LOGRES (Cacace et al. [1990]) at the Univ. of Milan,
- DedGin (Lefebre and Vieille [1989]) at ECRC, and

- The author's own NAIL project (Ullman [1989], Phipps [1990], Phipps, Derr, and Ross [1991]) at Stanford.

3. THE COMPETITION BETWEEN OO AND DD APPROACHES

Like the OO systems, the DD systems in this group offer the power of a Turing machine in their query language (which of course, earlier systems such as relational systems do *not* provide), so in a sense either should be an adequate extension. However, one soon notices three important differences.

1. While programming in an OO language is fairly conventional, programming in logical languages seems to be difficult at times, especially when the semantics is least-fixed-point semantics.
2. While the code written by applications programmers of OO systems can generally be run as written, code in the language of a DD system needs serious code optimization if it is to perform competitively.
3. Programs written in a logic language are more "declarative" than OO programs, in the sense that one can express algorithmic ideas at a very high level, without having to specify many of the details of order of computation.

Point (1) suggests that DD systems may be a hopeless cause, while point (2) answers the question posed at a recent panel: "Why are OO people writing systems, while DD people are writing papers?"

In the following sections, we shall try to explain why our initial perceptions regarding (1) may be wrong. We shall argue in Section 4 that

- It is not possible for a query language to be seriously logical and seriously object-oriented at the same time.
- To the extent that it is possible to combine the two philosophies, OO and DD, it is best done with a logical base, rather than trying to add logic to an OO system.

Point (3) is also contentious. OO advocates will point out that the object-oriented framework allows one to build levels of abstraction easily and that at a high-enough level, one can program as succinctly as with a logical language. We do not agree completely, and we shall address this matter in Section 5.

Finally, in Section 6, we shall look at the difference in the ways DD and OO systems classify facts into groups (classes, relations, or predicates, e.g.) and the implications of each style on the way these facts can be addressed or accessed. It appears to be an important goal of DD research to provide the versatility of access that OO systems provide.

4. WHY OO AND DD DON'T MIX

We shall discuss in this section two arguments for why one cannot have a system that is both truly deductive and truly object-oriented. These are

1. If you are serious about object-identity, then logical fixed points can get you into unneeded trouble.

2. If you are serious about the extensibility of your type system, then you cannot program in a declarative way.

Declarativeness and Object-Identity

First, let us examine "declarativeness," a property of a programming language that intuitively means one can program at a sufficiently high level that one expresses what one wants with relatively little detail about how the answer is to be computed. There is reason to believe that declarative languages are inherently logical, or that they are algebraic versions of some logic. The most common, and probably most important, example is relational algebra and calculus, which are known equivalents and jointly form the backbone of SQL and other relational query languages (see Ullman [1988]).

On the other hand, "object identity" is a notion that says each atom of data is distinct from all other atoms of data by having its own identity. The opposite, which can be termed *value-oriented* data, says that two atoms of data are the same if their value is the same. Relational systems are value-oriented, in the sense that two identical tuples in a relation are regarded as the same tuple (although implemented systems do not always eliminate duplicates, both for efficiency reasons and to make certain aggregation operators like SUM work properly).

Example 2: Let us consider what happens when we try to use a declarative logic language and also take object-identity seriously. The following logical rules express the definitions of paths in a graph from a database of *arc* facts.

r_1: `path(X,Y) :- arc(X,Y).`
r_2: `path(X,Y) :- path(X,Z) & arc(Z,Y).`

That is, r_1 says any arc is a path, and r_2 says that a path followed by an arc is also a path.

In a value-oriented interpretation, these rules produce as a relation the set of "path facts." That is, $path(a, b)$ is true if and only if there is a path from a to b in the graph defined by the *arc* relation. The set of true *path* facts can be computed in any of a number of simple ways, and the *path* relation is always finite if the *arc* relation is finite.

However, suppose we start treating object-identity seriously. Then *path* facts proved in different ways should be treated as different, that is, given distinct object identities. In particular, different paths from node a to node b result in different path-objects, and if the graph has cycles, there is an infinity of different path-objects generated from a finite *arc* relation.

Thus, taking object-identity seriously has converted a simple pair of logical rules, with a simple, finite, least-fixed-point semantics, into something whose meaning is an infinite set. □

Object Identity in Logic

Several years ago, the author gave several talks on the subject of OO and DD databases, with Example 2 included. It was also pointed out that in pure logical terms, enforcing object identity on rules r_1 and r_2 has the effect of converting them to rules with function symbols, such as

```
s1: path(X,Y,f) :- arc(X,Y).
s2: path(X,Y,g(I,Z)) :- path(X,Z,I) & arc(Z,Y).
```

Here, the third argument of *path* serves as an "object identity," since for each *path* fact, the third argument, which will be of the form

$$g(g(\cdots g(f, Z_1), \ldots, Z_{k-1}), Z_k)$$

gives us the list $Z_1, \ldots, Z_k$ of intermediate nodes on the path from the first argument of *path* to the second argument.

Since the rules s_1 and s_2 have an infinite least-fixed-point whenever there are cycles in *arc*, it was argued then, as we have done here, that object-identity is too great a burden if we are to use a logical language. The paper by Kifer and Wu [1989] in effect stood this argument on its head, pointing out that one could simulate object identity if one wished, in the manner suggested by rules s_1 and s_2. That is, function symbols can always be used in an extra argument to provide a trace of the derivation of a fact, and thus, if we wished, to provide distinct values in one argument of facts that would otherwise look the same.

Further, Kifer and Wu show how limited forms of object identity can be simulated by function symbols as well. That is, rules s_1 and s_2 might make us happy if there were no cycles in the graph, but if there were cycles, we might only wish to distinguish paths in some finite way, say by the second node in the path.

We should also mention the paper on "HiLog" by Chen, Kifer, and Warren [1989] in this regard. That paper shows how to simulate complex object types, especially sets, in first-order logic. Combined with the ideas in Kifer and Wu [1989], we can get the essence of object-oriented programming in logic.

Dynamic Typing and Ad-Hoc Queries

Our second argument why logic and object-orientation don't mix concerns dynamic type systems, that is, object-oriented languages in which it is possible to declare new types at run time. A very important feature of DD systems is the ability of the user to phrase ad-hoc queries, that is, short, easily written queries in a very high-level language. We claim that it is not possible to mix ad-hoc queries with dynamic typing.

If we join a relation $R(A, B)$ with a relation $S(B, C)$, we produce a new relation $T(A, B, C)$. Similarly, in logic, we can write a rule

```
t(A,B,C) :- R(A,B) & S(B,C).
```

with the same effect.

The relation T may have a new "type," since we may never before have encountered a relation with attributes A, B, and C. Nonetheless, the operations of relational algebra are immediately available for the relation T, because they are defined for any relation scheme in advance.

It is this predefinition of operations that makes ad-hoc querying possible in relational or logical systems. With it, we can build arbitrarily complicated expressions quickly; without it, we must stop each time we create a new type and laboriously write programs for whatever operations we need.

Destroying the Village to Save It

OO advocates will undoubtedly regard this argument as a fraud. The reason relational or logical systems allow new "types" to be created on the fly is that their type system is so poor. There isn't much you can do to relations, so you may as well define all the operations you can do once and for all.

They will also point out the advantages to an OO system, where you can define a rich collection of types, that may include relations of all arities and many other structures as well, predefine a rich set of operations for these structures, and then make ad-hoc queries to our heart's content. Further, OO systems typically define a few basic operations like read and write for components of any type, including dynamically declared types.

During the Vietnam war, a favorite excuse for having leveled a village where a few Viet Cong were suspected of hiding was "we had to destroy the village to save it." The argument about dynamic types is analogous. Sure, we can have dynamic types and ad-hoc queries, as long as we don't use it. We must destroy the feature to save it.

Admittedly, one can consider type systems richer than relations, and still have a logical language and ad-hoc query capability. Some of these type systems may in fact be very useful. However, we should see them for what they are, new data models rather than a way to get dynamic typing into a logical system.

5. THE NEED FOR DECLARATIVENESS

Since many of our arguments hinge on the need for declarativeness in our query language, we should examine the concept in detail. Recall that we have defined a declarative language as one in which it is possible to express what one wants with relatively little attention to how the answer is to be computed. Declarativeness is thus a relative concept, but we shall take as our base that conventional languages like Pascal are not declarative, while languages that let us program at a significantly higher level, like SQL, APL, or logic languages, are declarative.

Object-Bases

The first question we should examine is whether the new applications require declarativeness in their query language. Clearly some do not. For example one hears the argument that it would not make sense to store the data of a compiler in a DD or relational system, because the structures are fixed, the compiler has no need for ad-hoc queries, and performance is vital. Similar arguments apply to applications of the new database technology that are more realistic than a compiler. For example, we would not expect ad-hoc queries to be useful in most components of a CAD system.

Applications where there is a rich structure, efficiency is vital, and there is no need for ad-hoc queries require an "object-base" or "object-oriented back end." This kind of system is an important and useful development. Object-bases support a powerful type system, but they do not support (directly) a powerful query language. Object-bases can be a useful back end for either a DD or OO database system. But we should not confuse an object-base with an OO database system.

Applications That Require Ad-Hoc Queries

For the kinds of applications that do require ad-hoc querying, and therefore declarativeness, we should examine whether it is possible to use an OO language, and we shall conclude that it is not really practical to do so. But perhaps first we need to consider whether any of the new family of applications require both the powerful query languages of DD or OO systems and also ad-hoc query capability. We suggest the following classes of applications do require the system to respond to unexpected or unplanned queries.

1. *Scientific databases.* Important applications in this class will allow the scientist to discover new relationships among data. For example, satellite data could be used to discover relationships between temperature, location, vegetation, or a host of other variables. It is in the nature of scientific discovery that we cannot know in advance what questions will be asked.

2. *Market databases.* Ad-hoc querying of market data will allow improved financial decisions at a number of levels. For example, investors in the stock or commodity markets need to discover patterns and use them to predict the future. A query that has been planned in advance cannot offer an advantage, because it will then be "common knowledge" in the market and its effects will be canceled. Thus, market databases must support ad-hoc queries. Moreover, since time is of the essence, the relevant figure of merit is the sum of the times to write the query, debug it, compile it, and run it. That formula puts a strong premium on a very high-level, declarative language.

3. *Data Dredging Applications.* There is a large family of "data dredging" problems that are characterized by the need to look for patterns that vary over time. For example, Silberschatz et al. [1990] discusses a department store that stores a terabyte of data from its cash registers and has buyers examine the data for developing patterns of sales. By better predicting trends, and being able to move merchandise from one store to another where the demand is greater, they are able to improve sales by 3%, a significant amount in an industry where profit margins are rarely that great.

A variety of other new applications that need ad-hoc query capabilities can be found in Tsur [1991].

Declarativeness and Methods

Now, let us consider whether OO systems can duplicate the declarativeness and ad-hoc query capability of DD systems. Superficially, it appears not, since OO languages are all at the conventional level of abstraction, and a language like Smalltalk or C++ is no more declarative than Pascal or C.

However, an important argument for object-oriented programming in general is that it allows the creation of classes at progressively higher levels of abstraction, leading to data types and operations on these types that are at least as high as those found in relational or logical languages. That is surely true. For example, we could imagine a class of relations defined, with methods such as $join$, so when we send the message $join\ S$ to a relation R, we get the join $R \bowtie S$.

But there is more to declarativeness than programming at a very high level. In order to have a successful, declarative language, we need to be able to optimize queries and answer them in time close to what the best plan that a programmer could devise would take. One can argue that the reason very high-level languages like APL have not been more successful is that they have not proven amenable to an adequate degree of optimization. In fact, SQL and similar relational languages probably form the only examples of successful optimization for languages at a level significantly higher than Pascal.

One can conjecture about the reason. Probably part of the story is the fact that relational languages are not Turing complete, so one might expect to understand them better for the purposes of writing an optimizer. However, it is also clear that the relational model, or its equivalent in logical rules, has a particularly clean and simple semantics, leading to relatively simple strategies for producing optimal equivalents to a given expression of relational algebra.

Example 3: Consider the join of three relations, $R(A, B)$, $S(B, C)$, and $T(C, D)$, projected onto attributes A, and D, which we might write in relational algebra as $\pi_{A,D}(R \bowtie S \bowtie T)$, or in an equivalent logical rule

```
answer(A,D) :- R(A,B) & S(B,C) & T(C,D)
```

Depending on the data, it may be more efficient to join R with S first, or it may be preferable to join S with T first. An optimizer can easily isolate this issue, and based on statistics of the data, make a choice. Since one or the other join order could be orders of magnitude more expensive than the other, this optimization is vitally important. Whether expressed in algebra or in logic, the semantics of the notation makes it quite clear that either join order is an option. □

Now let us suppose that we define, in some OO system, a class that behaves like binary relations, and we have a *join* method for this class, so we can write an expression like

```
R join S join T
```

It appears we could evaluate this expression as

$$(R \bowtie S) \bowtie T$$

or as

$$R \bowtie (S \bowtie T)$$

But can we? It was never stated what the *join* method means. Is it associative, for example? If not, we cannot replace the left-to-right evaluation order by the right-to-left order; we have to take whichever order the language rules require.

It might be possible for the optimizer to examine the code with which the *join* method was defined. But the technology for machine "understanding" of code written in Pascal-level languages is not yet with us, and it is hard to see how to write an optimizer that could tell whether, say, a Pascal function defined an associative operation.

The conclusion is that, if we want to program in an object-oriented way, and still program at the level of relations or higher, we need to give the system at least the information embodied in the algebraic laws of relational algebra. That information cannot be deduced by the system, but must be built into the it. Thus, we are left again with a data model, perhaps the relational model, perhaps something more complex, but the only part of the query language that will be optimizable is the fixed set of methods for which an adequate semantics have been provided to the system. Again, we have destroyed the village to save it. We can have an object-oriented language as long as we freeze it and treat the methods as operators in some well-understood algebra.

6. MODELING IN DD AND OO SYSTEMS

To this point, we have examined certain features of DD and OO systems and concluded that there are some incompatabilities (Section 4). We have also argued that it is unlikely we shall be able to program at a very high level in a truly object-oriented system, and still produce optimized code (Section 5). Now, let us address an issue where DD systems seem to be deficient: the flexibility with which data can be represented.

From the beginning, data models have incorporated a notion of an "entity set" or collection of similar entities. Relations are the obvious embodiment of an entity set in the relational model, and the predicate serves in DD systems. Clearly the class is the corresponding notion in OO systems. However, these various embodiments of entity sets have different degrees of flexibility and ease of use.

- A relation has a fixed arity, and its arguments are given names (attributes).
- A predicate represents a relation and also has fixed arity, but its arguments have no names and meaning is associated with an argument position.
- A class generally does not have a fixed set of components, because it may have subclasses with different sets of fields or instance variables from what is found in the class itself. Components are known by name, rather than position.

Names Vs. Positions for Components

It is frequently more convenient to refer to components by name than by position in a list, especially if an entity set has many attributes. That is a feature of relational notation over logical notation. However, indicating components by position is not fatal; we've been using this system for arguments of procedures for decades, without the development of a serious movement to modify Pascal or similar languages.

Extensibility of the Component Set

Far more important than the position/name question for components is the ability to have variable sets of components, attributes, or methods (whichever term is appropriate) associated with an entity set. We believe that the extensibility of component sets in OO systems gives them an advantage over DD systems, and we need to simulate this capability in DD systems.

Example 4: To focus on the problem, let us consider the entity set *student*. In a plausible database, some students, but not all, would have attributes such as *advisor*, *spouse*, *TA-course*, *GRE-score*, *major*, *home-addr*, and many others. These attributes are all simple fields, so in an OO system the associated methods probably would be the trivial read/write methods. Thus, relational, DD, and OO systems would not treat these too differently.

However, in an OO system we might find a method for the *student* class like GPA that required execution of a fairly complex program on the data stored for a student. That sort of method might present problems for other kinds of systems. □

Handling Extensible Component Sets in Relational Systems

In relational systems, a problem such as that posed by Example 4 might be solved through null values. We would define a student relation with all the attributes any student might have. The absence of a value for an attribute can be indicated by a null value.

That approach is probably adequate if the system supports the notion that nulls are missing values, and the average number of nulls for a typical student is small. If the typical student has mostly null values, then we are wasting a lot of space, and probably time as well, since the nulls spread the useful data across a wide area of storage.

A nontrivial method like GPA mentioned in Example 4 can probably be handled as a view, since relational systems support aggregate operators like taking averages. However, in the most general case of a method, a relational system cannot simulate the OO system; the relational languages are simply not expressive enough.

Handling Extensible Component Sets in DD Systems

The variability of component sets is usually achieved in OO systems through the subclass system, where a class is divided into subclasses with different sets of methods. An inheritance mechanism for DD systems would serve the same purpose, and the reader should look at Zaniolo [1991] for extensions of DD systems to include inheritance.

For simple situations like that of Example 4, we can probably use function symbols to simulate the variable attribute sets without using nulls.

Example 5: We might have a predicate

$$students(i, a)$$

where i is the student ID of the student, and a is the name of another relation wherein is kept, wrapped in the appropriate function symbols, all the data about student i. Following the HiLog style, we might choose to call the set of attributes for each student i by the name $a(i)$. Thus, if 12345 is a student ID, there is a relation in the storage system whose name is the term $a(12345)$.

In the (unary) relation $a(12345)$, there might be values like $givenName(sally)$ or $major(cs)$. That is, the attributes with nonnull value appear as function symbols, wrapped around the value.

More general procedures as methods can also be handled this way. The GLUE language used in the NAIL system (see Phipps [1990]) allows a name to be a GLUE procedure, as an alternative to being a relation. For instance, we might find in relation $a(12345)$ an element $gpa\big(computeGpa(12345)\big)$, where $computeGpa$ is the name of a GLUE procedure that takes the student ID as an argument and computes the GPA by some formula. □

Data Without Entity Sets

Whether entity sets are represented by relations, predicates, or classes, one important function they serve is to assist in the search for entities with particular properties. The first step in a search for an entity is to decide on its entity set, e.g., the relation in which we must search. In relational or DD systems, this decision can even be made by the compiler, so we do not notice the time it takes.

However, there have been some recent proposals that suggest we might be better off without the classification that comes by partitioning the world into entity sets. The object-oriented programming language Self (Ungar and Smith [1987]) associates attributes (methods, that is) not with classes, but with the objects themselves. Recent claims are for performance close to that of C and much better than that of Smalltalk by using that approach. On the database side, there is a recent proposal called the Melampus data model (Cabrera et al. [1990], Richardson and Schwarz [1991a, b]) that regards objects as part of one set, undifferentiated by anything but the methods to which they respond.

Example 6: Consider a database in which every person in the world was represented. In the OO approach, we would have a class *people*, probably with many overlapping subclasses. The DD approach would probably wind up with one predicate for each property of people, associating the ID for a person with the value of the property.

However, look at the scale of the problem. There are several billion objects, and probably thousands of attributes or methods that are relevant to a subset of the objects. Most likely, the number of different sets of methods to which one or more objects responds is in the billions, because almost any pair of individuals will have different sets of properties that are meaningful. Note that is a stronger statement than the trivial observation that individuals most surely differ in the *value* of some commonly held property. Even if we wanted to invent class names for every set of meaningful properties, we would need class names like "unmarried-nonstudent-autoOwner-renter-taxpayer" and terms that are orders of magnitude more unwieldy.

Moreover, we do not need to use class names, as long as every object knows the methods to which it can respond. For example, we can imagine an SQL-like query

```
SELECT STUDENT-ID
    WHERE SPOUSE-NAME = "Fred"
```

that behaves as if it referring to a "universal relation." The intent is that each object for which the methods *student-id* and *spouse-name* made sense would be interrogated, and if the value of the latter was "Fred" then the value of the former would be produced. □

Problems With "Classless" Data

It seems that the lack of relation or predicate names does not present a serious impediment to introducing "classless" data into DD systems. The sample query of Example 6 makes good sense in SQL, and therefore in logic, although we must do away with the insistence on a fixed set of components for a predicate. In fact, there is a well-developed theory of universal relations, summarized in Chapter 17 of Ullman [1989], that shows how to make sense of queries that refer to entities by their attributes rather than by the relations into which they fall, and this theory carries over to DD systems as well.

However, making sense of queries is not the same as being able to execute them efficiently. In universal-relation systems, there is an underlying relational structure known to the compiler, so queries about attributes can be translated into conventional relational queries. In the "classless" systems under discussion, the conventional structure does not exist. An index on the value of an attribute will help locate entities efficiently when such a value is specified. Yet there are, in databases like that suggested by Example 6, far too many attributes to create indexes on all, or even a significant subset of the attributes. Further, if some of the attributes are methods with side-effects, it is not clear that we could ever create a meaningful index for that "attribute."

Open Problems

We thus propose the following open problems for research on "classless" data.

1. Develop the idea of classless data in a logical language. It is necessary to refer to entities by their attributes, and attributes are referred to by name rather than position in some list of arguments. It is necessary that the usual semantics for DD systems be followed, that is, the least-fixed-point semantics or something more powerful like well-founded semantics in cases where there is no unique least-fixed-point.

2. Develop methods for organizing classless data so we can expect to retrieve entities given values of some attributes or methods to which they can respond, yet the amount of space and time spent maintaining indexes is not prohibitive.

7. SUMMARY

In this paper, we have tried to argue the following points.

- Combining the DD and OO paradigms is not possible if we insist on a strict interpretation of certain OO features, especially object-identity and dynamic typing.

- There are applications where declarative and/or ad-hoc queries are not important. In these cases, an OO system may prove desirable. We believe that many of these applications really require an "object-base" or object-oriented data store, and are not applications for a query language at all.

- In applications where declarative, ad-hoc queries are important, we can never use an object-oriented query language in a nontrivial way. High-level operators must be given a fixed meaning understood by the system if efficient query processing is needed, whereupon the object-oriented language can be viewed as simply another data model.

- Whether you use a DD or OO paradigm, the problem of dealing with "classless" data presents an important implementation challenge.

BIBLIOGRAPHY

Apt, K. R., H. A. Blair, and A. Walker [1988]. "Towards a theory of declarative knowledge," in *Foundations of Deductive Databases and Logic Programming* (J. Minker, ed.), Morgan-Kaufmann, San Mateo, pp. 89–148.

Atkinson, M., F. Bancilhon, D. DeWitt, K. Dittrich, D. Maier, and S. Zdonik [1989]. "The object-oriented database system manifesto," RT Altair 30–89, Aug., 1989.

Baldwin, H. [1991]. "The object-oriented database of your desire," *UNIXWORLD*, March, 1991, pp. 87–90.

Bancilhon, F. [1988]. "Object-oriented database systems," *Proc. Seventh ACM Symposium on Principles of Database Systems*, pp. 152–162..

Bancilhon, F. and W. Kim [1990]. "Object-oriented database systems: in transition," *SIGMOD Record* **19**:4, pp. 49–53.

Cabrera, F., L. Haas, J. Richardson, P. Schwarz, and J. Stamos [1990]. "The Melampus project: toward an omniscient computing system," RJ–7515, IBM, Almaden.

Cacace, F., S. Ceri, S. Crespi-Reghizzi, L. Tanca, and R. Zicari [1990]. "Integrating object-oriented data modeling with a rule-based programming paradigm," TR–90.008, Dipartimento Di Elettronica, Polytechnic Univ. of Milan.

Chen, W., M. Kifer, and D. S. Warren [1989]. "HiLog: a first order semantics for higher order programming constructs," in *Second Intl. Workshop on Database Programming Languages*, Morgan-Kaufmann, San Mateo.

Deux, O. et al. [1990]. "The story of O_2," *IEEE Trans. on Knowledge Engineering*, March, 1990.

Gallaire, H. and J. Minker [1978]. *Logic and Databases*, Plenum Press, New York.

Kifer, M. and J. Wu [1989]. "A logic for object-oriented programming," *Proc. Eighth ACM Symposium on Principles of Database Systems*, pp. 379–393.

Kim, W., J. Garza, N. Ballou, and D. Woelk [1990]. "Architecture of the ORION next-generation database system," *IEEE Trans. on Knowledge Engineering*, March, 1990.

Lefebre, A. and L. Vieille [1989]. "On deductive query evaluation in the DedGin system," *Proc. First Intl. Conf. on Deductive and Object-Oriented Databases*, Kyoto.

Naqvi, S. A. [1986]. "Negation as failure for first-order queries," *Proc. Fifth ACM Symposium on Principles of Database Systems*, pp. 114–122.

Naqvi, S. A. and S. Tsur [1989]. *A Logic Language for Data and Knowledge Bases*, Computer Science Press, New York.

Phipps, G. [1990]. "GLUE: a deductive database programming language," *Proc. NACLP Workshop on Deductive Databases* (J. Chomicki, ed.), Aug., 1990.

Phipps, G., M. A. Derr, and K. A. Ross [1991]. "Glue-Nail: a deductive database system," *ACM SIGMOD International Conf. on Management of Data*, pp. 308–317.

Ramakrishnan, R. [1990]. "The CORAL deductive database system," *Proc. NACLP Workshop on Deductive Databases* (J. Chomicki, ed.), Aug., 1990.

Richardson, J. and P. Schwarz [1991a]. "MDM: an object-oriented data model," RJ-8228, IBM, Almaden.

Richardson, J. and P. Schwarz [1991b]. "Aspects: extending objects to support multiple, independent roles," *ACM SIGMOD International Conf. on Management of Data*, pp. 298–307.

Ross, K. A. [1991]. "On negation in HiLog," *Proc. Tenth ACM Symposium on Principles of Database Systems*, pp. 206–215.

Silberschatz, A., M. Stonebraker, and J. D. Ullman (eds.) [1990]. "Database systems: achievements and opportunities," *SIGMOD Record* **19**:4, pp. 6–22.

Tsur, S. [1991]. "Deductive databases in action," *Proc. Tenth ACM Symposium on Principles of Database Systems*, pp. 142–153.

Ullman, J. D. [1988]. *Principles of Database and Knowledge-Base Systems, Vol. I: Classical Database Systems*, Computer Science Press, New York.

Ullman, J. D. [1989]. *Principles of Database and Knowledge-Base Systems, Vol. II: The New Technologies*, Computer Science Press, New York.

Ungar, D. and R. Smith [1987]. "Self: the power of simplicity," *Proc. ACM Conf. on Object-Oriented Programming Systems, Languages, and Applications*, pp. 214–242.

Vaghani, J., K. Ramamohanarao, D. B. Kemp, Z. Somogyi, and P. J. Stuckey [1990]. "Design overview of the Aditi deductive database system," TR–90/14, Dept. of CS, Univ. of Melbourne.

Van Gelder, A. [1986]. "Negation as failure using tight derivations for general logic programs," *Proc. Symp. on Logic Programming*, IEEE, pp. 127–139.

Van Gelder, A., K. A. Ross, and J. S. Schlipf [1991]. "The well-founded semantics for general logic programs," *J. ACM* **38**:3, pp. 620–650.

Zaniolo, C. [1991]. "Object identity and inheritance in deductive databases: an evolutionary approach," in *Deductive and Object-Oriented Databases* (W. Kim, J.-M. Nicolas, and S. Nishio, eds.) North Holland, Amsterdam.