Incorporating Hierarchy in a Relational Model of Data

H. V. Jagadish
AT&T Bell Laboratories
Murray Hill, New Jersey 07974

ABSTRACT

We extend the relational model of data to allow classes as attribute values, thereby permitting the representation of hierarchies of objects. Inheritance, including multiple inheritance with exceptions, is cleanly supported. Facts regarding classes of objects can be stored and manipulated in the same way as facts regarding object instances. Our model is upwards compatible with the standard relational model.

1. INTRODUCTION

The relational model of data has a flat view of the world, with all information expressed in the form of tables. Due to its stark simplicity and its conceptual elegance, it has been successful in becoming a standard for database implementation. However, the flatness of tables prevents a clean representation of hierarchy.

Hierarchy is very popular as a data structuring technique. Tree structures provide a rich expressive technique for managing complex information. Object-oriented systems [12] are the current panacea for many software engineering problems. A central concept in these systems is that of an object class (or type), and of a hierarchy of such classes. Data and procedures are inherited from a general (base) class to more specific (derived) classes, obtained as restrictions of the general class.

Frame-based knowledge representation systems have become popular in recent years [6, 18]. Once more, the idea here is to organize entities into classes and to construct a hierarchy of these classes. Properties asserted for one class are inherited by all sub-classes of the class, and also by all instances of the class.

Hierarchy has also been proposed in the context of database design as a technique for organizing real-world knowledge before building the database schema [22, 23]. Semantic data models have been a focus of attention for the past few years. (See [14] for an excellent survey). Hierarchy plays an important role in well-known semantic models such as SDM [13], IFO [1], and TAXIS [17]. Inheritance with exceptions has explicitly been included in a data language by Borgida in [4].

In this paper, we attempt to fold the concept of hierarchy into a relational model of data storage. We do so by permitting classes to be used as attribute values in a relation. We introduce the concept of negative assertions to permit exceptions. Only two new relational operators are required: *consolidate*, to eliminate redundant tuples, and *explicate*, to flatten a relation to its complete extension. The resulting model is upwards compatible with the standard relational model. Existing databases can continue to be used with our model, and the meaning of the standard relational operators does not change.

Many benefits accrue from exploiting hierarchy in data representation. One can store the class membership once, and use a single tuple with the class name to substitute for many tuples with its constituent elements. If class membership is determined as a function, one could potentially have an infinite number of values that belong to a class. Thus a potentially infinite relation can be stored in constant space. Class membership may sometimes be undetermined or variable. If all members of a class participate in a relation (satisfy a predicate or possess a property), exactly this statement can be stored in the relation through the class mechanism. Contrast this clean representation with the traditional database approach of storing an extension of the class membership as the set of instances that satisfy the predicate, and then in addition storing an integrity constraint that ensures that the extension stored is exactly the membership of the class[1].

Default statements and certain integrity constraints can now be expressed within the data model. Given a statement about a class, an incompatible statement regarding a sub-class or instance can be interpreted as an exception to a default, or as a violation of an integrity constraint.

1. One could, of course, store the class membership in a separate relation and keep only a single tuple with a class name, even in the standard relational model. The problem then is that repeated joins are required causing a degradation in performance.

The intent of this paper is to present a data model that can serve as a standard interface providing "higher level" primitive operators than a standard relational model would in support of hierarchy. This model has been constructed carefully to make efficient implementation possible. A system that implements this model could be used as a back-end for, say, a frame-based knowledge representation system or a semantic net. The provision of the new primitive operators makes it possible to issue less queries to the database from the reasoning system, and permits the efficient evaluation of these more powerful queries within the database.

Other extensions have been proposed to the relational model. For example, the POSTGRES data model [20] makes several extensions to relational algebra to be able to support object-oriented databases. However, inheritance, an important operation in object-oriented systems, is not supported explicitly. We argue that our extension of the relational model should be incorporated in the data model of any back-end support system built for an object-oriented database.

Inheritance has been added to a logic programming language in the LOGIN project [2]. The motivation of this work in the context of logic programming is the same as ours in the context of relational databases. However, we are able to handle exceptions while LOGIN cannot. We hope that the inheritance with exceptions that we provide can effectively be used in support of a deductive database, thereby providing greater functionality than LOGIN. Further work is required to realize this. At a more abstract level, a body of work is rapidly being established in "sorted logics", which are regular logics integrated with a taxonomy of classes (called sorts) [7, 11, 28].

Section 2 presents our data model. Section 3 discusses operations with the data model. We conclude with some discussion in Section 4 of both implementation issues and possible extensions to the model.

2. THE DATA MODEL

First, a short recapitulation of basic relational algebra. Given a set of c *attribute names* $\mathbf{N} = \{a_1, \ldots, a_c\}$, with a domain $\mathbf{D}_i$ for each attribute a_i, define $\mathbf{D}^*$ to be the cartesian product of the domains $\mathbf{D}_i$ for i=1,...,c. A *tuple* is a mapping from N to $\mathbf{D}^*$. A *relation* is a set of tuples defined over the same domain and range.

Alternatively, one can also think of a relation R as specifying a subset of $\mathbf{D}^*$. It comprises an extension of the tuples (or ordered set of attribute values) that satisfy[2] a predicate corresponding to the relation. Rather than store every individual tuple that satisfies the predicate, we would like, in our model, to store only a few tuples, each of which represents many ordered sets of attribute-value mappings that satisfy the predicate.

2. If the "closed world" assumption [19] is made. Otherwise it is the set of tuples that are known to satisfy the predicate with nothing implied about other tuples in $\mathbf{D}^*$ not in R.

To keep matters simple, in Section 2.1 we only consider the case of a single-attribute relation. We extend these ideas to multiple-attribute relations in Section 2.2.

2.1 Single-Column Relations

Rather than have a single large domain over which the value of an attribute can range, define a hierarchy of sub-domains or *classes*. A *class*, for our purposes, is a set C such that $C \subset \mathbf{D}$. We shall write $\forall C$ as the value of a tuple, and mean that the relation includes (or is satisfied by) every value x where $x \varepsilon C$.

Class membership is transitive. If $a \varepsilon A$ and $A \subset B$, we shall say $a \varepsilon B$. In particular, any tuple with a value of $\forall B$ expands to a set of tuples that includes a. In this regard each instance can be thought of as a "level 0" class[3].

Any attempt to generalize real-world knowledge quickly finds itself in trouble due to exceptions. Most systems with inheritance provide some form of exception mechanism [8, 25]. If our class mechanism is to be useful, we too must allow some form of exceptions. We do so by introducing negation. (An appropriate front-end to the database could choose to issue warnings when an exception occurs, completely prevent exceptions, freely permit exceptions, or do one of the three depending on factors such as the class involved).

It is possible to negate a tuple. The negation applies to the relation. A negated tuple with a value $\forall A$ for its sole attribute in a relation R should be read, "for every element of A, relation R does not hold." Note that such negative statements do not create null values or a problem due to partial information.[4]. Through asserting a set of positive and negative tuples one can create exceptions to exceptions in any required exception hierarchy of arbitrary depth.

An *item* is an atomic element or a class, member or subset of D. Every tuple is an item with an associated truth value. The *truth value* of a tuple is a Boolean variable that is true for a positive (normal) tuple and false for a negated tuple. A tuple associated with an item A is written as t_A.

In the presence of negative assertions, there is a problem of an element x belonging (directly or indirectly) both to a set P with t_P being true, and to a set N, with t_N being false. We use the class hierarchy to handle this problem, following the lead

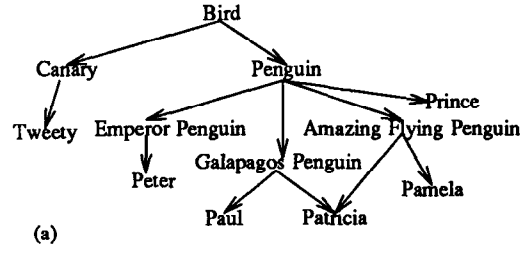
3. A distinction is often made between an instance, which is a leaf node in the hierarchy, and a class, which is a non-leaf node. Such a distinction is necessary, for example, in a programming language context where classes are types in the language and instances are variables of a type. We, in this paper, recognize this distinction to the extent that we treat classes as sets and instances as atomic elements. However, we could just as well treat each instance as a singleton set, and we do just that when convenient, not maintaining a clear distinction between $\{a\}$ and a. Such sloppiness is common in database work [16]. Throughout this paper, we shall use "member of" (ε) and "subset of" ($\subset$) as if they mean the same thing. No confusion results, since a set is always represented by its membership in the database context.

4. A negative statement is NOT equivalent to "it is not true that relation R holds for every element of A, (so that relation R could still hold for some elements of A)." Note also that in the absence of the closed-world assumption, the negated tuple should be read, "for every element of A, relation R is not known to hold."

suggested in [26] and [9].

The *hierarchy* graph for a domain is a rooted directed acyclic graph, with the domain itself being the root and with edges from each more general class to its derived more specific classes. Instances form the leaves of this graph. See Fig. 1a for an example. (If a reasoning system wishes to deal with classes and not consider extensions, then the leaves of the graph could represent classes as well rather than instances).

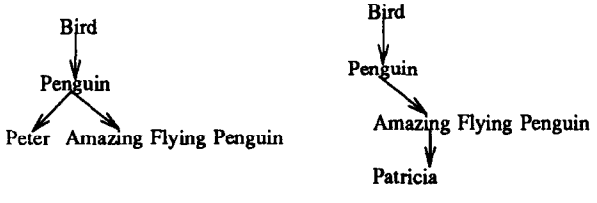
On this graph, define a *node elimination* procedure for a node i as follows: Delete the node i and all edges incident upon it. For each immediate predecessor, j, of i (before the deletion) considered in reverse topological order[5], for each immediate successor, k, of i considered in topologically sorted order, if there does not exist a directed path from j to k (after the deletion) introduce a directed edge from j to k. (By adding edges in the specified order and performing the check to not add an edge if a path already exists, we ensure that redundant edges are not added in the course of the elimination. Such redundant edges are always inefficient to store, and could sometimes lead to incorrect results. See Appendix). For a relation, a subsumption graph is obtained by eliminating all nodes in the hierarchy graph for which no tuples have been asserted.



(a)

Flying Creatures	
+	∀Bird
-	∀Penguin
+	∀Amazing Flying Penguin
+	Peter

(b)



(c) (d)

Figure 1. (a) A Class Hierarchy, (b) A Hierarchical Relation, (c) The Subsumption Graph for the Relation, and (d) The Tuple-Binding Graph for Patricia

5. Given a directed acyclic graph, a topological ordering of the nodes is a complete order of the nodes in which if there is an edge from node i to node j then node i precedes node j.

For any item, a *tuple binding graph* can be obtained by eliminating each node in the hierarchy graph for which there is no tuple in the relation, except the node corresponding to the item in question, and eliminating each node in the hierarchy graph from which there is no path to the item node. Informally, the tuple-binding graph for an item can be obtained from the subsumption graph of the relation by adding the item in question to the subsumption graph along with appropriate links to it as determined from the hierarchy graph, and then deleting all nodes that cannot reach the item. The nodes of the tuple-binding graph represent all tuples in the relation that are relevant to the determination of the truth value of the item in question. If there is a tuple associated with the item itself, then the tuple *binds strongest* to the item in question. Otherwise the strongest binding tuple(s) is the immediate predecessor(s) of the item. The truth value of an item is obtained as the truth value of the tuple that binds strongest to it.

Consider Figure 1. We have a single attribute relation in which we wish to store the names of all flying creatures. Rather than store a huge relation including the name of every individual creature that flies, we use a hierarchy of animals available to us. A fraction of this hierarchy is depicted in Fig. 1a. Now rather than list every bird as a flying creature, we need store only one tuple recording the fact that all birds are flying creatures. From this fact we infer that Tweety, an instance of Canary, which is a type of bird, is a flying creature. That is, an equivalent flat relation would have Tweety as one of its tuples. Penguins are birds but should not be included among flying creatures. We create an exception through a negated tuple, in effect canceling all penguins from the relation. We infer that Paul, a Galapagos penguin, even though a bird, is not a flying creature. If there is a class of Amazing Flying Penguins that are flying creatures, though penguins, we can "re-insert" these into the relation through a positive tuple (creating an exception to the negated tuple regarding penguins). Pamela has three tuples in the relation that are applicable. However, in the tuple binding graph (Pamela's tuple-binding graph is the same as Patricia's shown in Fig. 1d), Pamela has only one immediate predecessor, namely that all Amazing Flying Penguins are flying creatures. We therefore conclude that Pamela is a flying creature. Similarly, there is a specific tuple asserting that Peter is a flying creature, and this tuple overrides all other tuples applicable to Peter.

Classes need not be disjoint. An instance or sub-class could be a member of more than one class. This is known as multiple inheritance. In the absence of negated tuples, multiple inheritance does not pose any problems. The relation is satisfied for an element b if the relation includes a positive tuple t_C for some C such that $b \varepsilon C$.

However, with negated tuples in a multiple inheritance system, we are no longer guaranteed that each item will have exactly one tuple that binds strongest to it. If, for an item, there are multiple tuples of differing truth values as its immediate predecessors in the tuple-binding graph, (and there is no tuple associated with the item itself), then we have a *conflict*. We treat such a conflict as an inconsistent state of the database and do not permit it. (This approach is in contrast with the

standard AI approach of permitting conflicts and then attempting to "resolve" them in a more or less sensible manner [21, 27]). The conflict can be resolved by introducing a tuple that binds more strongly to the item(s) in conflict than all tuples with a different truth value.

The maintenance of consistency is a central database precept. Whenever an update is made we require that the update does not create an unresolved conflict. If an update creates a conflict, within the same transaction, before the update is committed, other updates must be made that resolve the conflict, and themselves create no new unresolved conflict. See Section 3.1 on maintaining consistency in the database.

Many inheritance systems adopt a default conflict resolution scheme (beyond the subsumption rule that we use). For example, left precedence is commonly used in object-oriented programming languages, such as LISP with Flavors. We require explicit conflict resolution in the data model that is implemented as part of the database. A front end can easily be added to provide any desired conflict resolution semantics, including left precedence, by compiling a user generated update request into a transaction that maintains consistency by performing additional updates for conflict resolution.

Returning to the example of Figure 1, consider Patricia, who is both an Amazing Flying Penguin and a Galapagos Penguin. Since nothing has been asserted about Galapagos penguins specifically not being flying creatures, there is no conflict. Patricia's only predecessor in the tuple binding graph is the tuple regarding Amazing Flying Penguins, and we conclude that Patricia is a flying creature. However, if a tuple were to be included in the relation stating that Galapagos penguins cannot fly, then we have a conflict since Patricia has two immediate predecessors in the tuple binding graph, one of them positive, and the other negative.

The functionality provided by single-attribute hierarchical relations is similar to that provided in the semantic nets proposed in [21, 26]. The major difference is that we distinguish between the inheritance hierarchy, which is treated as a taxonomy, and the relations in the database, which associate properties with objects in the taxonomy. This distinction is not usually made in semantic nets so that the set of flying things is considered as much a class as, say, birds. By making this distinction, we lose the ability to infer automatically, from the hierarchy, that, for example, Tweety can travel far since flying things can travel far. However, through the use of logic programming, such as PROLOG or DATALOG, on top of out hierarchical data model, we are able to provide an even more powerful inference mechanism with no loss of succinctness. Moreover, by making this "natural" distinction between taxonomy and association, we are able to provide inheritance over multi-attribute relations, as we shall see below, without having an attendant geometric growth in the size of the semantic net.

2.2 Multiple-Column Relations

For ease of exposition, we had thus far restricted ourselves to relations with a single attribute. In this section, we shall apply the ideas of the previous section to regular multi-attribute relations.

Each attribute of a standard relation ranges over a specified domain. Just as before, we can create a hierarchy of domains for each attribute. An *item* is now obtained as one member (class or element) from each of D_1, D_2, etc., the domains of the various attributes. Thus an item is a subset of D^*, the domain of the relation obtained as the cartesian product of the attribute domains. A tuple is an item along with a truth value that is stored in the relation. The truth value of a tuple can be true or false as before. An *atomic item* is obtained as the "cartesian product" of atomic values from the domains D_1, D_2, etc. A *composite item* is an item that is not an atomic item, so that it has at least one class, not an atomic value, in the cartesian product taken.

An *item hierarchy* is obtained as the cartesian product of the hierarchy graphs for the individual attribute domains. Consider, over some attribute domain, a hierarchy graph (V,E), with V its set of vertices, and E its set of edges. Its cartesian product with another hierarchy graph (W,F) results in a product graph (U,G), where $U = V \times W$ and there exists a directed edge from $u_i = (v_i, w_i)$ to $u_j = (v_j, w_j)$ in G, iff there exists an edge from v_i to v_j in E with $w_i = w_j$ or there exists an edge from w_i to w_j in F with $v_i = v_j$. Figures 2a and 2b show two simple hierarchy graphs, and Fig. 2c shows their product.

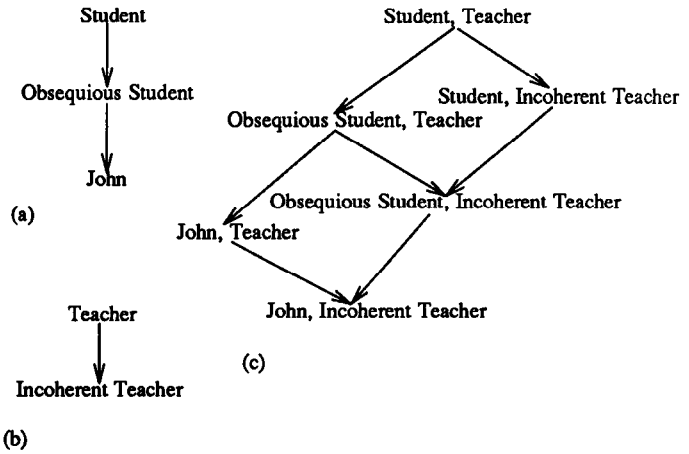


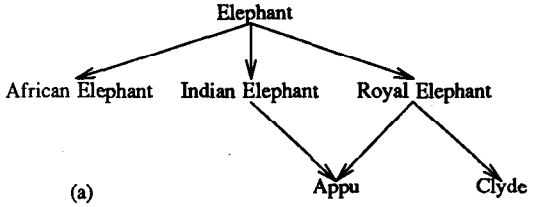
Figure 2. (a) A Student Hierarchy, (b) A Teacher Hierarchy, and (c) Their Product

Respects		
	Respector	Respectee
-	∀Student	∀Incoherent Teacher
+	∀Obsequious student	∀Teacher
-----	----------	----------
+	∀Obsequious student	∀Incoherent Teacher

Figure 3. A relation defined with Fig. 2c as the item hierarchy

A tuple-binding graph can be derived for each item as before. Conflicts of truth values are generated and resolved as before. However, new kinds of conflicts can now be generated. As we can see in Fig. 2, even if each of the individual attribute domains is organized as an inheritance tree so that multiple inheritance conflicts do not arise, in general, the cartesian product of the domains will not be a tree. Consider the

example relation shown in Figure 3 using the hierarchies defined in Fig. 2. Given that all Obsequious students respect all teachers, and that no student respects any incoherent teacher, we cannot determine whether obsequious students respect incoherent teachers. In other words, we have an inconsistent database if the *Respects* relation comprises only the tuples above the dashed line. The conflict is resolved through an explicit tuple asserting that all obsequious students do indeed respect all incoherent teachers.



(a)

	Animal	Color
+	∀elephant	Grey
-	∀royal elephant	Grey
+	∀royal elephant	White
-	Clyde	White
+	Clyde	Dappled

(b)

Figure 4. An Elephant class hierarchy, and an associated relation

For a more involved example, consider a variation on the well-known "Clyde the royal elephant" hierarchy [10] shown in Figure 4. Having said elephants are grey, it is not enough to say that royal elephants are white: we would then be implying that royal elephants were somehow both grey and white at the same time. An explicit cancellation is required, in effect requiring us to state that royal elephants are not grey but white. The same applies to Clyde, an instance of a royal elephant, who happens to be dappled rather than a pure white. Finally, consider a query regarding the color of Appu, known to be both a royal elephant and an Indian elephant. The applicable tuples in the relation refer to elephants and to royal elephants. Royal elephant binds more strongly to Appu than does elephant, so we conclude that Appu is not grey but white. Since there are no assertions specifically about the color of Indian elephants, the fact that Appu is an Indian elephant is treated as an irrelevant fact, just as Appu's membership in any other classes, not shown in the figure, is. See the Appendix for alternate strategies.

Explicit cancellation, of the type we saw in the preceding example, is common whenever a unique property (or procedure) is to be associated with an object. In our generic data model, there is no way of knowing that some property of some entity should be unique. Indeed, one can argue that such independence from specific application semantics is desirable. A front end, whether an object-oriented system or a frame-based knowledge representation system, can generate the negation of the "inherited" tuple automatically whenever an exception is stated, if that is the desired semantics. Alternatively, explicit cancellation could be used as a check that the user does indeed wish to create an exception. This explicit cancellation is akin to the *excuses* construct described in [5].

3. OPERATIONS

In the previous section we defined an extension to the relational data model in which universal quantification over a class could be used as an attribute, and in which exceptions were permitted through negated tuples. In this section we explore the implications of such a data model in terms of the operations to be carried out on it. In Section 3.1 we look at integrity enforcement, and in 3.2 at redundancy. Two new relational operations, *consolidate* and *explicate*, are introduced in Section 3.3. We examine the standard relational operations in Section 3.4.

Note that every hierarchical relation must be equivalent to a unique flat relation for a given item hierarchy; that is, it has a unique *model* of the atomic items that satisfy the given relation. Any manipulations on hierarchical relations should have the same affect whether performed on the hierarchical relations or on the equivalent flat relations. In this sense, the semantics of relational operators is not altered even in the case of hierarchical relations.

3.1 Integrity

A relational database may include integrity constraints in the form of restrictions on attribute values as a function of other attribute values, restrictions on the number of tuples that satisfy some selection criterion, and so forth, usually as part of a catalog or data dictionary. Such constraints are abundant in standard relational database products and are often implementation dependent. In general, they should continue to work on hierarchical relations as well, modulo the changes in the relational operations discussed in Section 3.4. We say no more about them here.

The hierarchical relational model requires two integrity constraints not relevant in a flat relation. One is the *type-irredundancy* constraint requiring that there be no cycles in the hierarchy graph. This is straightforward and is not discussed further. (See [3] for a careful analysis). The other is called the *ambiguity constraint*:

For each item in the cartesian product of the attribute domains of a relation, either there should be a tuple associated with the item, or every predecessor of the item in the tuple-binding graph (that is, every strongest binding tuple) should have the same truth value.

Either an update that violates this constraint can be disallowed, or the conflict between two classes A and B can be resolved. Such resolution can be through deleting the assertion for either A or B. It could also be through explicitly asserting a tuple t_X for every X member (or subset) of both A and B. The set of all such items X is known as the *complete conflict resolution set, C*. A set $M = \{X \varepsilon C$ such that there does not exist $Y \varepsilon C, X \subset Y\}$, is known as the *minimal conflict resolution set*. The complete conflict resolution set is unique for any given conflict on any given item hierarchy. The minimal conflict resolution set can be derived uniquely from it by virtue of the transitivity of subsumption. Any conflict resolving additions and deletions could themselves generate

new conflicts, which must then be resolved.

One tuple for each item in the minimal conflict resolution set will suffice to resolve the conflict at hand. Less tuples may suffice. If an item can be found that binds more closely to each of two or more items in the minimal conflict resolution set than any tuples with a different truth value, then a single tuple corresponding to this item may suffice for all the included members of the minimal conflict resolution set.

Note that our integrity maintenance is "optimistic" in that two sets are assumed disjoint unless there is evidence to the contrary[6]. Such evidence may be provided in one of two ways. One is for there to actually exist an atomic value (or cartesian product of atomic values) that is an element of both sets. The other is for there to be defined a class (or cartesian product of classes and atomic values) that is a subset of both sets, whether or not there exist any instances of this class. Through the creation of empty intersection classes wherever appropriate, a front-end could force a more "pessimistic" integrity maintenance, without explicitly specifying an extension.

3.2 Redundancy

A cornerstone of the relational model is that each relation is a *set* of tuples, with no tuples being repeated. However, the elimination of duplicates requires effort and many optimizers do not eliminate duplicates after individual operations to the extent safely possible. In fact, "relational" query languages often permit users to maintain and manipulate multi-sets, with duplicates not removed, usually for purposes of efficiency. For example, SQL offers a *SELECT* and a *SELECT UNIQUE*, where the former does not eliminate duplicates and the latter does.

As far as two identical tuples are concerned, hierarchical relations do not differ from flat relations in any way. Duplicate tuples are eliminated in exactly the same fashion as in standard relational databases. However, redundancy elimination is more general than simple duplicate elimination. The former involves removal of all redundant statements so that a minimal set of assertions about the relation is obtained. The latter involves removing exactly identical duplicate versions of tuples.

With regard to tuples that are redundant due to one or more different, usually more general, tuples, matters get complicated. Consider a tuple t_1 with $\forall A$ in a certain attribute, and another tuple t_2 identical to t_1 in all respects, including truth value, except that it has a, a member of A, instead of $\forall A$ in the corresponding attribute. Should t_2 be deleted as redundant, since it is dominated by t_1? The answer depends partly upon the semantics of the assertions of the two tuples. If t_1 was asserted due to a justification different from the one due to which t_2 was asserted, the two tuples should indeed both be retained in the database. If t_1 is later retracted, for example because its justification no longer was valid, t_2 should still remain valid. On the other hand, if t_1 was obtained as a generalization of several assertions such as t_2, it may be appropriate to delete t_2, once t_1 has been inserted into the relation. In general, there is no way for the database to know whether there is any dependence between the justifications for two (or more) tuples, and therefore assumes independence. As such the default condition is to let the two separate tuples t_1 and t_2 coexist, even though the latter contributes little effective additional information. When dependencies do exist, the user can issue a *consolidate* operation to remove duplicates. See Section 3.3.

There also are other reasons why redundancy elimination is problematic in a hierarchical relation. As has been discussed in the previous section, if assertions of opposite truth value are made regarding two incomparable sets with a non-empty intersection, then a conflict is created unless explicit assertion(s) are made regarding the intersection of the two sets. By definition, any such conflict-resolving tuple has to be "dominated" by either one or the other (exactly one) of the conflicting tuples. In consequence, one may hastily arrive at the erroneous conclusion that the conflict resolving assertion was redundant. For example, recall Figure 3. The tuple stating that all obsequious students respect all incoherent teachers may appear redundant since we have a more general tuple stating that all obsequious students respect all teachers. However, the former tuple was specifically added to resolve a conflict, and its elimination would produce an inconsistent state in the database.

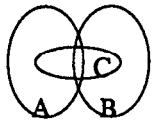


Figure 5. Venn Diagram showing the union of two sets subsuming a third set

It is possible for there to be two sets A and B neither of which individually dominates a third set C, while the union of A and B is a superset of C. See the Venn diagram in Fig. 5. Detecting the redundancy of sets such as C is not easy. In fact, finding the minimum number of sets regarding which assertions have to be made is *np*-hard (the minimum-cover problem, which is known to be *np*-complete, is a special case). Semantically, such a situation is quite awkward, and without a notion of union (or at least, disjoint partition), it is not possible to express the fact that C is a subset of A union B. If such a fact is true of the set membership at some point in time, there is no guarantee that it will remain true indefinitely. Therefore, we cannot consider a tuple regarding C a redundant assertion, given tuples regarding sets A and B.

Finally, one could have the opposite situation of the one above. There could be a set C that is partitioned into subsets

6. In other words, we are making a "closed world" assumption with regard to conflict, saying that there is no conflict unless we have a specific instance of a conflict. Such an assumption with regard to conflict does not require that we make a closed world assumption with regard to the data in the database. Even while acknowledging that there could be facts in the real world that the database does not know about, we can state that there is no conflict between the facts stored in the database. All we require is that there be no query that can be posed to the database that can observe a conflict without performing any updates.

A and B, such that every instance of C is an instance of at least one of A or B. If there are tuples t_A and t_B defined for the sets A and B, then a tuple t_C is redundant, in that it is always overridden by one or the other of the former tuples, for every member of the set C. Such redundancy cannot be detected unless there is a way to express the concepts of partition and mutual exhaustion in the data model. Further study is required to determine exactly what such expressive power should be, and what the complexity of implementation is. Even if we did detect that C was partitioned into A and B, it is not clear that t_C should be deleted, since it may be meaningful if there is a future deletion of t_A, say. Thus, we do not consider tuples such as t_C redundant in our data model.

Formally, the redundancy of a tuple is determined by means of the *subsumption graph* for the relation, obtained by eliminating each node in the item hierarchy graph for which there is no tuple in the relation.

Definition (Redundant Tuple):

A tuple t_A is redundant if and only if it has the same truth value as all its immediate predecessors in the subsumption graph of the relation.

A negated tuple without a (positive) tuple as a predecessor in the relation subsumption graph is redundant. Since negated tuples are used to establish an exception to a generalization and relations are a mapping from $\mathbf{D}^*$ to {0,1}, negated tuples correspond to elements of $\mathbf{D}^*$ that are mapped to zero, just as elements not mentioned in the relation are. To remain consistent with the definition of redundancy above, we introduce the concept of a universal negated tuple defined over $\mathbf{D}^*$, the product of the attribute domains. In the subsumption graph, for every tuple node with no predecessors, we create an arc from this universal negated tuple. The definition of redundancy will then detect negated tuples with no predecessors as redundant.

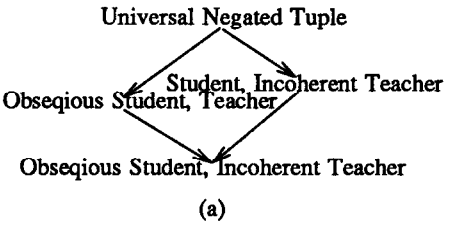
Note that a relation that includes redundant tuples is still a well-defined entity in that it has a unique (and irredundant) extension as a flat relation. In general, there will be multiple tuples in a relation that are applicable to any specific item. That some of these are redundant does not affect anything. Redundant tuples are eliminated in our model only when explicitly requested by the user through a *consolidate*.

3.3 New Relational Operators

In manipulating hierarchical relations, two new operators are useful. These are *consolidate*, to eliminate redundant tuples, and *explicate*, to flatten a relation. Both operations modify the physical form in which a hierarchical relation is expressed, without affecting the information content or altering the equivalent flat relation.

3.3.1 Consolidation

Like all relational operators, *consolidate* takes as its argument a relation, and produces as its result a relation. It "draws" the subsumption graph for the argument relation, determines the redundant tuples from the graph and then eliminates them to obtain the result relation. When a tuple is deleted from the relation, the corresponding node is eliminated from the subsumption graph following the node elimination procedure presented in Sec. 2.1. Since the elimination of redundant tuples alters the subsumption graph, the result of the consolidation will be sensitive to the order in which the redundant tuples are deleted. It can be shown that there is a unique minimum relation with no redundant tuples, and that this miimum can be achieved if the nodes of the subsumption graph are examined in topologically sorted order [15]



(a)

(b)

Respects		
	Respector	Respectee
+	∀Obseqious student	∀Teacher

Figure 6. (a) The Subsumption Graph and (b) Consolidation of *Respec*

Consider Fig. 6a, showing the subsumption graph for the *Respects* relation of Fig. 3. Proceeding in topologically sorted order over the subsumption graph, we notice that the tuple stating that students do not respect incoherent teachers is redundant since its only predecessor is the universal negated tuple. This tuple is then eliminated, but an edge is not created from the universal negated tuple to (obsequious student, incoherent teacher), since there already exists a path between the two, through (obsequious student, teacher). Thus the tuple stating that obsequious students respect incoherent teachers is also found redundant, its only predecessor also being positively asserted. The final result, after both eliminations, shown in Fig. 6b, has exactly the same extension as the relation in Fig. 3, and yet has fewer tuples in it, due to consolidation, even though there were no duplicate tuples in Fig. 3.

3.3.2 Explication

The *explicate* operator takes a relation as its argument, along with a specification of a subset of the attributes of the relation, and produces a relation as the result. The result relation is an (in fact, the only) *extension* of the input relation and has no universally quantified classes as values for the specified attribute; that is, all tuples in the relation after explication correspond to atomic items. This operator is useful when a count, average, or other statistical operation is to be performed over the relation.

Consider the item hierarchy for the relation. Explication involves introducing a tuple for each leaf node in the hierarchy, and removing tuples for all higher level nodes. Many of the tuples thus generated will be negated. The final subsumption graph obtained consists of a set of independent nodes with no links between them. As such, all the negated tuples obtained are redundant, and can be removed by a *consolidate* that follows.

If the explication is only over some attributes of the relation, rather than the entire relation, tuples need be introduced only for selected leaf nodes in the hierarchy. Negated tuples obtained are not redundant, and no consolidation need follow.

In all situations, a simple algorithm to perform explication is to traverse the relation subsumption graph in reverse topologically sorted order. For the tuple at each node, enumerate the membership of classes that are values for the attributes to be explicated. Insert each tuple obtained from such enumeration into the result relation unless a tuple corresponding to the same item has already been inserted.

3.4 Standard Relational Operators

Standard relational operators continue to work with hierarchical relations. For lack of space, rather than describe the algorithms, a few examples are worked out. The interested reader can refer to [15] for details.

We begin, in Figs. 7 and 8, by showing some simple selections on the *Respects* relation of Fig. 3. Whenever one has a system that produces answers that are deduced from, rather explicitly stated in, facts that the system has been told (or has stored in it), the question of *justification* arises. For example, if one obtains an unexpected answer to some query, one may wish to find out how the answer was obtained, either to confirm the correctness of the answer or to debug a poorly specified input to the system. One can, in our model, not only obtain the result of a selection, but also find out which tuples in the relation were applicable. Fig. 9 demonstrates this feature on the relation in Fig. 4.

Respects (selected on first attribute)		
	Respector	Respectee
+	∀Obsequious student	∀Teacher
+	∀Obsequious student	∀Incoherent Teacher

Figure 7. Who do obsequious students respect?

Respects (selected on first attribute)		
	Respector	Respectee
+	John	∀Teacher
+	John	∀Incoherent Teacher

Figure 8. Who does John respect?

	Animal	Color
-	Clyde	Grey

(a)

	Animal	Color
+	∀elephant	Grey
-	∀royal elephant	Grey

(b)

Figure 9. (a) A Selection on the Animal-Color relation, and (b) Its Justification

Set operations apply to the explicated item sets represented by the relations, and not to the actual set of tuples physically used to store the relations. Fig. 10 demonstrate various set

(a)

Jack Loves	
+	∀Bird
-	∀Emperor Penguin

(b)

Jill Loves	
+	∀Penguin
-	Peter

(c)

Jack and Jill Between them Love	
+	∀Bird
+	∀Penguin
-	Peter

(d)

Jack and Jill Both Love	
+	∀Penguin
-	∀Emperor Penguin
-	Peter

(e)

Jack Loves but Jill Does Not	
+	∀Bird
-	∀Penguin
-	∀Emperor Penguin
-	Peter

(f)

Jill Loves but Jack Does Not	
-	∀Penguin
+	∀Emperor Penguin
-	Peter

Figure 10. (a),(b) Two Relations, (c) Their Union, (d) Their Intersection, and (e),(f) Their Set Difference

operations, using the taxonomy example from Fig. 1. Finally, Fig. 11 shows the join of two relations followed by a projection back on one of the original relation. Notice that there is no loss of information in the process.

In many of the examples, notice that redundant tuples are present in the result even when there were no redundant tuples in the arguments supplied to the relational operators. These redundant tuples may be removed, if desired, through a *consolidation.*

4. CONCLUSION

We have presented a model of data incorporating hierarchy into the relational model in a natural way. The model presented is completely upwards compatible with the standard relational model. It features two new options: set-valued attributes and negative assertions. With the help of these it is possible to condense information in a database in a semantically meaningful way. It is also possible to make statements regarding classes of objects within the data model without separately making the statement for each object in the class. Exceptions to generalized statements are permitted. One can use the class mechanism to enforce certain types of integrity constraints within the data model itself. The

	Animal	Enclosure Size
+	∀elephant	3000
-	∀Indian elephant	3000
+	∀Indian elephant	2000

(a)

	Animal	Color	Enclosure Size
+	∀elephant	Grey	3000
-	∀royal elephant	Grey	3000
-	∀Indian elephant	Grey	3000
+	∀royal elephant	White	3000
-	Clyde	White	3000
-	Appu	White	3000
+	Clyde	Dappled	3000
+	∀Indian elephant	Grey	2000
-	Appu	Grey	2000
+	Appu	White	2000

(b)

	Animal	Color
+	∀elephant	Grey
-	∀royal elephant	Grey
+	∀royal elephant	White
-	Clyde	White
+	Clyde	Dappled
+	∀Indian elephant	Grey
-	Appu	Grey
+	Appu	White

(c)

Figure 11. (a) An Enclosure-Size Relation defined over the hierarchy of Fig. 4a, (b) Its Join with the Animal-Color Relation of Fig. 4b, and (c) Its Projection Back on Animal-Color

hierarchical relational model can be used as a basis for implementing a frame-based knowledge representation system. The model shows promise of efficient implementation, though some further work is needed in this direction.

The model presented appears promising in two other respects, both of which are currently the topic of further research. First, through the use of existential rather than universal quantifiers, and the use of three-valued (positive, negative, and unknown) rather than two-valued assertions, it may be possible to have a sound and conceptually pleasing treatment of partial information. Second, we can relax the assumption in this paper that the class hierarchy is specified by the user based upon some semantic notions. Instead, the database system could mechanically organize traditional relation(s) given into hierarchical relations with "classes" being defined in such a way that storage is minimized.

APPENDIX

In the semantic network literature, there are two alternate theories of the correct mechanism to perform multiple inheritance with exceptions [27]. In terms of the tuple binding graph for an item, on-path preemption assumes that a tuple *i* binds more strongly to the item than a tuple *j* iff every path from *j* to the item must pass through *i*. Off-path preemption assumes that a tuple *i* binds more strongly to the item than a tuple *j* iff there is a path from *j* to *i*, in addition to paths from *i* to the item and *j* to the item. Furthermore, there may be applications in which it is not appropriate to assume any preemption at all, and to declare a conflict whenever two or more different truth values are inherited by a node. The techniques presented in this paper are applicable irrespective of the semantics used for preemption. All the relational operations, both the standard ones, and the new ones, stay the same. The difference arises only in the construction of the inheritance hierarchy and the tuple binding graph. We consider each of the possible semantics in turn below.

In all the examples worked out in the paper, we have used off-path preemption, which in most cases appears to closest match human intuition. Off-path preemption is obtained in our model through ensuring that there are no redundant edges in the inheritance hierarchy specified[7]. Thus, for example, a redundant link in the hierarchy of Fig. 1 could be used to state that Pamela is a Penguin. Since all immediate predecessors of a node in its tuple-binding graph are involved in determining the truth value for the corresponding item, Amazing Flying Penguin would no longer bind more strongly than Penguin, and there would be a conflict at Pamela.

No preemption requires exactly the opposite strategy. The transitive closure of the hierarchy graph is used. Every node in the tuple binding graph then becomes an immediate predecessor of the item in question. If there is even one tuple with a differing truth value, we have a conflict.

Considering Fig.1 once more, on-path preemption would suggest that since Patricia is a Galapagos penguin, it may or may not be able to fly, in spite of its being an amazing flying penguin, and in spite of nothing having been explicitly stated about Galapagos penguins! To implement on-path preemption, redundant edges should not be deleted when eliminating a node. Thus, in the derivation of the tuple binding graph for Patricia, at the time that Galapagos penguin is deleted, an edge is inserted from Penguin to Patricia, even though a path already exists from Penguin to Patricia (through Amazing Flying Penguin).

Finally, there may be circumstances in which one wishes to assert some general preference relation [24] over nodes in the hierarchy, so that whenever two nodes have conflicting tuples and apply to some item, then one dominates the other. Such arbitrary preference rules can be introduced by placing special edges in the hierarchy. These edges do not represent set inclusion in the way that the other links in the hierarchy do, but are used to induce the proper tuple binding graph. After these special edges have been introduced, the semantics of off-path preemption apply. Thus whenever there is a conflict at a node in the absence of such edges, the conflict may be resolved through the special edge that renders one of the conflicting immediate predecessors reachable from the other.

7. Formally, we wish to retain only the transitive reduction of the hierarchy graph.

Acknowledgement

I am grateful to Jim Baumann and David Etherington for carefully reading a draft of this paper and for many useful suggestions.

REFERENCES

[1] S. Abiteboul and R. Hull, "IFO: A Formal Semantic Database Model," *ACM Trans. on Database Systems*, **12** , 4 (Dec. 1987) , 525-565.

[2] H. Ait-Kaci and R. Nasr, "LOGIN: A Logic Programming Language with Built-In Inheritance," *Journal of Logic Programming*, **3** (1986) , 187-215.

[3] P. Atzeni and D. S. Parker, "Formal Properties of Net-based Knowledge Representation Schemes," *Proc. 2nd IEEE Int'l Conf. on Data Engineering* , 1986, 700-706.

[4] A. Borgida, "Language Features for Flexible Handling of Exceptions in Information Systems," *ACM Transactions on Database Systems*, **10** , 4 (Dec. 1985) , 565-603.

[5] A. Borgida, "Modeling Class Hierarchies with Contradictions," *Proc. ACM-SIGMOD 1988 Int'l Conf. on Management of Data* , Chicago, IL , June 1988, 434-443.

[6] R. J. Brachman and J. G. Schmolze, "An Overview of the KL-ONE Knowledge Representation System," *Cognitive Science*, **9** , 2 (April-June 1985) .

[7] A. G. Cohn, "A More Expressive Formulation of Many-Sorted Logic," *Journal of Automated Reasoning*, **3** , 2 (1987) , 113-200.

[8] D. W. Etherington and R. Reiter, "On Inheritance Hierarchies with Exceptions," *Proc. AAAI-83* , Washington, D.C. , 1983, 104-108.

[9] D. W. Etherington, "Formalizing Non-monotonic Reasoning Systems," *Artificial Intelligence*, **31** (1987) , 41-85.

[10] S. E. Fahlman, *NETL: A System for Representing and Using Real-World Knowledge*, MIT Press, Cambridge, MA, 1979.

[11] A. M. Frisch, "A General Framework for Sorted Deduction: Fundamental Results on Hybrid Reasoning," *First Int'l Conf. on the Principles of Knowledge Representation and Reasoning* , Toronto, Ontario , May 1989.

[12] A. Goldberg and D. Robson, *Smalltalk-80: The Language and its Implementation*, Addison-Wesley, 1981.

[13] M. Hammer and D. McLeod, "Database Descriptions with SDM: A Semantic Database Model," *ACM Trans. on Database Systems*, **6** , 3 (Sep. 1981) , 351-386.

[14] R. Hull and R. King, "Semantic Database Modeling: Survey, Applications, Research Issues," *ACM Computing Surveys*, **19** , 3 (Sep. 1987) , 210-260.

[15] H. V. Jagadish, "Incorporating Hierarchy in a Relational Model of Data," AT&T Bell Laboratories Technical Memorandum, 1989.

[16] D. Maier, *The Theory of Relational Databases*, Computer Science Press, 1983.

[17] J. Mylopoulos and H. K. T. Wong, "Some Features of the Taxis Data Model," *Proc. 6th Int'l Conf. Very Large Data Bases* , Montreal, Canada , 1980.

[18] P. F Patel-Schneider, "Small can be Beautiful in Knowledge Representation," *Proc. IEEE Workshop on Principles of Knowledge-Based Systems* , Denver, Colorado , Dec. 1984.

[19] R. Reiter, "On Closed World Databases," in *Logic and Databases* , H. Gallaire, J. Minker, and J. Nicolas (ed.) , Plenum Press , 1978.

[20] L. A. Rowe and M. R. Stonebraker, "The POSTGRES Data Model," *Proc. 13th Int'l Conf. Very Large Data Bases* , Brighton, England , Sept. 1987, 83-96.

[21] E. Sandewall, "Non-monotonic Inference Rules for Inheritance with Exceptions," *Proceedings of the IEEE*, **74** (1986) , 1345-1353.

[22] Ulrich Schiel, "An Abstract Introduction to the Temporal Hierarchical Model," *Proc. 9th Int'l Conf. on Very Large Databases* , 1983, 322-330.

[23] J. M. Smith and D. C. P. Smith, "Database Abstractions: Aggregation and Generalisation," *ACM Trans. Database Syst.*, **2** , 2 (June 1977) , 105-133.

[24] G. A. Story and M. A. Jones, "Inheritance Reasoning with Preferences," AT&T Bell Laboratories Technical Memorandum, 1988

[25] B. Stroustrup, *The C++ Programming Language*, Addison-Wesley., 1986.

[26] D. S. Touretzky, *The Mathematics of Inheritance Systems*, Morgan Kaufmann, Aug. 1987.

[27] D. S. Touretzky, J. F. Horty, and R. H. Thomason, "A Clash of Intuitions: The Current State of Nonmonotonic Multiple Inheritance Systems," *Proc. 10th Int'l Joint Conf. on Artificial Intelligence* , Milan, Italy , 1987.

[28] C. Walther, *A Many-Sorted Logic Based on Resolution and Paramodulation*, Morgan Kaufman, Los Altos, CA, 1987.