Attribute-Oriented View Definitions in Relational and Deductive Databases

Inderpal Singh Mumick*
Savera Systems, mumick@savera.com
Kenneth A. Ross**
Columbia University, kar@cs.columbia.edu

Abstract. In some database applications it is natural to define individual attributes of a relation as derived attributes, separately from the definition of other attributes. This approach minimizes the complexity of the schema. We propose a formalism in which rules are divided into two classes: "Defining rules" specify the existence of tuples in the relation, together with some of their attributes. "Refining rules" specify derived attributes of tuples that already exist according to the defining rules. A potential problem with this approach is that view definitions may appear to be recursive, as some attributes of a relation are defined in terms of others. We show how this apparent recursion can be dealt with by identifying the attributes through which the recursion occurs. We present semantics based on these notions, and show how they allow the natural formulation of a motivating example. We discuss evaluation of rules with our semantics, demonstrating that goal-directed evaluation and incremental evaluation are possible, and that only rules defining derived attributes that are necessary for a query need to be evaluated in order to answer the query. Our proposal is directly relevant to the semantics of virtual columns in SQL3.

1 Introduction

It is common in object-oriented languages, such as C++, to define a class with some attributes stored in data fields, and other intensional attributes defined using member functions. Such intentional attributes are called *derived* attributes. Compared with solutions in which the only way to define derived data is via separate additional views, derived attributes offer several advantages: (a) Derived attributes can eliminate the need to copy all attributes of an existing table into a new table, thus minimizing the number of attributes that need to be repeated in different parts of the schema, and (b) Derived attributes can reduce the complexity of the schema that is visible to the user.

---

* This work was performed while the author was at AT&T Laboratories.

** The research of Kenneth Ross was supported by a grant from the AT&T Foundation, by a David and Lucile Packard Foundation Fellowship in Science and Engineering, by a Sloan Foundation Fellowship, by NSF grants IRI-9209029, CDA-90-24735, and by an NSF Young Investigator award.

In this paper we consider derived attributes in a relational (or deductive) data model. (We explain why object-oriented solutions do not apply in Section 6.) Derived attributes may be defined for both base tables and views. We present a syntax and semantics for a language including derived attributes, and describe query evaluation techniques for the language.

"Virtual columns" can be used to define derived attributes in SQL3 [ISO96]. Thus our approach is also relevant to the semantics of SQL3. Virtual columns can be legally defined in a way that introduces mutual recursion between tables with virtual columns. SQL3 does not yet define the semantics of such specifications. Our solution gives candidate semantics for recursive virtual columns in SQL3.

A naive incorporation of derived attributes into a relational/deductive language can lead to unintentional recursion, and a fixpoint semantics different from the intended meaning. We motivate the semantic problem with an example.

1.1 Problem Description

Consider a university information system consisting of three loosely-coupled databases. On one database system the Computer Science department manages its local information. On another database system, the Electrical Engineering department manages its local information. On the third database system, the School of Engineering manages schoolwide information. Each of the three database systems can interact with the others by referring to relations (base tables or views) that the other systems have declared public.

Several administrative units within the university require both the EE and the CS departments to make publicly available a relation denoting the advisor for each course affiliated with the department. An advisor for a course is the person that a student needs to consult in order to determine whether the student has the prerequisites for the course. The relation should state (at least) the course, the advisor's name, the course enrollment, and the advisor's office. The departments define relations *cs* and *ee* as complex views (*complex-f_1* and *complex-f_2*) over their internal databases:

$$\begin{array}{l} cs(\mathit{course}\ C, \mathit{advisor}\ A, \mathit{enrollment}\ E, \mathit{office}\ O) \coloneq \mathit{complex}\text{-}f_1(C, A, E, O) \\ ee(\mathit{course}\ C, \mathit{advisor}\ A, \mathit{enrollment}\ E, \mathit{office}\ O) \coloneq \mathit{complex}\text{-}f_2(C, A, E, O) \end{array}$$

complex-f_1 and *complex-f_2* are complex queries deriving the tuples in the *cs* and *ee* relations. We use a notation where each attribute of a relation is named. This notation will permit us to refer only to the attributes of interest.

The Dean of the Engineering School realizes that some courses are affiliated with both departments. For example, the Networks course is part of the major for CS (where it is called CS1234) and for EE (where it is called EE5678). The Dean makes available in the school database a relation $\mathit{equiv}(\mathit{cs}\ X, \mathit{ee}\ Y)$ of equivalent courses. *equiv*(*cs CS1234*, *ee EE5678*) is a record in this relation. *equiv* is a one-to-one partial relationship between courses in the two departments.

The Dean now makes a request of the two departments: In the event that the same course is affiliated with both departments, the respective advisor relations should refer to an alternate advisor, namely the advisor from the opposite department. For courses affiliated with just one department, the alternate advisor field should be null.

The schema described here has been slightly simplified by considering only two departments, with a separate relation for each. In reality, the school of engineering is likely to have multiple departments, and we would like to have one relation about the courses offered by all departments with department name as an attribute. The problem of defining alternate advisors and the solution we describe here would apply equally well to such a schema.

1.2 Schema Design Principles for Views

Before we proceed to discuss the potential solutions to the problem of defining an alternate advisor, we discuss what constitutes a "good" solution. Primarily, the solution must be *correct.* However, there are additional properties of a solution that from a software engineering viewpoint make it preferable.

Encapsulation: If a view V is to be defined in terms of a view W, then it is *good* to explicitly mention W in the definition of V; it is *bad* to unfold the definition of W into the definition of V. Suppose that as the database system evolves over time, the database administrator adapts the query defining W (but does not change the schema of W). If W is unfolded into V then V also needs to be modified. If V referred to W, then V does not need to be changed.

Conversely, if a relationship between two (or more) tables or views T_1 and T_2 is used frequently as a single "concept," as a subexpression in many queries and/or views, then it is *good* to define a view for this relationship, and use the view in queries. Defining a view once and using the view for querying relieves the users of redefining the relationship in every query, reducing the probability of error. Further, it can be easier to redefine the relationship.

Note that if a view V can be defined using a single subgoal U (or a proper subset of subgoals $\bar{U}$) in the definition of view W, then the view really depends on the subgoal U, and encapsulation does not mean that V should be defined using W, even if that is semantically correct. In other words, using a proper subset of subgoals defining W is not considered a violation of encapsulation.

Minimize Schema Redundancy: Redundancy between base tables and views, and between different views causes problems as the database schema evolves.

Consider, for example, a relation R with attributes a and b, and suppose view V, with attributes a, b and c is defined as the join of R with another relation S. For simplicity assume that the join is guaranteed to be one-to-one and that all tuples from R participate in the join. Suppose that both R and V are visible in the database schema. A user who is interested in the relationship between attributes a and b can reference either R or V. We might expect that some users will define new views in terms of R, others in terms of V. Now suppose that after some time the application for which view V was initially defined is no longer being used. Even though no user is interested in the join of R and S, we cannot remove view V because some users have used V to obtain the relationship between attributes a and b. Even worse, if relation S is deleted then many views are affected even though none of them actually uses the attribute c from S.

Thus we desire a mechanism that enforces the property that references to the unit of information one is interested in (in the example above it is the a/b relationship) are not affected by changes to extraneous information (the c attribute in the example). Further, we want that a particular unit of information is visible in as few places as possible (in the example above, the a/b relationship should be visible only once). Thus, we require that a relation or view such as $R(a, b)$ not be made visible if the relationship represented by the relation/view is obtainable from another visible relation/view (in this case, from $V(a, b, c)$).

We do not prohibit a relationship from appearing in multiple places in the schema. For example, we may want to associate the employees of a department with the department, and also to associate the department with each employee. However, we would discourage multiple representations of the same relationships if they could be separately modified leading to the problem outlined above.

Example 1. Consider a view $V(a\ X, b\ Y, c\ Z) :- R(a\ X, b\ Y), S(b\ Y, c\ Z)$. According to our schema redundancy principle, a user who needs only attributes a and b to define a new view, and who can (in a semantically correct way) choose between referring to R or referring to V, should use R instead of V. This is good because if the definition of V changes this user is not affected.

The encapsulation principle would have said the opposite: use the view rather than its definition in defining a new view. The encapsulation principle does not apply because we are using a proper subexpression of the definition of V to define our new view. If the user's choice was instead between referring to V and referring to the join of R and S, then the encapsulation principle would apply, and would advocate the use of V. □

1.3 An Attempted Solution

In response to the Dean's request, the two departments independently come up with the same clever "solution" to the problem, i.e., to adjust their definitions of *cs* and *ee* relations as follows:

$$
\begin{aligned}
r1 : &\ \mathit{cs}(\mathit{course}\ C, \mathit{advisor}\ A, \mathit{enrollment}\ E, \mathit{office}\ O, \mathit{alternate}\ X) :- \\
&\quad \mathit{complex}\text{-}f_1(C, A, E, O), \mathit{equiv}(\mathit{cs}\ C, \mathit{ee}\ C'), \mathit{ee}(\mathit{course}\ C', \mathit{advisor}\ X) \\
r2 : &\ \mathit{cs}(\mathit{course}\ C, \mathit{advisor}\ A, \mathit{enrollment}\ E, \mathit{office}\ O, \mathit{alternate}\ \mathit{null}) :- \\
&\quad \mathit{complex}\text{-}f_1(C, A, E, O), \neg \mathit{equiv}(\mathit{cs}\ C) \\
r3 : &\ \mathit{ee}(\mathit{course}\ C, \mathit{advisor}\ A, \mathit{enrollment}\ E, \mathit{office}\ O, \mathit{alternate}\ X) :- \\
&\quad \mathit{complex}\text{-}f_2(C, A, E, O), \mathit{equiv}(\mathit{cs}\ C', \mathit{ee}\ C), \mathit{cs}(\mathit{course}\ C', \mathit{advisor}\ X) \\
r4 : &\ \mathit{ee}(\mathit{course}\ C, \mathit{advisor}\ A, \mathit{enrollment}\ E, \mathit{office}\ O, \mathit{alternate}\ \mathit{null}) :- \\
&\quad \mathit{complex}\text{-}f_2(C, A, E, O), \neg \mathit{equiv}(\mathit{ee}\ C)
\end{aligned}
$$

Each department reasons that they can use the advisor relation from the other department in order to fill in the additional field in their own view. However, the net effect is that they have introduced *recursion*.

The semantics of the mutually recursive views is not what the departments might have expected. For courses that are not shared between the two departments, the views compute the correct tuples. However, the mutual recursion prevents all tuples for courses that are shared between the cs and ee departments from being computed into the views, according to the least fixpoint semantics.

The departments then considered several solutions to their problem. One was to unfold the definition of $complex\text{-}f_1$ from the CS department into the definition of *ee* within the EE database system. This solution did not satisfy the Dean (who was a professor of database design) because it violated encapsulation.

A second option considered was to make externally visible an additional layer of intermediate views *ics* and *iee* that included all but the alternate advisor attribute. The desired *cs* and *ee* views could then be specified in terms of these intermediate views. This solution also did not satisfy the Dean because it violated the principle of minimizing schema redundancy: it made visible the views *ics* and *iee* all of whose information was also contained in the *cs* and *ee* views.

A third alternative considered was to define a view containing only the alternate advisor and course number, separately from the main advisor views. However this approach did not satisfy the Dean because it also violated the principle of encapsulation. Many users required access to advisors and alternate advisors for courses. Such users would have to embed the join of the two relations into their own queries and view definitions, repeating the join at several places.

1.4 Outline of the Proposed Solution

In this section we describe a better solution to the problem. We wish to avoid the disadvantages mentioned above, particularly the schema redundancy and encapsulation problems. The solution we propose is based on the observation that the relations represent two different sorts of information about courses: (a) That a course exists, with values for some of its required attributes, and (b) that an existing course may have some of its derived attribute values defined intensionally. We thus separate the two roles of the view definition, essentially allowing the existence of a record in the relation to be specified separately from the values of some of its attributes. Despite this separation, references to tuples with derived attributes look exactly like references to tuples with stored attributes.

In a sense our solution cheats by extending the database language so that our problem is easy to solve. However, this extension is similar to virtual columns in SQL3 [ISO96], and the problems this extension presents, and our solution, apply to SQL3 as well. Here is our solution to the Dean's problem:

$$r5 : cs(course\ C, advisor\ A, enrollment\ E, office\ O) \text{ :- } complex\text{-}f_1(C, A, E, O)$$
$$r6 : cs(course\ C : alternate\ X) \text{ :- } equiv(cs\ C, ee\ C'), ee(course\ C', advisor\ X)$$
$$r7 : ee(course\ C, advisor\ A, enrollment\ E, office\ O) \text{ :- } complex\text{-}f_2(C, A, E, O)$$
$$r8 : ee(course\ C : alternate\ X) \text{ :- } equiv(cs\ C', ee\ C), cs(course\ C', advisor\ X)$$

There is a new syntactic element, namely the ":" in the heads of two of the rules.

For now we shall just give the intuition behind these rules. (We shall precisely define the semantics later, and also show how to evaluate such views.)

Rules with standard heads determine the *existence* of a tuple in the defined relation. We call these *defining rules*, and we call the attributes mentioned in the heads of defining rules *core attributes*. Rules with the new symbol ":" in the head are used to *refine* tuples in the relation that already exist. We call these *refining rules*, and attributes mentioned to the right of the ":" symbol are called *derived attributes*. The intuition for the rules above is that the existence of a course in *cs* is defined according to *complex-*f_1 (and symmetrically for *ee*), while the alternate-advisor field of a course *already in this relation* is defined according to the second rule. The rule head $cs(course\ C : alternate\ X)$ is interpreted as a *modifier* of an existing set of tuples. This rule head essentially says that any existing tuple with a *course* attribute C should be extended to have an *alternate* attribute X. We shall formalize this extension process later.

There are three possible cases we need to consider in the application of the refining rules. For a given existing tuple the refining rules may (a) generate no values for the derived attributes, (b) generate a single value for the derived attributes, and (c) generate multiple values for the derived attributes. We consider case (c) later, and focus on cases (a) and (b) here.

If there are no bindings for the derived attributes, then the tuple is unchanged; derived attributes for this tuple have no value.[3] If there is a single binding for the derived attributes, then the derived attributes get those values in the extended tuple.

The rules $r5 - r8$ given above are more concise than the previous proposed rules $r1 - r4$ that had to separate out the special case in which there were no equivalent courses, mentioning *complex-*f_1 more than once. However, it seems that our rules retain the mutual recursion since *cs* and *ee* are still defined in terms of one another. So why is our proposal a solution?

The reason that our proposal solves the problem is that the apparent recursion in our rules is not *real* recursion. The recursive subgoal in each refining rule only requires core attributes in the opposite rule. Because existence of tuples is defined separately from the value of certain attributes, one can project the defining rules onto the core attributes and still expect to get a correct answer. For example, we expect that $\pi_{course, advisor, enrollment, office}(cs) = complex\text{-}f_1$ independent of how the alternate advisors are defined. Using this observation, we can evaluate the rules above as if they were written as

$$
\begin{aligned}
&cs(course\ C, advisor\ A, enrollment\ E, office\ O) \text{ :- } complex\text{-}f_1(C, A, E, O)\\
&cs(course\ C : alternate\ X) \text{ :- } equiv(cs\ C, ee\ C'), complex\text{-}f_2(C', X, \_, \_)\\
&ee(course\ C, advisor\ A, enrollment\ E, office\ O) \text{ :- } complex\text{-}f_2(C, A, E, O)\\
&ee(course\ C : alternate\ X) \text{ :- } equiv(cs\ C', ee\ C), complex\text{-}f_1(C', X, \_, \_)
\end{aligned}
$$

with the effect of removing the recursion from the rules.

We claim that our solution also satisfies both schema design principles of Section 1.2. It respects encapsulation since we do not unfold the definitions of *cs*

[3] This is similar in some ways to having a "null" value.

or *ee*, nor do we require the user to repeat any join in the queries. It minimizes schema redundancy because no information is represented redundantly in multiple visible relations, and queries about advisors of a course are not affected by the existence (or nonexistence) of alternate advisors.

Paper Outline The rest of this paper is organized as follows. In Section 2 we describe our rule syntax, with separate notions of "defining rules" and "refining rules." In Section 3 we introduce two operations called $\circ$-composition and $\diamond$-composition, and show how these operations can be used to define two semantics (the $\circ$-semantics and the $\diamond$-semantics). Section 4 examines issues of rule evaluation, demonstrating that our rules can be evaluated using the same sorts of optimizations as standard rules. We compare our work with related work, including F-logic, in Section 6. Section 7 presents conclusions.

2 Extending the Rule Language

Definition Atom: Each predicate symbol has an associated set of unique attribute names. An *atom* is a predicate symbol applied to arguments consisting of a constant or a variable, labeled with an attribute name. Each attribute name in an atom q must be unique within q, and must also be associated with the predicate symbol of q; However, not all attribute names of q's predicate need to appear in q. The order of attribute names in an atom is immaterial. We assume that attribute names are drawn from a finite (but arbitrarily large) domain of attribute names. A *literal* is either an atom or a negated atom. □

Definition 1. (Defining Rule): We consider *defining rules*, of the form

$$h(\vec{X}) \coloneq b(\vec{X}, \vec{Y}).$$

h is the *head* of the rule, and is an atom with labeled variables $\vec{X}$. (A vector represents some number of variables.) The head may contain constants. b is the body of the rule, and is a conjunction of literals with the conjunction containing labeled variables $\vec{X}$ and $\vec{Y}$. □

Definition 2. (Modifier Atom): A *modifier* atom is an atom in which the attributes are divided into two groups. This is syntactically represented by separating the attributes with a ":" symbol. The attributes to the left of the ":" symbol are called the *previous* attributes, and attributes to the right of the ":" symbol are called the *derived* attributes. □

The intuition behind the separation of attributes in modifier atoms is as follows. A modifier atom does not get explicitly added to the set of tuples when the rule fires. Instead, the presently available set of tuples is examined for tuples that match on the previous attributes. For every such match we extend the tuple (in a fashion to be described later) using the values in the derived attributes.

Definition 3. (Refining Rule): A *refining rule* has the form

$$h(\vec{X}) :- b(\vec{X}, \vec{Y})$$

where the head h is a modifier atom rather than an ordinary atom. b is the body of the rule, and is a conjunction of literals with the conjunction containing labeled variables $\vec{X}$ and $\vec{Y}$. An attribute that appears as a derived attribute in the head of a refining rule is called a *derived attribute* of the rule. A view definition using refining rules will be called an attribute-oriented view definition. □

Definition 4. (Query): A *query* is a conjunction of literals. An *answer* to a query (given a semantics for the database) is a set of tuples of bindings to the variables in the query such that the query is satisfied (according to the semantics) if and only if the variable-bindings belong to the answer. □

3 Semantics

Rules with labeled attribute arguments can be mapped into a traditional unlabeled formalism by fixing the arity of predicates and mapping attribute names to argument positions. As long as head atoms mention every attribute in a rule, the semantics can be defined using standard least fixpoint techniques. (Attributes in body atoms that are not mentioned are replaced by dummy variables; in negated atoms the translation is only slightly more complex.) In this paper, however, head atoms may contain a partial subset of the attributes of a predicate, and use the special ":" construct. We must define a semantics for these rules.

3.1 Tuples, Modifiers and Composition

Tuples: A *tuple* for a predicate p is a variable-free atom, i.e., a set of attribute-value pairs where attribute names are unique in the set, values for an attribute are of the appropriate type, and all attribute names are associated with p. Note that a tuple does not necessarily need to mention *all* attributes of p. A tuple that mentions *no* attributes will be written as $p()$.

Subsumption: Let p be a predicate with attributes a, b, and c. A tuple $t_1 = p(a\ 1, b\ 2)$, can be extended to $t_2 = p(a\ 1, b\ 2, c\ 3)$ by deriving the c attribute. In this case we say that tuple t_2 *subsumes* tuple t_1.

We shall use a special value *mult* below. The intuition is that *mult* represents an overdefined attribute value. In a multi-valued logic this notion is similar to a local contradiction. Thus we expect $p(a\ 1, b\ 2)$ to be *subsumed* by $p(a\ 1, b\ \mathit{mult})$.

Formally, A tuple t_2 *subsumes* a tuple t_1, written as $t_1 \sqsubseteq t_2$, if for every attribute value pair $(a\ v)$ in t_1 either $(a\ v)$ is in t_2 or $(a\ \mathit{mult})$ is in t_2. A set T_2 of tuples *subsumes* a set T_1 of tuples (written $T_1 \sqsubseteq T_2$) if for each $t_1 \in T_1$ there exists $t_2 \in T_2$ such that t_2 subsumes t_1.

If T is a set of tuples, we define the *canonical version* of T, written $\mathcal{C}(T)$, to be the set of tuples formed by excluding from T all subsumed tuples. (The exclusion can be done in any order.) We define $\bigsqcup_{i \in I}(T_i)$ to be $\mathcal{C}(\bigcup_{i \in I}(T_i))$.

Modifiers: A *modifier* for a predicate is a variable-free modifier atom involving that predicate. We say that a modifier m *is compatible with* a tuple t if all previous attributes of m are defined in t and have, either the same value in tuple t as in modifier m, or have the value *mult* in tuple t. The *union* of a tuple t and a modifier m, written $t \cup m$ is defined as the union of the set of attribute-value pairs in tuple t with the set of attribute-value pairs in the derived attributes of m. $t \cup m$ yields a tuple, unless a derived attribute name of m already exists in tuple t with a different value. We now define two kinds of compositions, *circle composition* ($\circ$-composition), and *diamond composition* ($\diamond$-composition). Composition of a modifier with a tuple yields a set of tuples.

Definition 5. ($\circ$-Composition of modifier and tuple): If modifier m is compatible with a tuple t, and if $t \cup m$ is a tuple, then we define $m \circ t = \{t, t \cup m\}$. Otherwise, $m \circ t$ is defined to be equal to $\{t\}$. $\Box$

Definition 6. ($\diamond$-Composition of modifier and tuple): If modifier m is compatible with a tuple t, then $m \diamond t$ is defined to be $\{t \cup m\}$, except that all duplicated attribute-value pairs of the form $\{(a\ v_1), (a\ v_2)\}$ in $t \cup m$ are replaced by a single attribute value pair (a *mult*). If modifier m is not compatible with tuple t, then $m \diamond t$ is defined to be equal to $\{t\}$. $\Box$

Note that the circle composition always *retains* the old tuple, and *adds* a new tuple if one can be formed by composition. The diamond composition *replaces* the old tuple if a new tuple can be formed by composition. We now generalize the composition to apply a set of modifiers to a set of tuples.

Definition 7. ($\circ$-Composition of modifier set and tuple set) Let M be a set of modifiers and T a set of tuples. For each $m \in M$, define: $m \circ T = \bigcup_{t \in T}(m \circ t)$. We define $M \circ T$ as the smallest set U of tuples that contains T and has the property that for all $m \in M$, $m \circ U = U$. $\Box$

Definition 8. ($\diamond$-Composition of modifier set and tuple set) Let M be a set of modifiers and T a set of tuples. For each $m \in M$ define: $m \diamond T = \bigsqcup_{t \in T}(m \diamond t)$. We define $M \diamond T$ as the smallest set (under the subsumption ordering $\sqsubseteq$) U of tuples that subsumes T and has the property that for all $m \in M$, $m \diamond U = U$. $\Box$

The intuition behind the two composition operations is that they both *extend* a set of tuples, but in different ways. The $\circ$-composition operation *adds* to a set of tuples T those tuples formed by adding some attribute-value pairs to tuples in T according to M. The $\diamond$-composition operation *replaces* tuples in T with tuples formed by adding some attribute-value pairs according to M; if there are conflicting attribute-value pairs, then the special value *mult* is used.

Example 2. **Compositions:** Let t be the tuple $p(a\ 1, b\ 2, c\ 3)$. The modifier $m_1 = (a\ 1 : d\ 4)$ is compatible with t, and $m_1 \circ t = m_1 \diamond t = \{p(a\ 1, b\ 2, c\ 3, d\ 4)\}$. The modifier $m_2 = (b\ 4 : e\ 5)$ is not compatible with t, and so $m_2 \circ t = m_2 \diamond t = \{t\}$. The modifier $m_3 = (d\ 4 : e\ 5)$ is also not compatible with t, and so

$m_3 \circ t = m_3 \diamond t = \{t\}$. However $m_3 \circ m_1 \circ t = \{p(a\ 1, b\ 2, c\ 3, d\ 4, e\ 5)\}$, and similarly for $m_3 \diamond m_1 \diamond t$. The modifier $m_4 = (a\ 1 : c\ 5)$ is compatible with t, although $m_4 \circ t = \{t\}$ due to the multiple values for attribute c. $m_4 \diamond t$ is equal to $\{p(a\ 1, b\ 2, c\ \mathit{mult})\}$. The modifier $m_5 = (c\ 7 : d\ 11)$ is compatible with $p(a\ 1, b\ 2, c\ \mathit{mult})$, so $m_5 \diamond p(a\ 1, b\ 2, c\ \mathit{mult}) = \{p(a\ 1, b\ 2, c\ \mathit{mult}, d\ 11)\}$.

$m_5 \diamond m_4 \diamond t = p(a\ 1, b\ 2, c\ \mathit{mult}, d\ 11)$; note that neither of the c values (in m_4 and t) that caused the *mult* value was actually 7, yet m_5 still applies to give a value for attribute d. This last example demonstrates that one should be suspicious of attributes of tuples defined in terms of an attribute with a *mult* value: *mult* indicates a form of *error*. □

We let $\mathcal{T}$ denote the domain of sets of tuples, and let $\mathcal{C}(\mathcal{T})$ denote the domain of sets of tuples in which no tuple is subsumed by another tuple.

Lemma 9. *$(\mathcal{T}, \subseteq)$ is a complete lattice with $\bigcup$ as its least upper bound operation. Let $F_M(T) = \bigcup_{m \in M} m \circ T$. Then, (1) $F_M(T)$ is monotonic on $(\mathcal{T}, \subseteq)$, and (2) $T \subseteq F_M(T)$ for all $T \in \mathcal{T}$.*

$(\mathcal{C}(\mathcal{T}), \sqsubseteq)$ is a complete lattice with $\bigsqcup$ as its least upper bound operation. Let $G_M(T) = \bigsqcup_{m \in M} m \diamond T$. Then, (1) $G_M(T)$ is monotonic on $(\mathcal{C}(\mathcal{T}), \sqsubseteq)$, and (2) $T \sqsubseteq G_M(T)$ for all $T \in \mathcal{C}(\mathcal{T})$). ▮

Lemma 10. *Given any modifier set M and tuple set T, there exists a unique value of $M \circ T$. Given any modifier set M and tuple set $T \in \mathcal{C}(\mathcal{T})$), there exists a unique value of $M \diamond T$.* ▮

Interpretations: An *interpretation* for a set of predicates is a set of tuples using those predicates. (In this paper the set of predicates used will be understood from the context, so we shall not explicitly mention the set of predicates.) A tuple t is *satisfied* in an interpretation I, that is, $I \models t$, if there is a tuple t in I that subsumes t. One can define satisfaction for a conjunction of atoms (or negated atoms) in a straightforward way. A *canonical interpretation*, $\mathcal{C}(I)$ is an interpretation I in which no tuple in I is subsumed by another tuple in I. Every interpretation I can be converted to a canonical interpretation $\mathcal{C}(I)$ such that for any tuple t, $I \models t$ iff $\mathcal{C}(I) \models t$.

3.2 Diamond and Circle Semantics

We define two kinds of semantics that differ in the way they treat incompatible extensions of a tuple. Two extensions to a tuple are incompatible if they assign different values to a derived attribute. Our choices for what to do in case of multiple definitions of an attribute are limited. We cannot choose an arbitrary value from among the competing values without introducing nondeterminism. We do not want to permit a *set* of values for some attribute(s) within a single tuple because that would change the type of those attributes in addition to violating first normal form. We cannot simply omit such attributes altogether because then there is no distinction between no-value and many-values. Our first semantics, the diamond semantics, chooses to represent the many-values case using the constant *mult*. The second semantics, the circle semantics, chooses to represent incompatible extensions in separate tuples.

Diamond ($\diamond$) Semantics: One semantics, called the diamond semantics, is appropriate when we assume that a tuple t cannot be extended in two incompatible ways leading to multiple occurrences of t within two different tuples in the result. Such a semantics is appropriate when we expect derived attributes to be functionally determined by the previously defined attributes in the view. A consequence of this approach is that tuples are extended in only one way, using the $\diamond$-composition operation. The cardinality of a set of tuples does not change when $\diamond$-composed with a set of modifiers if the defining rules all contain a key attribute.[4] This property may be important when the multiplicity of tuples needs to be controlled. In the event of there being two incompatible extensions of a tuple, a situation usually indicating an error, the attribute value *mult* is used.

Circle ($\circ$) Semantics: The second semantics is appropriate when we assume that a tuple p *can* be extended in two incompatible ways leading to multiple occurrences of p within two different tuples in the result. With this approach, tuples are potentially extended in multiple ways, using the $\circ$-composition operation. The cardinality of a set of tuples may increase when $\circ$-composed with a set of modifiers. This property may be important when the multiplicity of tuples is used to indicate a semantically meaningful property, as illustrated below:

Example 3. Suppose that we wish to add multiple *sections* for a single course if the enrollment is sufficiently high. The *cs* department policy is to add a second section when the enrollment exceeds sixty students. Using the circle semantics we could add the following rules

$$\begin{array}{l} cs(course\ C : section\ 1) \\ cs(course\ C, enrollment\ E : section\ 2) :\!-\ E > 60 \end{array}$$

If our base data was

$$\begin{array}{l} cs(course\ CS1234, advisor\ Smith, enrollment\ 75, office\ Rm510) \\ cs(course\ CS2222, advisor\ Jones, enrollment\ 45, office\ Rm515) \end{array}$$

then the circle semantics of the result would be (canonically)

$$\begin{array}{l} cs(course\ CS1234, advisor\ Smith, enrollment\ 75, office\ Rm510, section\ 1) \\ cs(course\ CS1234, advisor\ Smith, enrollment\ 75, office\ Rm510, section\ 2) \\ cs(course\ CS2222, advisor\ Jones, enrollment\ 45, office\ Rm515, section\ 1) \end{array}$$

We could then further refine each section with the corresponding *instructor*, *meeting-time*, etc. using refining rules. The benefit of structuring the database in this way is that we have a single uniform representation of courses together with their sections. We can ask queries about courses without mentioning section

[4] The cardinality may *decrease*, as in $\{p(a\ 1 : b\ 2), p(b\ 2 : a\ 1)\} \diamond \{p(a\ 1), p(b\ 2)\}$.

numbers, and would expect to get the same answer as if we simply queried the base data. If we *do* mention the section numbers, then we have access to both the section information and the course information without additional joins. We enjoy this benefit without having to deal with explicit nesting and unnesting.

For now assume that defining rules do not mention derived attributes in their subgoals, and that refining rules do not have negation. These conditions can be relaxed, as discussed in the full version of this paper.

Definition 11. (Fixpoint Diamond and Circle Semantics) Consider a program P where the defining rules do not mention derived attributes in their subgoals, and where refining rules do not have negation. The semantics of the program P is determined according to the following sequence of steps:

1. Use the least fixpoint of the defining rules to determine an initial set of tuples T_0. These may be considered the tuples that "exist" with some attributes of these existing tuples yet to be defined.
2. Let T_i be the current interpretation. (Initially $i = 0$.) Let T_A be the immediate consequence operation [Llo87] on the refining rules. T_A maps a set of tuples T to a set of modifiers M. Let $M_i = T_A(T_i)$ be the set of modifiers by applying the refining rules to the current interpretation.
3. Let $T_{n+1} = M_n \diamond T_n$ (or $T_{n+1} = M_n \circ T_n$, depending on the desired semantics) be the interpretation obtained by composing modifiers M_n with tuples T_n. If $T_{n+1} = T_n$, then stop; T_n represents the semantics of the program. If $T_{n+1} \neq T_n$, then repeat the previous step (generating a new set of modifiers M_{n+1}) and the present step (generating T_{n+2}), until a fixpoint is reached. □

Lemma 12. *Definition 11 assigns a unique finite semantics to a program P.* ▮

Example 4. **(Diamond and Circle Semantics)** Consider the rules

$$
\begin{array}{l}
cs(course\ C, advisor\ A, enrollment\ E, office\ O) \coloneq complex\text{-}f_1(C, A, E, O) \\
cs(course\ C : alternate\ X) \coloneq equiv(cs\ C, ee\ C'), ee(course\ C', advisor\ X) \\
ee(course\ C, advisor\ A, enrollment\ E, office\ O) \coloneq complex\text{-}f_2(C, A, E, O) \\
ee(course\ C : alternate\ X) \coloneq equiv(cs\ C', ee\ C), cs(course\ C', advisor\ X) \\
equiv(cs\ CS1234, ee\ EE5678)
\end{array}
$$

with the following extension for the complex view definitions:

$$
\begin{array}{l}
complex\text{-}f_1(CS1234, Smith, 75, Rm510),\ \ complex\text{-}f_1(CS2222, Jones, 45, Rm515) \\
complex\text{-}f_2(EE5678, Brown, 75, Rm654),\ \ complex\text{-}f_2(EE7777, Black, 55, Rm612)
\end{array}
$$

Our initial set T_0 of tuples is

$$
\begin{array}{l}
cs(course\ CS1234, advisor\ Smith, enrollment\ 75, office\ Rm510) \\
cs(course\ CS2222, advisor\ Jones, enrollment\ 45, office\ Rm515) \\
ee(course\ EE5678, advisor\ Brown, enrollment\ 75, office\ Rm654) \\
ee(course\ EE7777, advisor\ Black, enrollment\ 55, office\ Rm612)
\end{array}
$$

Applying T_A to T_0 gives M equal to

$$\{cs(course\ CS1234 : alternate\ Brown), ee(course\ EE5678 : alternate\ Smith)\}$$

$T_1 = M \diamond T_0$ is then given by

$cs(course\ CS1234, advisor\ Smith, enrollment\ 75, office\ Rm510, alternate\ Brown)$
$cs(course\ CS2222, advisor\ Jones, enrollment\ 45, office\ Rm515)$
$ee(course\ EE5678, advisor\ Brown, enrollment\ 75, office\ Rm654, alternate\ Smith)$
$ee(course\ EE7777, advisor\ Black, enrollment\ 55, office\ Rm612)$

In $M \circ T_0$, $cs(course\ CS1234, advisor\ Smith, enrollment\ 75, office\ Rm510)$ and $ee(course\ EE5678, advisor\ Brown, enrollment\ 75, office\ Rm654)$ both remain. T_A applied to T_1 gives M again and $M \diamond T_1 = T_1$, and so we have reached a fixpoint. Note that even though $M \diamond T_0$ and $M \circ T_0$ are not identical they are equivalent; one is the canonical version of the other.

To see the difference between the two semantics, imagine that an additional (probably erroneous) tuple $equiv(cs\ CS1234, ee\ EE7777)$ had been included in the database. $M \diamond T_0$ is then given by

$cs(course\ CS1234, advisor\ Smith, enrollment\ 75, office\ Rm510, alternate\ mult)$
$cs(course\ CS2222, advisor\ Jones, enrollment\ 45, office\ Rm515)$
$ee(course\ EE5678, advisor\ Brown, enrollment\ 75, office\ Rm654, alternate\ Smith)$
$ee(course\ EE7777, advisor\ Black, enrollment\ 55, office\ Rm612, alternate\ Smith)$

while $M \circ T_0$ is

$cs(course\ CS1234, advisor\ Smith, enrollment\ 75, office\ Rm510)$
$cs(course\ CS1234, advisor\ Smith, enrollment\ 75, office\ Rm510, alternate\ Brown)$
$cs(course\ CS1234, advisor\ Smith, enrollment\ 75, office\ Rm510, alternate\ Black)$
$cs(course\ CS2222, advisor\ Jones, enrollment\ 45, office\ Rm515)$
$ee(course\ EE5678, advisor\ Brown, enrollment\ 75, office\ Rm654)$
$ee(course\ EE5678, advisor\ Brown, enrollment\ 75, office\ Rm654, alternate\ Smith)$
$ee(course\ EE7777, advisor\ Black, enrollment\ 55, office\ Rm612)$
$ee(course\ EE7777, advisor\ Black, enrollment\ 55, office\ Rm612, alternate\ Smith)$ □

4 Evaluation

Our semantics can be obtained by the iterated construction described in the previous section. However, that construction can be simplified by observing the way attributes are defined in terms of other attributes. For example, suppose two attributes are defined independently from a common subset of attributes. If $p.a$ and $p.b$ are independently derived from $p.c$, and our query asks for just $p.a$, then we need only evaluate the rules for $p.a$ and not those for $p.b$.

The importance of these observations is that defining a large number of derived attributes for a predicate does not need to hurt the performance of queries that do not reference these derived attributes. We can perform query evaluation using just the rules that are relevant for the query.

We formalize this intuition using the notion of an evaluation graph. For simplicity, in this section we do not allow defining rules to mention derived-attribute subgoals in their bodies and we do not permit negation in the refining rules (as in Section 3). The extensions of the concepts in this section to handle such rules are straightforward.

Definition 13. (Evaluation Graph) Let P be a program, and let P_A be the refining rules of P, which are assumed to be negation-free. The evaluation graph of P, denoted $E(P)$ is a labeled directed graph defined as follows. The nodes in the graph are of the form "predicate.attribute". There is an edge from $p.a$ to $q.b$ if one of the following holds: Either (a) Some rule r in P_A has head predicate p, derived attribute a in the head, and a subgoal in the body with predicate q that mentions attribute b, or (b) $p = q$, and some rule r in P_A has head predicate p with derived attribute a in the head and previous attribute b in the head. In either case we label the specified edge with the rule r.

A node of the evaluation graph is called *basic* if the corresponding attribute of the given predicate is mentioned in the head of a defining rule. A node of the evaluation graph is called a *query node* with respect to a query Q if Q mentions the predicate and attribute of the node. The *query subgraph* $Q(P)$ of P with respect to a query Q is the set of nodes reachable from query nodes in $E(P)$ together with the edges between them. □

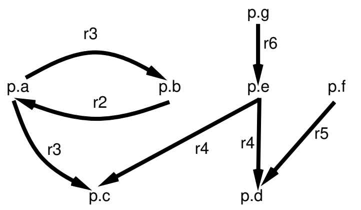
Definition 14. (Founded nodes and rules) Let P be a program, and G be a subgraph of $E(P)$. The sets of *founded nodes* and *founded rules* in G are defined as the minimal sets of nodes and rules that satisfy the following conditions: (a) Every basic node in G is a founded node. (b) If every edge in G labeled with rule r has a founded destination node in G, then r is a founded rule. (c) Every node in G that is the source of a founded rule is a founded node. □

The intuition behind founded nodes is that there is some potential way to instantiate the given attribute by firing a rule on previously instantiated tuples. Nodes that are not founded correspond to attributes that cannot be instantiated since there is no way to initiate a derivation.

Lemma 15. *Any node that is not founded in $E(P)$ represents a* useless *attribute, i.e., an attribute that will never be given a value according to our semantics.* ▮

Example 5. The program on the left below has the evaluation graph on the right.

$r_1 : p(c\ 3, d\ 4)$
$r_2 : p(a\ 1 : b\ 2)$
$r_3 : p(b\ 1 : a\ 2) :\!-\ p(c\ 3)$
$r_4 : p(d\ 4 : e\ 1) :\!-\ p(c\ 3)$
$r_5 : p(d\ 4 : f\ 1)$
$r_6 : p(e\ 1 : g\ 1)$



$p.c$ and $p.d$ are basic nodes and hence founded. r_4 and r_5 are thus founded rules, and $p.e$ and $p.f$ are founded nodes. It follows then that r_6 and $p.g$ are founded. r_3 and r_2 are not founded. $p.a$ and $p.b$ are also not founded.

r_2 defines $p.b$ in terms of $p.a$, while r_3 defines $p.a$ in terms of $p.b$. Without other rules defining $p.a$ or $p.b$, neither $p.a$ nor $p.b$ can be instantiated. □

Lemma 16. *In order to answer a query Q it suffices to evaluate only founded refining rules in $Q(P)$ in addition to the defining rules in P.* ▮

Example 6. Consider the program of Example 5. To answer the query ?-$p(f\ X)$ one needs evaluate only the subprogram consisting of r_1 and r_5. To answer the query ?-$p(g\ X)$ one needs evaluate only the subprogram consisting of r_1, r_4 and r_6. To answer the query ?-$p(e\ X), p(f\ X)$ one needs evaluate only the subprogram consisting of r_1, r_4 and r_5. □

5 Query Optimization Techniques

In this section we describe three sorts of query optimization techniques. Further query optimization techniques are presented in the full version of this paper.

Rule Splitting: It may be possible to narrow a rule set beyond the techniques of Section 4 by splitting rules. For example, consider a rule r of the form

$$p(a\ X : b\ Y, c\ Z) :- f(X, Y, Z)$$

where f is the body of the rule. If our query was ?-$p(b\ X)$ and r was founded then r would be needed in order to answer the query. However, it seems that r might be doing more work than necessary, defining the c attribute in addition to the b attribute. Can we split r into two rules

$$\begin{array}{l} r_1 : p(a\ X : b\ Y) :- f(X, Y, Z) \\ r_2 : p(a\ X : c\ Z) :- f(X, Y, Z) \end{array}$$

so that our evaluation needs just r_1 for this query and not r_2? The answer to this question is given in Lemma 18 below.

Definition 17. A rule r with n derived attributes is *split* by replacing it with n rules that are identical to r except that each has a single distinct derived attribute from r. □

Lemma 18. *Rule splitting is sound with respect to the $\diamond$-semantics, but unsound with respect to the $\circ$-semantics.* ▮

Semi-Naive Evaluation: One can extend "semi-naive evaluation" [Ull89] to our context in a straightforward fashion. One can think of incremental (delta) modifier sets in addition to incremental tuple sets. A new modifier needs to be applied to all corresponding previous tuples, and a new tuple needs to be matched with all corresponding previous modifiers. The differential behavior is also applied within the computation of $M \diamond T$ or $M \circ T$. If standard rules have derived-attribute subgoals then the defining rules and refining rules may be evaluated jointly in a differential fashion.

The only interesting point to consider is when a tuple is considered "new." Consider a subgoal of the form $p(a\ X, b\ Y)$. If an old tuple $p(a\ 1)$ is extended to become $p(a\ 1, b\ 2)$, then that extension is relevant for the rule having the subgoal, and so this extension yields a "new" tuple for the rule. However, if $p(a\ 1, b\ 2)$ were further extended to become $p(a\ 1, b\ 2, c\ 3)$ then that extension is *not* relevant for the rule because the subgoal does not refer to the c attribute.

Thus, a tuple is new for a rule at the moment that it gets values for all attributes mentioned in the rule, as long as those values haven't been seen before. Implementing this notion of "new" requires some extra bookkeeping since different rules require different delta sets.

Goal-Directed Behavior: Magic-sets techniques [BR91, Ram91] also can be extended in a straightforward fashion, thinking of the refining rules as if they were ordinary rules. Magic rules/subgoals can restrict the attributes of modifier atoms just as they restrict the attributes of standard atoms for a given query. There is one minor wrinkle, however, for the $\diamond$-semantics when predicates containing derived attributes are used to define other views.

Consider the query ?-$v(b\ X)$ on the following set of rules:

$$\begin{array}{l} q(b\ 2),\ q(b\ 3),\ p(a\ 1),\ t(b\ 4) \\ p(a\ 1 : b\ X) \coloneq q(b\ X) \\ v(b\ X) \coloneq t(b\ X), p(a\ 1, b\ X) \end{array}$$

According to the $\diamond$-semantics, the extension of p should be $p(a\ 1, b\ \mathit{mult})$, and so the answer to the query should be $v(b\ 4)$ since *mult* subsumes every value. A magic sets transformation (after some rule unfolding) results in

$$\begin{array}{l} q(b\ 2),\ q(b\ 3),\ p(a\ 1),\ t(b\ 4) \\ p(a\ 1 : b\ X) \coloneq t(b\ X), q(b\ X) \\ v(b\ X) \coloneq t(b\ X), p(a\ 1, b\ X) \end{array}$$

The problem is that no tuples of the form $p(a\ 1, b\ X)$ are now generated because relations q and t do not overlap. The $\diamond$-semantics of these rewritten rules would make v empty. The solution to the problem above is to avoid sideways information passing strategies that bind derived attributes of refining rules.

6 Related Work

We have chosen to express our rules by naming attributes rather than by relying on the positions of arguments to determine the corresponding attribute. This choice does not preclude our semantics from being applied to relations using argument positions rather than attribute names. Using attribute names is more convenient here because we can avoid having to write many "null" constants and dummy variables in atoms.

Virtual columns can be used to define derived attributes in SQL3 [ISO96], and can be legally defined in a way that introduces mutual recursion between tables with virtual columns. However, it is difficult to write SQL3 code for virtual columns in a declarative way. We could not express our example declaratively in SQL3 using virtual columns with the desired semantics. (See the full version of the paper for our best attempt.) The problems introduced by having recursion between attributes have not previously been identified in the SQL standard [Mat97]. We advocate our semantics for recursive virtual columns in SQL3 in order to be able to express such examples with their intended meaning.

Object-Oriented programming languages (such as C++ [Str91]) and object-oriented databases (such as Ode [AG89]) model structured collections as classes, and have member functions of each class that can be used to implement derived attributes (and much more). No *declarative* semantics has been proposed in C++, Ode, and other general object-oriented systems.

However, logic-based object-oriented proposals, such as O-Logic [Mai86] and F-logic [KLW95] have studied the notion of derived attributes and the problem of recursion between the derived attributes. O-Logic and F-Logic require unique object-identifiers, both for base objects and for derived objects. Creation of identifiers for derived "objects" can be difficult in a relational system when multiple tuples need to be collapsed into one tuple, or when multiple tuples can be derived from a single tuple based upon the values of the single tuple. One could imagine mapping the O-logic and F-logic semantics onto a relational system by introducing the notion of tuple identifiers in defining the semantics. Besides the fact that tuple identifiers are not necessary for defining relational semantics, one can show that such a mapping does not yield an equivalent semantics due to the basic difference between the value-based and object-based nature of the two domains. (See the full version of this paper.)

Other technical differences between F-Logic and our semantics include: (1) In F-Logic there is no distinction between rules that define the *existence* of an object/tuple, and rules that define an attribute of that object/tuple: all objects are required to have values for all derived attributes. In contrast, we distinguish between core and derived attributes, and do not require derived attributes to be defined for all tuples. (2) In F-Logic, a derivation of two different values for a single-valued attribute implies the equality of those two values. If one includes "freeness axioms" within the F-Logic system (to prevent such equalities from happening) then the resulting rules have no semantics when two different values are generated for a single object. In contrast, both of our circle and diamond approaches always give a semantics when conflicting attribute values are generated. Our use of the *mult* constant allows us to locally contain an erroneous duplicate value rather than denying a semantics for the whole program.

7 Conclusions

It is often natural to use attribute-oriented view definitions, as we have shown in an extended example. Compared with a formalism requiring derived data to be defined in views separate from the stored data, the complexity and redundancy of the schema apparent to a user is reduced. While derived attributes have been studied in the object-oriented literature, there has been little work on derived attributes in the relational model, and object-oriented solutions do not directly apply in a relational context.

Attribute-oriented view definitions often appear to be recursive simply because some attributes of one relation are defined using other attributes. Using standard least fixpoint semantics, this recursion leads to an unnatural semantics. We have shown that this apparent recursion can be dealt with by separating the rules into two classes. "Defining rules" specify the existence of tuples in a relation, together with some of their attributes. "Refining rules" specify derived attribute values of tuples that already exist according to the defining rules. We presented two semantics based on these notions, and showed how they allow the natural formulation of our motivating example. We discussed evaluation techniques for rules with our semantics, demonstrating that goal-directed evaluation and incremental evaluation are possible, and that only rules defining derived attributes that are necessary for a query need to be evaluated in order to answer the query.

The need for a clear understanding of derived attributes is made more important by the inclusion of virtual columns in SQL3 [ISO96]. This paper represents the first study of semantic problems with virtual columns of the type that can appear in SQL3. The diamond semantics is a candidate SQL3 semantics.

Acknowledgements

We thank Nelson Mattos for his explanation of virtual columns in SQL3.

References

[AG89] Rakesh Agrawal and Narain Gehani. Ode (object database and environment): the language and the data model. In *Proc. SIGMOD*, pages 36–45, May 1989.

[BR91] C. Beeri and R. Ramakrishnan. On the power of magic. *Journal of Logic Programming*, 10:255–300, 1991.

[ISO96] ISO_ANSI. ISO-ANSI working draft: Database language SQL3, 1996.

[KLW95] M. Kifer, G. Lausen, and J. Wu. Logical foundations of object-oriented and frame-based languages. *J.ACM*, pages 741–843, 1995.

[Llo87] J. W. Lloyd. *Foundations of Logic Programming*. Springer-Verlag, New York, 2nd edition, 1987.

[Mai86] D. Maier. A logic for objects. In J. Minker, editor, *Workshop on Foundations of Deductive Databases and Logic Programming*, pages 6–26, August 1986.

[Mat97] Nelson Mattos. Personal communication on virtual columns in SQL3, 1997.

[Ram91] R. Ramakrishnan. Magic templates: A spellbinding approach to logic programs. *Journal of Logic Programming*, 11:189–216, 1991.

[Str91] Bjarne Stroustrup. *The C + + Programming Language (Second Edition)*. Addison-Wesley, Reading, Massachusetts, USA, 1991.

[Ull89] J. D. Ullman. *Principles of Database and Knowledge Base Systems*. Computer Science Press, Rockville, MD, 1989. (Two volumes).