The Logical Data Model*

Gabriel M. Kuper
IBM T. J. Watson Research Center
Yorktown Heights, New York

Moshe Y. Vardi
IBM Almaden Research Center
San Jose, California

September 4, 1997

Abstract

We propose an object-oriented data model that generalizes the relational, hierarchical, and network models. A database scheme in this model is a directed graph, whose leaves represent data and whose internal nodes represent connections among the data. Instances are constructed from objects, which have separate names and values. We define a logic for the model, and describe a nonprocedural query language that is based on the logic. We also describe an algebraic query language and show that it is equivalent to the logical language.

*A preliminary version of this paper, under the title "A new approach to database logic", appeared in *Proc. 3rd ACM Symp. on Principles of Database Systems*, Waterloo, April 1984, pp. 86-96. For a more extensive coverage of the material presented here the reader is referred to the first author's Ph.D. dissertation *The logical data model: a new approach to database logic*, Dept. of Computer Science, Stanford University, 1985.

1. Introduction

Research in database theory during the 1970's and the early 1980's has focused mainly on the relational model [Cod70], due to its elegance and mathematical simplicity. This very simplicity, however, has gradually been recognized as one of the major disadvantages of the relational model: it forces the stored data to have a flat structure that real data does not always have [SS77, Cod79]. This has motivated a great deal of research during the past decade on *structured* data models: the so-called *semantic data models* (cf. [**?**, **?**]), *nested relations* (cf. [**?**, **?**]), and *complex objects* (cf. [**?**]). The reader is referred to [**?**, **?**, **?**] for excellent surveys.

Two works that we found in particular inspiring are by Jacobs [Jac79, Jac82] and by Hull and Yap [HY82]. Jacobs describes "database logic," a mathematical model for databases that claims to generalize all three principal data models. Hull and Yap [HY82] describe the "format model," which generalizes the relational and hierarchical models. In the format model, database schemes are viewed as trees, in which each leaf represents data, and each internal node represents some connection among the data.

Both these models are unsatisfactory in their ability to restructure data, i.e., the ability to query the database. While Hull and Yap ignore the issue of a data manipulation language, Jacobs' treatment is an overkill—his query language enables one to write noncomputable queries [Var83].

Furthermore, both approaches fail to model a significant aspect of hierarchical and network database management systems, which is the ability to use *virtual records*. Virtual records are essentially pointers to physical records, and they are used to avoid redundancy in the database [Ull82]. Note that virtual records introduce cyclicity not only in the schema level but also at the instance level.

In the model we propose here, a database scheme is an arbitrary directed graph. As in the format model, leaves (i.e., nodes with no outgoing edges) represent data, and internal nodes represent connections among the data. While it is not hard to model cyclicity at the schema level, it is not quite apparent how to do it at the instance level, without running into cyclic definitions. Our solution is to distinguish between object names and object values, or, equivalently, between the address space and the data space. This distinction goes back to Codd's notions of *surrogates* [Cod79]. Thus, instances in our model consist of objects, which have separate names and values. This enables us to give semantics to instances in a well-defined way.

A data model consists of several components [TL82]. The first is the database structure mentioned above, which describes the static portion of the database. The second component is a way to specify integrity constraints on the database, that restrict the allowed instances of the schema. We shall describe a logic in which integrity constraints can be specified. Our logic is inspired by Jacob's *database logic* [Jac82], but unlike database logic, our logic is effective. That is, given a database and a sentence in the logic, one can test effectively whether the sentence is true in the database or not.

The third component is a way to restructure data, in order to describe user views, queries, etc. We describe two such mechanisms, a logical, i.e., nonprocedural, query language and an algebraic, i.e., procedural, query language that are analogous to Codd's relational calculus and relational algebra, and we prove that these two languages are equivalent. These languages have a novel feature: not only can they access a nonflat data structure, e.g., a hierarchy, but the answers they produce do not have to be flat either. Thus, the language really does have the ability to restructure data and not only to retrieve it, as opposed to the other approaches cited above.

2. Introduction to the Logical Data Model

The *logical data model* (LDM) is a generalization of Hull and Yap's format model [HY82]. The format model fails to model an important part of network and hierarchical database systems, namely the ability to use virtual records. We model this by introducing cyclicity into the database schemas. An LDM schema is a labeled directed multigraph.[1] Each node has a particular *type*. The leaves of the schema (i.e., nodes with no outgoing edges) are all of the *basic* type, denoted graphically by $\square$. The instance of each node contains the data values stored in the database. Each interior node has one of the following types:

1. Product, denoted graphically by $\otimes$. The domain of such a node is the Cartesian product of the domains of its children.

2. Power, denoted graphically by $\circledast$. Such a node has exactly one child.

[1] In a *multigraph* one can have more than one edge between two nodes.

The domain of such a node is the set of all finite subsets of the domain of its child.

3. Union, denoted graphically by $\uplus$. The domain of such a node is the disjoint union of the domains of its children.

Person	*Parent*
Rehoboam	Solomon
Solomon	David
Solomon	Batsheba
David	Jesse

Figure 1: The *Person–Parent* relation

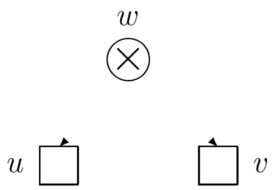

Figure 2: The *Person–Parent* relation as an LDM schema

Example 1: Fig. 1 shows a genealogy database as a relation. We can represent the structure of this relation by the LDM schema in Fig. 2. It consists of two nodes u and v of type $\square$ that correspond to the *Person* and *Parent* attributes respectively, and one node w of type $\otimes$ that contains pairs of related attributes.

For the moment, an instance $\mathbf{I}$ of an LDM schema will be an assignment to each node u of a set $I(u)$ of values from the corresponding domain. An instance corresponding to the data in Fig. 1 consists of the following assignments:

$$\begin{aligned} I(u) &= \{\text{Rehoboam}, \text{Solomon}, \text{David}\} \\ I(v) &= \{\text{Solomon}, \text{David}, \text{Batsheba}, \text{Jesse}\} \end{aligned}$$

and

$$\begin{aligned} I(w) = \{&(\text{Rehoboam}, \text{Solomon}), (\text{Solomon}, \text{David}), (\text{Solomon}, \text{Batsheba}), \\ &(\text{David}, \text{Jesse})\} \end{aligned}$$

$\blacksquare$

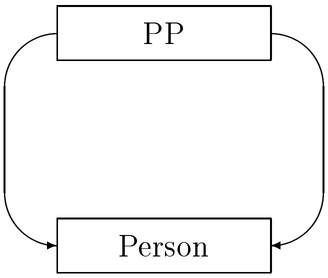

Figure 3: The genealogy as a network

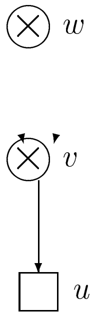

Figure 4: LDM schema corresponding to Fig. 3

Example 2: The genealogy could be represented by the network in Fig. 3. In this network there are two record types, *Person* containing the names of the people in the database, and a dummy record *PP*. There are two links (sets) that connect each dummy record to a person and his parents.

The idea behind the mapping from the network to the LDM schema in Fig. 4 is as follows. Each record type R_i is mapped into a product node v_{R_i}. For each field of R_i, v_{R_i} has a child of type $\Box$. For each link (set) in the network with R_i as a member, let R_j be the owner of the link. Then v_{R_j} is a child of v_{R_i}.

In Fig. 4, w is v_{PP} and v is v_{Person}. u corresponds to the field of the *Person* record, i.e., the person's name, and the two arcs from w to v correspond to the two links.

If the network had the same contents as the relation in Fig. 1, the corresponding instance of the LDM schema in Fig. 4 would be

$$
\begin{aligned}
I(u) &= \{\text{Rehoboam, Solomon, David, Batsheba, Jesse}\} \\
I(v) &= \{(\text{Rehoboam}), (\text{Solomon}), (\text{David}), (\text{Batsheba}), (\text{Jesse})\}
\end{aligned}
$$

and

$$
\begin{aligned}
I(w) = \{&((\text{Rehoboam}), (\text{Solomon})), ((\text{Solomon}), (\text{David})), \\
&((\text{Solomon}), (\text{Batsheba})), ((\text{David}), (\text{Jesse}))\}
\end{aligned}
$$

∎

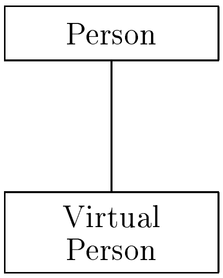

Figure 5: The genealogy as a hierarchy

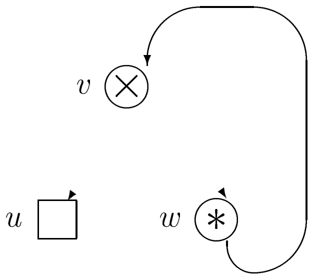

Figure 6: LDM schema corresponding to the hierarchy

Example 3: Fig. 5 shows a hierarchical representation of the genealogy. In this hierarchy, each *Person* record is related to the linked list of his parents. Even though the hierarchical model uses linked lists, this is really just a matter of the implementation. Intuitively, the user should see only the connection between a person and the set of his parents. We therefore map each record type R_i into a product node v_{R_i} as we did for the network model, with a child of type $\Box$ corresponding to each of R_i's fields. However, if R_i is a member of the link (R_i, R_j), then instead of connecting v_{R_i} to v_{R_j} directly, we connect them through a node of type $\circledast$. The corresponding LDM schema is then the cyclic schema in Fig. 6. ∎

When we have cyclic schemas, such as this one, we cannot define an instance in the same way. Writing down the instance, in this example, is very complicated, and in general, when the data as well as the schema is cyclic, we cannot write it down at all. This is similar to one of the problems with Jacobs' database logic. The mathematical theory we develop to deal with this problem is closely related to the non-well-founded sets of [Acz85]. Our approach to defining an instance of a schema is to separate the concepts of object *name* and object *value*. Intuitively, an object name is an address and the object value is the content of that address. An instance **I** then consists of:

1. an assignment of a set $I(u)$ of object names to each node u of the schema, and

2. an assignment of an object value $val(l)$ to each name l in $I(u)$.

The names are taken from a fixed infinite set L which will usually be the set of natural numbers. Values are taken from a fixed infinite set D of data values, or are built up from object names.

We now show what some of the instances in the previous examples look like when we use names and values.

Example 4: The relational instance in Example 1 consists of the following assignment of names to nodes.

$$
\begin{aligned}
I(u) &= \{1, 2, 3\} \\
I(v) &= \{4, 5, 6, 7\}
\end{aligned}
$$

and

$$I(w) = \{8, 9, 10, 11\}$$

We then assign a value $val(l)$ to each of these names. This assignment is shown in Fig. 7. ∎

$I(u)$

l	$val(l)$
1	Rehoboam
2	Solomon
3	David

$I(v)$

l	$val(l)$
4	Solomon
5	David
6	Batsheba
7	Jesse

$I(w)$

l	$val(l)$
8	$(1, 4)$
9	$(2, 5)$
10	$(2, 6)$
11	$(3, 7)$

Figure 7: Instance of the LDM schema that corresponds to a relation

Example 5: In Fig. 8 we show the instance that corresponds to the hierarchy of Example 3. ∎

$I(u)$

l	$val(l)$
1	Rehoboam
2	Solomon
3	David
4	Batsheba
5	Jesse

$I(v)$

l	$val(l)$
6	$(1, 11)$
7	$(2, 12)$
8	$(3, 13)$
9	$(4, 14)$
10	$(5, 14)$

$I(w)$

l	$val(l)$
11	$\{7\}$
12	$\{8, 9\}$
13	$\{10\}$
14	$\emptyset$

Figure 8: Instance of the LDM schema that corresponds to a hierarchy

3. Formal Description of the Logical Data Model

Definition 1: A *schema* is a tuple $\mathbf{S} = \langle V, E, <, \mu \rangle$ where:

1. (V, E) is a directed multigraph.
2. $<$ is a total order on E.
3. μ is a function from V to the set of types $\{\square, \circledast, \otimes, \textcircled{$\cup$}\}$, that satisfies the following conditions (see Fig. 9)
   (a) $\mu(v) = \square$ iff v is a leaf.
   (b) If $\mu(v) = \circledast$, then v has exactly one child.
   (c) If $\mu(v) = \textcircled{$\cup$}$, then the children of v are distinct nodes (if $\mu(v) = \otimes$, however, there can be multiple edges from v to a node w).

The order on the edges is used to induce an order on the components of tuples. $\mu(v)$ is called the *type* of v. For readability, we use the following abbreviations

1. $\mu(v) = (\circledast, w)$ is an abbreviation for "$\mu(v) = \circledast$ and its child is w."
2. (a) $\mu(v) = (\otimes, n)$ is an abbreviation for "$\mu(v) = \otimes$ and v has n children."

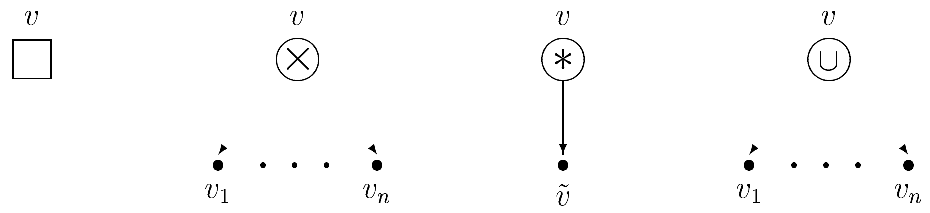

Figure 9: Nodes in LDM schemas

   (b) $\mu(v) = (\otimes, n, v_1, \ldots, v_n)$ is an abbreviation for "$\mu(v) = \otimes$, there are exactly n edges $e_1, \ldots, e_n$ with tail v, these edges are in the order $e_1 < \cdots < e_n$ and their heads are $v_1, \ldots, v_n$."

3. (a) $\mu(v) = (\uplus, n)$ is an abbreviation for "$\mu(v) = \uplus$ and v has n children."

   (b) $\mu(v) = (\uplus, n, v_1, \ldots, v_n)$ is an abbreviation for "$\mu(v) = \uplus$, there are exactly n edges $e_1, \ldots, e_n$ with tail v, these edges are in the order $e_1 < \cdots < e_n$ and their heads are $v_1, \ldots, v_n$."

We are really overloading the symbol μ, but in practice this will not cause any confusion. Some other abbreviations that we use include referring to elements of V and E as nodes and edges, respectively, of $\mathbf{S}$, and referring to $<$ as an order on the children of a node of $\mathbf{S}$. We shall ignore the order $<$ when it is clear from the context, and we shall often refer to a schema as $\langle V, E, \mu \rangle$.

An instance of $\mathbf{S}$ consists of two parts: An assignment of a set of object names to each node of $\mathbf{S}$, and an assignment of an object value to each object name.

Proviso: We assume a fixed infinite set L of object names and a fixed infinite set D of data values.

Definition 2: An *instance* of $\mathbf{S}$ is a tuple $\mathbf{I} = \langle I, val \rangle$ that satisfies the following:

1. $I : V \to 2^L$ is an assignment of sets of object names to nodes. We require that $I(v)$ and $I(w)$ be disjoint whenever v and w are distinct nodes of $\mathbf{S}$.

2. val is a mapping with domain $\cup_{v \in V} I(v)$, i.e., from the set of all the names that are in the instance. The mapping val must satisfy:

   (a) If $\mu(v) = \Box$ and $l \in I(v)$, then $val(l)$ is a member of the set D of data values.

   (b) If $\mu(v) = (\otimes, n, v_1, \ldots, v_n)$ and $l \in I(v)$, then $val(l)$ is a tuple $(l_1, \ldots, l_n)$ such that for each i, $1 \leq i \leq n$, l_i is an element of $I(v_i)$.

   (c) If $\mu(v) = (\circledast, w)$ and $l \in I(v)$, then $val(l)$ is a finite subset of $I(w)$.

   (d) If $\mu(v) = (\textcircled{\scriptsize $\cup$}, n, v_1, \ldots, v_n)$ and $l \in I(v)$, then $val(l) \in I(v_1) \cup \cdots \cup I(v_n)$.

Definition 3: A *finite instance* of **S** is an instance $\mathbf{I} = \langle I, val \rangle$ of **S** such that for each node v of **S**, $I(v)$ is finite.

We shall be interested here mostly in finite instances, since these correspond to real databases. If l is in $\cup_{v \in V} I(v)$, we say that it is a name *in* **I**, and $val(l)$ is called *its* value. The set $\cup_{v \in V} val[I(v)]$ is called the set of values *in* **I**.

Definition 4: Let **I** be an instance of the schema **S**, and let v be a node of **S** of type $(\otimes, n, v_1, \ldots, v_n)$. Let l be any name in $I(v)$. If $1 \leq i \leq n$, then $\Pi_i(l)$ will be the i^{th} component of $val(l)$. We shall also use the notation $\Pi_{v_i}(l)$ for this component, whenever this does not result in any ambiguity.

The following definition says when it is meaningful to compare two names, i.e., if v and w are nodes of **S**, $l_1 \in I(v)$ and $l_2 \in I(w)$, is it possible for l_1 and l_2 to have the same value?

Definition 5: We say that two nodes v and w in a schema **S** are *similar* iff they are of the same type and have the same children, i.e., if one of the following holds:

1. $\mu(v) = \mu(w) = \Box$.

2. For some node u, $\mu(v) = \mu(w) = (\circledast, u)$.

3. For some n and nodes $u_1, \ldots, u_n$, $\mu(v) = \mu(w) = (\otimes, n, u_1, \ldots, u_n)$.

4. For some n and nodes $u_1, \ldots, u_n$, $\mu(v) = \mu(w) = (\circledcirc, n, u_1, \ldots, u_n)$.

A query on an LDM schema **S** will involve the addition of some nodes to **S**. For this we need the following definitions.

Definition 6: Let $\mathbf{S} = \langle V, E, <, \mu \rangle$ be a schema. $\mathbf{S}' = \langle V', E', <', \mu' \rangle$ is an *extension* of **S** iff

1. $V \subseteq V'$
2. (a) $E \subseteq E'$
   (b) If $(v_1, v_2) \in E' - E$, then v_1 is in V', i.e. all new edges are either between new nodes, or from a new node to a node in V.
3. $<'|_{E \times E} = <$, i.e., $<'$ is a conservative extension of $<$.
4. $\mu' \mid_V = \mu$, i.e., μ' is a conservative extension of μ.

The intuition behind (1) and (2) is that the schema $\mathbf{S}'$ adds some extra nodes $(V' - V)$ to V, and adds some edges $(E' - E)$ to E, but it does not change the schema **S**, i.e., it does not add edges between nodes in V, or from nodes in V to new nodes.

Let $\mathbf{S}'$ be an extension of **S**. We define an extension of **I** to an instance of $\mathbf{S}'$ as follows.

Definition 7: Let $\mathbf{S}'$ be an extension of **S**, and let $\mathbf{I} = \langle I, val \rangle$ be an instance of **S**. We say that an instance $\mathbf{I}' = \langle I', val' \rangle$ of $\mathbf{S}'$ is an extension of **I** to $\mathbf{S}'$ iff

1. For all v in V, $I'(v) = I(v)$.
2. If v is a node of **S** and $l \in I(v)$, then $val'(l) = val(l)$.

The proof of the following lemma is straightforward.

Lemma 1: Let $\mathbf{S}'$ be an extension of **S**, and let $\mathbf{I}'$ be an instance of $\mathbf{S}'$. Then there is a unique instance **I** of **S** such that $\mathbf{I}'$ is the extension of **I** to $\mathbf{S}'$. This instance is called the *restriction* of $\mathbf{I}'$ to **S**. ∎

We conclude this section with a definition of isomorphism. Two instances are isomorphic if they are essentially the same, i.e., if they differ only by renaming. As we shall want to show that the result of a query is well-defined up to isomorphism, we give a stronger definition of isomorphism. Let $\mathbf{I}$ be an instance of $\mathbf{S}$, let $\mathbf{S}'$ be an extension of $\mathbf{S}$ and let $\mathbf{I}_1$ and $\mathbf{I}_2$ be extensions of $\mathbf{I}$ to $\mathbf{S}'$. We shall say that $\mathbf{I}_1$ and $\mathbf{I}_2$ are isomorphic relative to $\mathbf{S}$, if there is an isomorphism between $\mathbf{I}_1$ and $\mathbf{I}_2$ that leaves the objects of $\mathbf{I}$ fixed. In the case of a query, this means that an isomorphism relative to the database leaves the contents of the database fixed.

Definition 8: Let $\mathbf{S}'$ be an extension of $\mathbf{S}$ and let $\mathbf{I} = \langle I, val \rangle$ be an instance of $\mathbf{S}$. Let $\mathbf{I}_1 = \langle I_1, val_1 \rangle$ and $\mathbf{I}_2 = \langle I_2, val_2 \rangle$ be two extensions of $\mathbf{I}$ to $\mathbf{S}'$. We say that $\mathbf{I}_1$ and $\mathbf{I}_2$ are *isomorphic relative to* $\mathbf{S}$ iff there is a mapping

$$g\colon \bigcup_{v \in \mathbf{S}'} I_1(v) \xrightarrow[\text{onto}]{1\text{–}1} \bigcup_{v \in \mathbf{S}'} I_2(v)$$

such that

1. For each node v of $\mathbf{S}$, g is the identity on $I(v)$.
2. For each node v of $\mathbf{S}'$, g maps $I_1(v)$ onto $I_2(v)$.
3. If v is a node of $\mathbf{S}'$ and $l \in I_1(v)$, then
   (a) If v is of type $\square$, then $val_2(g(l)) = val_1(l)$.
   (b) If v is of type $(\otimes, n)$, then
   $$val_2\Big(g(l)\Big) = \Bigg(g\Big(\Pi_1\Big(val_1(l)\Big)\Big), \ldots, g\Big(\Pi_n\Big(val_1(l)\Big)\Big)\Bigg)$$
   (c) If v is of type $\circledcup$, then $val_2(g(l)) = g(val_1(l))$.
   (d) If v is of type $\circledast$, then $g[val_2(l)] = val_1[g(l)]$.

As a special case of this definition we get the definition of ordinary isomorphism.

Definition 9: Let $\mathbf{I}_1 = \langle I_1, val_1 \rangle$ and $\mathbf{I}_2 = \langle I_2, val_2 \rangle$ be instances of $\mathbf{S}$. We say that $\mathbf{I}_1$ and $\mathbf{I}_2$ are *isomorphic* iff they are isomorphic relative to the empty schema, i.e., the schema with $V = E = \mu = \emptyset$.

4. LDM Logic

In this section we define the LDM logic. The logic is similar to the relational tuple calculus, and will be used as part of the logical query language. The logic can also be used to specify integrity constraints on LDM schemas, and to define views. Throughout this section $\mathbf{S} = \langle V, E, \mu \rangle$ will be a fixed schema, and $\mathbf{I} = \langle I, val \rangle$ a fixed instance of $\mathbf{S}$.

Each variable in the logic has a fixed *sort*, where the sorts are the elements of V. The sorts define the domains over which the variables range. For example, if x is a variable of sort v, then x ranges over $I(v)$. The analogue to this in the relational calculus is a tuple variable that ranges over a specific relation. We shall usually write a variable with its sort as a subscript, e.g., x_v. Variables with different subscripts denote distinct variables, so that x_u is a different variable from x_v. Even though variables range over object names, we think of them as ranging over objects. Thus, we shall refer to "the name of x_v" and to "the value of x_v".

Definition 10: The *atomic formulas over* $\mathbf{S}$ are the following:

1. $x_v \; \pi_t \; y_w$, where w is a node of type $\otimes$ and v is its t^{th} child.
2. $x_v \; \rho \; y_w$, where w is a node of type ☺ and v is one of its children.
3. $x_v \in y_w$, where w is of type $(\circledast, v)$.
4. $x_v =_n y_v$.
5. $x_v =_v y_w$, where v and w are similar nodes.
6. $x_v =_v d$, where d is a data element in D, and v is of type $\square$.

The atomic formula $x_v \; \pi_t \; y_w$ means that the name of x_v is the t^{th} component of the value of y_w. Note that we have to mention which component of w we are referring to, since there may be multiple edges from w to v. However, we shall also write $x_v \; \pi \; y_w$ when this is unambiguous. The atomic formula $x_v \; \rho \; y_w$ means that the value of y_w is x_v. Since there is only one edge from w to v, we use ρ rather than ρ_t. The atomic formula $x_v \in y_w$ means that x_v is a member of the value of y_w.

There are several kinds of equality. The atomic formula $x_v =_n y_v$ means that the names of x_v and y_v are equal. Since $I(v)$ and $I(w)$ are disjoint whenever $v \neq w$, we do not need atomic formulas of the form $x_v =_n y_w$ for $v \neq w$. The atomic formula $x_v =_v y_w$ means that the values of x_v and y_w are equal. This is meaningful only when v and w are similar nodes. Finally, the atomic formula $x_v =_v d$ means that the data value of x_v is equal to the data element d.

Definition 11: A *well-formed LDM formula over a schema* $\mathbf{S}$ is:

1. An atomic formula
2. $\phi_1 \vee \phi_2$, where ϕ_1 and ϕ_2 are well-formed formulas.
3. $\neg\phi_1$, where ϕ_1 is a well-formed formula.
4. $(\forall x_v)\phi_1$, where ϕ_1 is a well-formed formula.

The free variables of ϕ are defined in the same way as in first-order logic.

We use $\phi_1 \wedge \phi_2$, $(\exists x_v)\phi$, $\phi_1 \Rightarrow \phi_2$ and $\phi_1 \Leftrightarrow \phi_2$ with the standard meanings. Another useful abbreviation is the following.

Definition 12: The formula "$x_v =_v (x^1_{v_1}, \ldots, x^n_{v_n})$" where v is a node of type $(\otimes, n, v_1, \ldots, v_n)$ means "$x^1_{v_1}\ \pi_1\ x_v \wedge \cdots \wedge x^n_{v_n}\ \pi_n\ x_v$."

We now define satisfaction of LDM formulas. Let $\phi(x^1_{v_1}, \ldots, x^n_{v_n})$ be an LDM formula whose free variables are $x^1_{v_1}, \ldots, x^n_{v_n}$. Let $l_1, \ldots, l_n$ be an assignment of object names to the free variables in the formula, i.e., each l_i is a member of the corresponding $I(v_i)$. $\models_{\mathbf{I}} \phi(l_1, \ldots, l_n)$ means that ϕ is satisfied by $l_1, \ldots, l_n$ in the instance $\mathbf{I}$. When $\mathbf{I}$ is clear from the context, we shall write $\models$ instead of $\models_{\mathbf{I}}$.

Definition 13: Let $\phi(x^1_{v_1}, \ldots, x^n_{v_n})$ be a formula with free variables $x^1_{v_1}, \ldots, x^n_{v_n}$, and let $l_i \in I(v_i)$ for all i, $1 \leq i \leq n$. Then $\models_{\mathbf{I}} \phi(l_1, \ldots, l_n)$ iff the following hold:

1. If ϕ is $x^i_v\ \pi_t\ y^j_w$, then $\models_{\mathbf{I}} (x^i_v\ \pi_t\ x^j_w)(l_1, \ldots, l_n)$ iff $l_i = \Pi_t(l_j)$.
2. If ϕ is $x^i_v\ \rho\ y^j_w$, then $\models_{\mathbf{I}} (x^i_v\ \rho\ x^j_w)(l_1, \ldots, l_n)$ iff $l_i = val(l_j)$.
3. If ϕ is $x^i_v \in x^j_w$, then $\models_{\mathbf{I}} (x^i_v \in x^j_w)(l_1, \ldots, l_n)$ iff $l_i \in val(l_j)$.

4. If ϕ is $x_v^i =_n x_v^j$, then $\models_{\mathbf{I}} (x_v^i =_n x_v^j)(l_1, \ldots, l_n)$ iff $l_i = l_j$.
5. If ϕ is $x_v^i =_v x_w^j$, then $\models_{\mathbf{I}} (x_v^i =_v x_w^j)(l_1, \ldots, l_n)$ iff $val(l_i) = val(l_j)$.
6. If ϕ is $x_v^i =_v d$, then $\models_{\mathbf{I}} (x_v^i =_v d)(l_1, \ldots, l_n)$ iff $val(l_i) = d$.
7. $\models_{\mathbf{I}} (\phi_1 \vee \phi_2)$ iff $\models_{\mathbf{I}} \phi_1$ or $\models_{\mathbf{I}} \phi_2$.
8. $\models_{\mathbf{I}} \neg\phi$ iff $\models_{\mathbf{I}} \phi$ does not hold.
9. If ϕ is a formula with free variables $x_{v_1}^1, \ldots, x_{v_n}^n, y_w$, then

$$\models_{\mathbf{I}} \Big((\forall y_w)\phi\Big)(l_1, \ldots, l_n) \text{ iff for all } l \in I(w), \quad \models_{\mathbf{I}} \phi(l_1, \ldots, l_n, l)$$

Definition 14: Let ϕ be an LDM sentence. We say that $\mathbf{I}$ satisfies ϕ iff $\models_{\mathbf{I}} \phi$ holds.

Example 6: This example and the next one will be over the LDM schema of Fig. 6 with the instance of Fig. 8. The LDM formula $\phi(x_u, y_v) = (x_u\ \pi_1\ y_v)$ says that the name of x_u is equal to the first component of the value of y_v. $\models_{\mathbf{I}} \phi(l_1, l_2)$ holds for the (l_1, l_2) pairs $(1, 6)$, $(2, 7)$, $(3, 8)$, $(4, 9)$, and $(5, 10)$. ∎

Example 7: The following constraint says that each name in u is related to exactly one set in w. For example, '8' and '9' as parents of '2' must be in one set rather that in two different sets. The constraint is

$$\phi = (\forall x_u)(\forall y_v^1)(\forall y_v^2)(\forall z_w^1)(\forall z_w^2)\Big(y_v^1 =_v (x_u, z_w^1) \wedge y_v^2 =_v (x_u, z_w^2) \\ \Rightarrow z_w^1 =_n z_w^2\Big)$$

In other words, each name in u (x_u) has at most one name in w (z_w^1 and z_w^2) associated with it. This association is through y_v^1 and y_v^2.

Note that this constraint says that each *name* in u is associated with at most one set in w, rather than saying that each *person* in the database is associated with at most one such set. There could still be duplication in u, e.g., two names with the data value "Solomon." One way to prevent this is through the constraint

$$\psi = (\forall x_u^1)(\forall x_u^2)(x_u^1 =_v x_u^2 \Rightarrow x_u^1 =_n x_u^2)$$

∎

The following lemma shows that a slightly restricted logic has the same power. This restricted logic does not have atomic formulas that compare data values of internal nodes. This lemma makes subsequent proofs and definitions simpler.

Lemma 2: Let $\phi(x^1_{v_1}, \ldots, x^n_{v_n})$ be an LDM formula whose free variables are the variables $x^1_{v_1}, \ldots, x^n_{v_n}$. There is an LDM formula $\psi(x^1_{v_1}, \ldots, x^n_{v_n})$ with the same free variables, that does not contain any atomic subformula of the form $x_u =_v y_w$ with $\mu(v)$ and $\mu(w)$ different from $\square$, such that ψ is equivalent to ϕ. That is, for all instances $\mathbf{I}$ of $\mathbf{S}$ and all $l_1, \ldots, l_n$, $l_i \in I(v_i)$, $\models_{\mathbf{I}} \phi(l_1, \ldots, l_n)$ iff $\models_{\mathbf{I}} \psi(l_1, \ldots, l_n)$.

Proof: The proof is by induction on the size of ϕ. We show how to construct ψ for formulas of the form $x_u =_v y_v$, where u and v are similar and not of type $\square$. The result then follows immediately.

We distinguish between the possible types of v and w.

1. If u and v are of type $(\circledast, w)$, then $\psi(x_u, y_v)$ is $(\forall z_w)(z_w \in x_u \Leftrightarrow z_w \in y_v)$, where z_w is a new variable. Let $\mathbf{I}$ be an instance of $\mathbf{S}$. Then

$$\begin{aligned} \models_{\mathbf{I}} (x_u =_v y_v)(l_1, l_2) &\Leftrightarrow val(l_1) = val(l_2) \\ &\Leftrightarrow \text{For all } l \text{ in } I(w), l \in val(l_1) \Leftrightarrow l \in val(l_2) \\ &\Leftrightarrow \models_{\mathbf{I}} \Big((\forall z_w)(z_w \in x_u \Leftrightarrow z_w \in y_v)\Big)(l_1, l_2) \end{aligned}$$

   and therefore $\phi \Leftrightarrow \psi$ is valid.

2. If u and v are of type $(\otimes, n, w_1, \ldots, w_n)$, then $\psi(x_u, y_v)$ is

$$(\forall z^1_{w_1}) \cdots (\forall z^n_{w_n})\Big((z^1_{w_1}\ \pi_1\ x_u \Leftrightarrow z^1_{w_1}\ \pi_1\ y_v) \\ \wedge \cdots \wedge (z^n_{w_n}\ \pi_n\ x_u \Leftrightarrow z^n_{w_n}\ \pi_n\ y_v)\Big),$$

   where $z^1_{w_1}, \ldots, z^n_{w_n}$ are n distinct new variables. Let $\mathbf{I}$ be an instance of $\mathbf{S}$. Then $\models_{\mathbf{I}} (x_u =_v y_v)(l_1, l_2)$ is equivalent to $val(l_1) = val(l_2)$. If $val(l_1) = (l^1_1, \ldots, l^n_1)$ and $val(l_2) = (l^1_2, \ldots, l^n_2)$, $val(l_1) = val(l_2)$ is equivalent to the conjunction of $l^i_1 = l^i_2$, for $i = 1, \ldots, n$. In other words, $val(l_1) = val(l_2)$ iff, for $i = 1, \ldots, n$

$$\models_{\mathbf{I}} \Big((\forall z^i_w)(z^i_{w_i}\ \pi_i\ x_u \Leftrightarrow z^i_{w_i}\ \pi_i\ y_v)\Big)(l_1, l_2).$$

Therefore $\models_{\mathbf{I}} (x_u =_v y_v)(l_1, l_2)$ is equivalent to

$$\models_{\mathbf{I}} (\forall z^1_{w_1}) \cdots (\forall z^n_{w_n})\Big((z^1_{w_1}\ \pi_1\ x_u \Leftrightarrow z^1_{w_1}\ \pi_1\ y_v) \\ \wedge \cdots \wedge (z^n_{w_n}\ \pi_n\ x_u \Leftrightarrow z^n_{w_n}\ \pi_n\ y_v)\Big)(l_1, l_2),$$

i.e., $\phi \Leftrightarrow \psi$ is valid.

3. If u and v are of type $(\circledcirc, n, w_1, \ldots, w_n)$, then $\psi(x_u, y_v)$ is

$$(\exists z^1_{w_1})(z^1_{w_1}\ \rho\ x_u \wedge z^1_{w_1}\ \rho\ y_v) \vee \cdots \vee (\exists z^n_{w_n})(z^n_{w_n}\ \rho\ x_u \wedge z^n_{w_n}\ \rho\ y_v),$$

where $z^1_{w_1}$, …, $z^n_{w_n}$ are n distinct new variables. Let $\mathbf{I}$ be an instance of $\mathbf{S}$. Then $\models_{\mathbf{I}} (x_u =_v y_v)(l_1, l_2)$ is equivalent to $val(l_1) = val(l_2) = l$. This holds iff for some i, $1 \le i \le n$, $l \in I(w_i)$, in which case

$$\models_{\mathbf{I}} \Big((\exists z^i_{w_i})(z^i_{u_i}\ \rho\ x_u \wedge z^i_{w_i}\ \rho\ y_v)\Big)(l_1, l_2)$$

and once more $\phi \Leftrightarrow \psi$ is valid. ∎

From now on, we shall assume that $x_u =_v y_v$ can appear as a subformula only when $\mu(v) = \mu(w) = \square$, as far as proofs are concerned, but shall be able to use the more general form when convenient.

The proof of the following lemma, that says that satisfaction is preserved under isomorphism, is straightforward.

Lemma 3: Let $\mathbf{S}'$ be an extension of $\mathbf{S}$, and let $\mathbf{I}_1$ and $\mathbf{I}_2$ be extensions of $\mathbf{I}$ to $\mathbf{S}'$. Let g be an isomorphism from $\mathbf{I}_1$ to $\mathbf{I}_2$ relative to $\mathbf{S}$, and let $\phi(x^1_{v_1}, \ldots, x^n_{v_n})$ be an LDM formula. Then

$$\models_{\mathbf{I}_1} \phi(l_1, \ldots, l_n) \Leftrightarrow \models_{\mathbf{I}_2} \phi\big(g(l_1), \ldots, g(l_n)\big) \ \blacksquare$$

Finally, we show that our logic is effective over finite instances.

Lemma 4: Let $\phi(x^1_{v_1}, \ldots, x^n_{v_n})$ be an LDM formula over $\mathbf{S}$ whose free variables are the variables $x^1_{v_1}, \ldots, x^n_{v_n}$. Let $\mathbf{I}$ be a finite instance and let $l_i \in I(v_i)$ for all i, $1 \le i \le n$. Then $\models_{\mathbf{I}} \phi(l_1, \ldots, l_n)$ can be determined effectively.

Proof: We show this by induction on the size of the formula. For atomic formulas testing for satisfaction is straightforward. Testing for disjunction and negation is also clearly effective. For quantification we make use of the finiteness of $\mathbf{I}$. In order to test whether $\models_{\mathbf{I}} ((\forall y_w)\phi)(l_1, \ldots, l_n)$, we test whether $\models_{\mathbf{I}} \phi(l_1, \ldots, l_n, l)$ for each l in the finite set $I(w)$. ∎

5. The Logical Query Language

In the relational model the result of a query is a relation. A natural generalization would be for the result of a query in our model to be another LDM schema, called henceforth the *query schema*, together with an instance of that schema. We modify this slightly, by not requiring that the query's schema be an independent schema, but instead allowing the successors of nodes in the query to be nodes in the database schema.

Continuing the analogy with the relational calculus, the natural thing to do would be to let the query be some LDM formula ϕ. The resulting instance would then consist of those objects that satisfy ϕ.

There are two problems with this approach. First, while it is clear how an LDM formula can select objects that satisfy certain conditions, it is not clear how an LDM formula can construct new objects. One solution would be to prevent the query from referring directly to object names, but rather have the query refer only to object values. We could then find all possible values than might appear in the result, assign them arbitrary names, and show that the result is unique up to isomorphism.

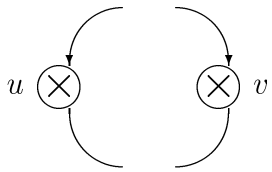

Figure 10: LDM schema

$I_1(u)$

l	$val(l)$
1	(3)
2	(4)

$I_1(v)$

l	$val(l)$
3	(1)
4	(2)

Figure 11: A possible result of the query

This still does not solve the second problem, which is how to deal with cyclicity. Not only do we need the ability to refer directly to object names in order to deal with cycles, but even then the result of the query is not always

$I_2(u)$

l	$val(l)$
1	(3)
2	(4)

$I_2(v)$

l	$val(l)$
3	(2)
4	(1)

Figure 12: Another possible result of the query

defined uniquely. For example, if the query schema is that of Fig. 10, and the query specifies that $I(u)$ and $I(v)$ each contain at least two different names, then there is no way to choose between the two incomparable instances in Fig. 11 and 12. Our solution to this problem is to restrict the queries to ones not that do not contain cycles, while allowing cyclicity in the database. Furthermore, we allow the query to refer explicitly to names only in nodes of the database. For a more detailed discussion of the motivation underlying our approach see [Kup85]. More recent work, e.g. IQL ([**?**]), has shown how query languages could be defined to allow cyclic queries.

5.1. The Query Language

Definition 15: Let $\mathbf{S} = \langle V, E, \mu \rangle$ be an LDM schema. A query on $\mathbf{S}$ consists of a tuple $\mathbf{Q} = \langle \mathbf{S_Q}, \Phi_{\mathbf{Q}}, \prec_{\mathbf{Q}} \rangle$ where

1. $\mathbf{S_Q}$ is an extension of $\mathbf{S}$.
2. $\prec_{\mathbf{Q}}$ is a topological order on the nodes in $V_{\mathbf{Q}} - V$, i.e., $\prec_{\mathbf{Q}}$ is a linear order such that if v is a child of w then $v \prec_{\mathbf{Q}} w$.
3. $\Phi_{\mathbf{Q}}$ is a set of pairs $\langle v, \phi_v \rangle$ that assigns a formula ϕ_v to the node v, for each node v in $V_{\mathbf{Q}} - V$. The formula ϕ_v that corresponds to the node v satisfies
   (a) ϕ_v has only one free variable, and it is of sort v.
   (b) All other variables in ϕ_v are bound. Each of their sorts is either a node of the database schema $\mathbf{S}$ or is a query node that precedes v under $\prec_{\mathbf{Q}}$.

The order $\prec_{\mathbf{Q}}$ is used to specify the order in which we define the result of the query. Each formula ϕ_v specifies the contents of v in terms of database nodes, and of query nodes that precede v.

Before continuing with the formal details we give several examples of queries. The database schema in all these examples is the genealogy schema of Fig. 6 with the instance of Fig. 8.

Example 8: The schema of $\mathbf{Q}_1$ is shown in Fig. 13. The formula $\phi_{u'}(x_{u'})$ is

$$(\exists y_u)(x_{u'} =_v y_u)$$

In other words we want $I(u')$ to be a copy of $I(u)$. (We eliminate, however, any duplication that may be in $I(u)$.) The result of the query is shown in Fig. 14.[2] ∎

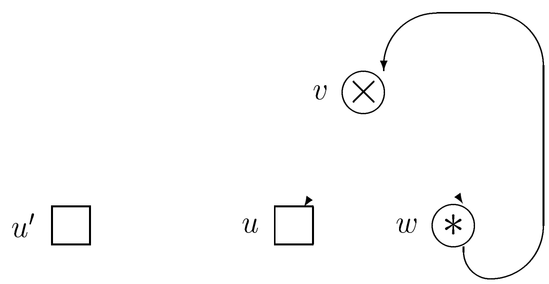

Figure 13: Schema of $\mathbf{Q}_1$

Example 9: The schema of $\mathbf{Q}_2$ is shown Fig. 15. We want v' to contain the set of parents of Solomon, so we have the formulas

$$\phi_{u'}(x_{u'}) = (\exists y_u^1)(\exists y_u^2)(\exists z_v^1)(\exists z_v^2)(\exists z_w^3)\Big((y_u^1 =_v x_{u'}) \wedge (y_u^2 =_v \text{“Solomon”}) \\ \wedge (z_v^2 =_v (y_u^2, z_w^3)) \wedge (z_v^1 \in z_v^w) \wedge (y_u^1 \; \pi_1 \; z_v^1)\Big)$$

and $\phi_{v'}(x_{v'}) = (\forall y_{u'})(y_{u'} \in x_{v'})$.

[2]In all these examples, the result is defined only up to isomorphism relative to $\mathbf{S}$, i.e., the choice of names is arbitrary

$I(u')$

l	$val(l)$
17	Rehoboam
18	Solomon
19	David
20	Batsheba
21	Jesse

Figure 14: Result of $\mathbf{Q}_1$

The intuition behind this is that $\phi_{u'}(x_{u'})$ says that there is some name (y_u^2) in $I(u)$ with the value "Solomon," and another name (y_u^1) with value equal to $x_{u'}$. The rest of the formula says that y_u^1 is a parent of y_u^2. $\phi_{v'}(x_{v'})$ says that $I(v')$ contains all the names in $I(u')$ in one set. The result of the query is shown in Fig. 16. ∎

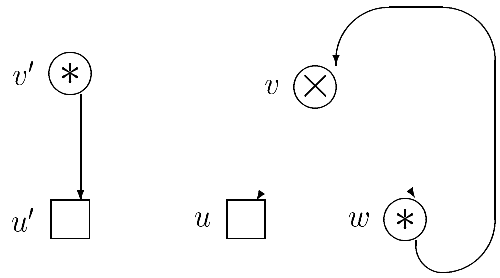

Figure 15: Schema of $\mathbf{Q}_2$

We now formally define the result of a logical query. We start by looking at queries that add just one node to the schema. We shall call such queries *simple queries.*

Definition 16: A query $\mathbf{Q}$ is called a *simple query* if $|V_{\mathbf{Q}} - V| = 1$. We shall use the notation $\mathbf{Q}_v$ for a simple query with $V_{\mathbf{Q}} - V = \{v\}$.

Let $\mathbf{Q}_v$ be a simple query on a schema $\mathbf{S}$ and let $\mathbf{I}$ be an instance of $\mathbf{S}$. The result of $\mathbf{Q}_v$ on $\mathbf{I}$ is an extension $\mathbf{I}_v$ of $\mathbf{I}$ to $\mathbf{S}_{\mathbf{Q}_v}$. In order to define $\mathbf{I}_v$

$I(u')$

l	$val(l)$
17	David
18	Batsheba

$I(v')$

l	$val(l)$
19	$\{17, 18\}$

Figure 16: Result of $\mathbf{Q}_2$

we have to define what names $I_v(v)$ contains, and what the values of these names are. We would like $I_v(v)$ to contain all the objects that satisfy $\phi_v(x_v)$. The problem with using this as a definition of $\mathbf{I}_v$ is that $\phi_v(x_v)$ is satisfied by *names* and since $I_v(v)$ has not yet been defined it is meaningless to talk about the objects that satisfy ϕ_v. It might seem that this is really a trivial problem, but suppose that $\phi_v(x_v)$ included the conjunct $(\forall y_v)(\forall z_v)(y_v =_n z_v)$, i.e., $I(v)$ can contain at most one name. If the rest of ϕ_v allowed several possibilities for the value of this name, there would be no way to choose which one should be in the result.

The reason this is not a problem for us is that such a formula is not allowed in our query language—all bound variables in our language must refer to database nodes or to nodes that precede v, but cannot refer to v itself. As a result of this restriction, it turns out that although ϕ_v refers to names, ϕ_v is actually a statement about values. We can therefore find the values that satisfy ϕ_v and pick the names arbitrarily.

Definition 17: Let val be a value (i.e., anything that could be the value of x_v). We say that val is a *candidate value for* v[3] if the following holds. Let l be some new name, i.e., a name that does not appear in $\mathbf{I}$. Let $\mathbf{I}_v$ be the extension of $\mathbf{I}$ to $\mathbf{S}_{\mathbf{Q}_v}$ with $I_v(v) = \{l\}$ and $val_v(l) = val$. Then $\models_{\mathbf{I}} \phi_v(l)$.

By using this arbitrary name we are able to express the fact that val is one of the objects that should be in the result of the query. Note that this definition is where we make use of the fact that the query is acyclic.

We first show that the particular choice of name is unimportant.

Lemma 5: Let val be an data value and let $\mathbf{I}_1$ and $\mathbf{I}_2$ be two extensions of $\mathbf{I}$ to $\mathbf{S}_{\mathbf{Q}_v}$ defined by, respectively, $I_1(v) = \{l_1\}$, $val_1(l_1) = val$, and $I_2(v) = \{l_2\}$,

[3]This really depends on $\mathbf{Q}$ and $\mathbf{I}$ as well, but they should be clear from the context.

$val_2(l_2) = val$. Then $\models_{\mathbf{I}_1} \phi_v(l_1) \Leftrightarrow \models_{\mathbf{I}_2} \phi_v(l_2)$.

Proof: By definition ϕ_v has only one free variable of sort v, i.e., the variable x_v. By inspection we can see that the only atomic formulas that can contain x_v are $x_w\ \pi_i\ x_v$, $x_w\ \rho\ x_v$, $x_w \in x_v$, $x_v =_v d$, $x_v =_v x_w$ and $x_v =_n x_v$. The last of these is always true, and it is easy to see that the truth of the others depends only on the value of x_v and not on its name. The proof is then a straightforward induction. ∎

We now define the result of $\mathbf{S}_{\mathbf{Q}_v}$. Take all the candidate values for v, pick a new name for each one of them and put all of these names into $I_v(v)$. Notice that the set of candidate values can be infinite in principle, even when I is a finite instance. Queries with the property that the set of candidate values is finite correspond to the safe queries in the relational model. In the next section we look at this issue in more detail.

Definition 18: The result of $\mathbf{Q}_v$ is the extension $\mathbf{I}_v$ of $\mathbf{I}$ to $\mathbf{S}_{\mathbf{Q}_v}$ defined as follows. Let R be the set of all the candidate values for v and let $\{l_{val} \mid val \in R\}$ be a set of new names, i.e., names that do not appear in $\mathbf{I}$. $I_v(v)$ is defined as the set $\{l_{val} \mid val \in R\}$ with $val_v(l_{val}) = val$ for each $val \in R$.

We now show that this definition has the desired properties. In other words, the result is well defined up to isomorphism relative to $\mathbf{S}$, everything in the result satisfies ϕ_v and we cannot add anything else that satisfies ϕ_v to the result without introducing duplication.

The proof of the following lemma is similar to the proof of Lemma 5.

Lemma 6: Let $\mathbf{I}_1$ be an extension of $\mathbf{I}$ to $\mathbf{S}_{\mathbf{Q}_v}$. Let l be an element of $I_1(v)$ and let $\mathbf{I}_2$ be the extension of $\mathbf{I}$ to $\mathbf{S}_{\mathbf{Q}_v}$ defined by $\mathbf{I}_2(v) = \{l\}$ and $val_2(l) = val_1(l)$. Then $\models_{\mathbf{I}_1} \phi_v(l) \Leftrightarrow \models_{\mathbf{I}_2} \phi_v(l)$. ∎

Lemma 7:

1. Let $\mathbf{I}_1$ and $\mathbf{I}_2$ be two results of $\mathbf{Q}_v$. Then $\mathbf{I}_1$ and $\mathbf{I}_2$ are isomorphic relative to $\mathbf{S}$.

2. Let $\mathbf{I}_v$ be the result of $\mathbf{S}_{\mathbf{Q}_v}$. Then for each l in $I_v(v)$, $\models_{\mathbf{I}_v} \phi_v(l)$.

Proof:

1. Let l_1 be an element of $I_1(v)$. Since $val_1(l_1)$ is a candidate value for v, there must be some l_2 in $I_2(v)$ with $val_2(l_2) = val_1(l_1)$. Since, by the definition of the result of the query, both $I_1(v)$ and $I_2(v)$ have no duplication, we immediately get a 1–1 correspondence between the names of $I_1(v)$ and $I_2(v)$. It is straightforward to show that this correspondence is an isomorphism.

2. Let l be an arbitrary element of $I_v(v)$, and let $\mathbf{I}^* = \langle I^*, val^* \rangle$ be the extension of $\mathbf{I}$ to $\mathbf{S}_{\mathbf{Q}_v}$ defined by $I^*(v) = \{l\}$ and $val^*(l) = val(l)$. By Lemma 6
$$\models_{\mathbf{I}_v} \phi_v(l) \Leftrightarrow \models_{\mathbf{I}^*} \phi_v(l)$$
Since $val(l)$ is a candidate value for v, we can extend $\mathbf{I}$ to an instance $\mathbf{I}^{**}$ of $\mathbf{S}_{\mathbf{Q}_v}$ by defining $I^{**}(v) = \{l^{**}\}$, $val(l^{**}) = val(l)$. We then have $\models_{\mathbf{I}^{**}} \phi_v(l^{**})$. By Lemma 5, $\models_{\mathbf{I}^*} \phi_v(l)$ and therefore $\models_{\mathbf{I}} \phi_v(l)$. ∎

We now define the result of an arbitrary query $\mathbf{Q}$. To do this, we first define composition of queries.

Definition 19: Let $\mathbf{Q}_1$ be a query and let $\mathbf{Q}_2$ be a query on $\mathbf{S}_{\mathbf{Q}_1}$. $\mathbf{Q}_2 \circ \mathbf{Q}_1$ is the query on $\mathbf{S}$ that we get by composing them in the following way. $\mathbf{Q}_2 \circ \mathbf{Q}_1$ consists of $\mathbf{S}_{\mathbf{Q}_2 \circ \mathbf{Q}_1} = \mathbf{S}_{\mathbf{Q}_2}$, $\Phi_{\mathbf{Q}_2 \circ \mathbf{Q}_1} = \Phi_{\mathbf{Q}_1} \cup \Phi_{\mathbf{Q}_2}$ and

$$\prec_{\mathbf{Q}_2 \circ \mathbf{Q}_1} = \prec_{\mathbf{Q}_1} \cup \prec_{\mathbf{Q}_2} \cup \{(v, w) \mid v \in V_{\mathbf{Q}_1}, w \in V_{\mathbf{Q}_2}\}$$

Lemma 8: $\mathbf{Q}_2 \circ \mathbf{Q}_1$ is a query on $\mathbf{S}$. ∎

Let the nodes added by the query $\mathbf{Q}$ be $V_{\mathbf{Q}} - V = \{v_1, \ldots, v_n\}$ where $v_1 \prec \cdots \prec v_n$. We define a sequence of simple queries $\mathbf{Q}_{v_1}, \ldots, \mathbf{Q}_{v_n}$, as follows. Each $\mathbf{Q}_{v_i}$ is a query on the schema of $\mathbf{Q}_{v_{i-1}}$. $\mathbf{Q}_{v_i}$ adds the node v_i to that schema, and the formula for v_i is ϕ_{v_i}. It is easy to see that $\mathbf{Q} = \mathbf{Q}_{v_n} \circ \cdots \circ \mathbf{Q}_{v_1}$ and we use this to define the result of $\mathbf{Q}$.

Definition 20: The result of the query $\mathbf{Q}$ on $\mathbf{I}$ is the result of applying the queries $\mathbf{Q}_{v_1}, \ldots, \mathbf{Q}_{v_n}$ successively to $\mathbf{I}$.

Lemma 9: The result of $\mathbf{Q}$ is unique up to isomorphism. In other words, different choices of names at each step yield isomorphic results.

Proof: This is a straightforward application of the first part of Lemma 7. ∎

Theorem 10: Let $\mathbf{I_Q}$ be the result of the query $\mathbf{Q}$ on the instance $\mathbf{I}$.

1. Let v be a node added by $\mathbf{Q}$ and let l be an element of $I_{\mathbf{Q}}(v)$. Then $\models_{\mathbf{I_Q}} \phi_v(l)$.
2. Let v is a node added by $\mathbf{Q}$ and let l_1 and l_2 be two different names in $I(v)$. Then $val(l_1) \neq val(l_2)$, i.e., there is no duplication in the result.
3. $\mathbf{I_Q}$ is a maximal extension of $\mathbf{I}$ to $\mathbf{S_Q}$ that satisfies 1–2. This means that there is no extension $\mathbf{I}^*_{\mathbf{Q}}$ with $I^*_{\mathbf{Q}}(v) \supseteq I_{\mathbf{Q}}(v)$ for all $v \in V_{\mathbf{Q}} - V$ that satisfies 1–2 and such that for at least one v the inclusion is proper.

Proof:

1. Let $\mathbf{Q}^*$ be the query $\mathbf{Q}_{v_k} \circ \cdots \circ \mathbf{Q}_{v_1}$ where $v = v_k$ and let $\mathbf{I}_{\mathbf{Q}^*}$ be the result of $\mathbf{Q}^*$. By Lemma 7, $\models_{\mathbf{I}_{\mathbf{Q}^*}} \phi_v(l)$. It is easy to see that $\mathbf{I_Q}$ is an extension of an isomorphic image of $\mathbf{I}_{\mathbf{Q}^*}$ and that extending $\mathbf{I}_{\mathbf{Q}^*}$ to $\mathbf{I_Q}$ does not affect the satisfaction of ϕ_v.
2. Obvious.
3. Assume that such an $\mathbf{I}^*$ exists. Let $v = v_k$ be the first of the nodes v_1, $\ldots$, v_n for which $I^*_{\mathbf{Q}}(v) \neq I_{\mathbf{Q}}(v)$ and let $\mathbf{Q}^*$ be the query $\mathbf{Q}_{v_k} \circ \cdots \circ \mathbf{Q}_{v_1}$. From 1 and 2 it follows immediately that both $\mathbf{I_Q}$ and $\mathbf{I}^*_{\mathbf{Q}}$ restricted to $\mathbf{S}_{\mathbf{Q}^*}$ are results of $\mathbf{Q}^*$. Lemma 9 then implies that $\mathbf{I}^*_{\mathbf{Q}}$ and $\mathbf{I_Q}$ are isomorphic, a contradiction. ∎

5.2. Safe Queries

In the previous section we observed that the set of candidate values at a query node can be infinite in principle, even over finite instances. For example:

Example 10: $\phi_{u'}(x_{u'})$ is ($x_{u'} \neq_v$ "David"). This query is unsafe since the set of candidate data values is $R = D - \{\text{David}\}$, an infinite set.

Definition 21: A query $\mathbf{Q}$ on a schema $\mathbf{S}$ is *safe on* a finite instance $\mathbf{I}$ of $\mathbf{S}$ if the set of candidate data values at each node is finite. $\mathbf{Q}$ is *safe* if it is safe on every finite instance $\mathbf{I}$ of $\mathbf{S}$.

Let v be a query node. Assume that we have defined the result of $\mathbf{Q}$ on all the nodes that precede v, and that each of these preceding nodes contains a finite set of names. If $\mu(v) = \otimes$, $\circledast$ or $\circledcirc$, the set of candidate values for v is a subset of, respectively, the Cartesian product, powerset or union of the instances of v's children and therefore is finite. The only case when the set of candidate values may be infinite is when $\mu(v) = \square$, since the set D of data values is infinite.

Lemma 11: $\mathbf{Q}$ is safe on $\mathbf{I}$ iff for every query node of type $\square$ the set of candidate data values for v is finite. ∎

The following two examples use the database and query schema of Fig. 13 and the database instance of Fig. 8.

Example 11: $\phi_{u'}(x_{u'})$ is $(\exists y_u)(x_{u'} =_v y_u) \vee (x_{u'} =_v$ "Absalom"$)$. This query is safe since the set of candidate data values is $R = \{$ Jesse, David, Batsheba, Solomon, Rehoboam, Absalom$\}$.

Testing safety of relational queries can be reduced to testing safety of LDM queries [Kup85]. As a consequence, testing for safety is, in general, undecidable. It is decidable, however, to test whether a given query $\mathbf{Q}$ on a schema $\mathbf{S}$ is safe on a given instance $\mathbf{I}$. We now describe the decision procedure.

Lemma 12: Let $w_1, \ldots, w_n$ be all of the nodes of the database schema $\mathbf{S}$ that are of type $\square$, and let $\{d_1, \ldots, d_k\}$ be the constants that occur in the formula of $\mathbf{Q}$. $\mathbf{Q}$ is safe on $\mathbf{I}$ iff for each query node v of type $\square$, each candidate value for v is either a) the value of an element of some $I(w_i)$ or b) one of the d_j's.

Proof: One direction is obvious—if this condition holds then $\mathbf{Q}$ is safe on $\mathbf{I}$. We prove the converse by induction on the query nodes $V_{\mathbf{Q}} - V = \{v_1, \ldots, v_n\}$ where $v_1 \prec \cdots \prec v_n$. Let $v = v_i$ be a query node of type $\square$, and assume that the lemma holds for the nodes that precede v_i and that $\mathbf{Q}$ is safe on $\mathbf{I}$. Let $\mathbf{I}_{i-1}$ be the result of $\mathbf{Q}_{v_{i-1}}$.

Since $\mathbf{Q}_v$ is safe on $\mathbf{I}$ the set of candidate data values for v is a finite set R. We have to show that

$$R \subseteq \{d_1, \ldots, d_k\} \cup \bigcup_{\substack{\mu(w) = \Box \\ w \in V}} I(w)$$

Call the right hand of this equation S. If the lemma is false, then there is some element val in $R - S$. By the induction hypothesis

$$S = \{d_1, \ldots, d_k\} \cup \bigcup_{\substack{\mu(w) = \Box \\ w \in V \text{ or} \\ w \in V_{\mathbf{Q}}, w \prec v}} I_{i-1}(w)$$

Since val is a candidate data value for v, if we extend $\mathbf{I}_{i-1}$ to an instance $\mathbf{I}_i^1$ of $\mathbf{S}_{\mathbf{Q}_v}$ by defining $I_i^1(v) = \{l\}$ and $val_1(l) = val$, we have $\models_{\mathbf{I}_i^1} \phi_v(l)$. Let val' be an arbitrary element of $D - S$, and extend $\mathbf{I}_{i-1}$ to an instance $\mathbf{I}_i^2$ of $\mathbf{S}_{\mathbf{Q}_v}$ by defining $I_i^2(v) = \{l\}$ and $val_2(l) = val'$. Since val and val' do not appear either in the database, in preceding query nodes, or in the query formulas, an induction on the size $\phi_v(x_v)$ shows that

$$\models_{\mathbf{I}_i^1} \phi_v(l) \Leftrightarrow \models_{\mathbf{I}_i^2} \phi_v(l)$$

The key point in the induction is that x_v can occur in $\phi_v(x_v)$ only in

1. Atomic formulas of the form $x_v =_v d_j$ and $x_v =_v y_w$, where w is a node of type $\Box$ that is either in V or is one of the nodes $v_1, \ldots, v_{i-1}$. Such a formula is false whenever the data value of x_v is not in S.

2. x_v are $x_v =_n x_v$ and $x_v =_v x_v$. These formulas are always true.

We have therefore shown that all the elements of the infinite set $D - S$ are candidate values, contradicting the assumption that $\mathbf{Q}$ is safe on $\mathbf{I}$. ∎

The technique of the proof gives us an effective procedure for determining whether a simple query $\mathbf{Q}_v$ is safe on a finite instance $\mathbf{I}$. Take some data value $d_0 \in D$ that does not occur anywhere in the database or in the query formulas. Test if d_0 is a candidate value (it is not difficult to see that this can be done effectively). In a similar way to the proof of this lemma, we can show that $\mathbf{Q}_v$ is safe on $\mathbf{I}$ iff d_0 is not a candidate value for v. Intuitively, if some such d_0 is in the result, the result is infinite since d_0 cannot be distinguished from any other such data value. This proves the following theorem.

Theorem 13: Let $\mathbf{Q}$ be a query on $\mathbf{S}$ and let $\mathbf{I}$ be a finite instance of $\mathbf{S}$. There is a decision procedure to test whether $\mathbf{Q}$ is safe on $\mathbf{I}$. If $\mathbf{Q}$ is safe on $\mathbf{I}$, then the result can be computed effectively. ∎

6. The Algebraic Query Language

6.1. The Algebraic Operators

In this section we define a set of algebraic operators. We then show that any safe logical query is equivalent to some sequence of algebraic operations, and, conversely, each sequence of algebraic operations is equivalent to a safe logical query.

Since a logical query adds some nodes to the database schema and leaves the instance of the database schema unchanged, each algebraic operator must do the same. Therefore, a selection operator, for example, should not delete tuples that do not satisfy the selection condition, but should rather create a copy of the database node. That copy should contain only those tuples that satisfy the condition. In fact this copying of tuples is what is really done in the relational model—a query does not throw away those tuples in the database that do not meet a selection condition, but rather copies those tuples that do. This issue is not addressed explicitly in relational database theory, since the theory does not deal with what happens to temporary relations that are created while computing the result of a query. As we shall see in the next section, it will be necessary to delete nodes, in certain circumstances. We would still like to make the deletions explicit, rather that hide them in other algebraic operations.

In this section $\mathbf{S}$ will be a database schema with instance $\mathbf{I}$. The algebra consists of operations of the form $w \leftarrow \alpha(v_1, \ldots, v_n)$. Here α is the name of the operator, and its arguments $v_1, \ldots, v_n$ are nodes in the schema $\mathbf{S}$. α adds the node w to the schema, and extends $\mathbf{I}$ to the new schema. We define each operator as a simple logical query, by giving

1. The types of its arguments.

2. The type of w and the list of its children.

3. An LDM formula $\phi_w(x_w)$ that specifies the contents of $I(w)$.

6.1.1. Operators that Copy and Combine Existing Nodes

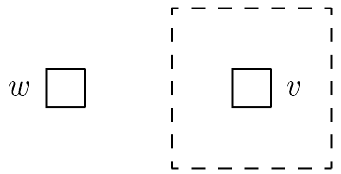

Figure 17: The algebraic operation $w \leftarrow \Box(v)$

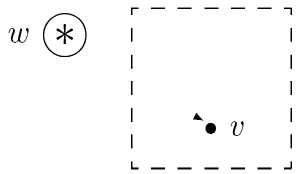

Figure 18: The algebraic operation $w \leftarrow \circledast(v)$

1. $w \leftarrow \Box(v)$ creates a copy of the node v, as is shown in Fig. 17. (In all these figures the schema **S** is shown in the box on the right, and the node that is created by the operation is on the left.) For each distinct data value in $I(v)$, $I(w)$ contains exactly one name with this data value. Note that duplication in $I(v)$ is eliminated in $I(w)$.

    (a) v is a node of **S** that has type $\Box$.

    (b) w is of type $\Box$.

    (c) $\phi_w(x_w)$ is $(\exists y_v)(x_w =_v y_v)$.

2. $w \leftarrow \Box(d)$ creates a node of type $\Box$ that contains just the data value d.

    (a) d is a data value in the data domain D.

    (b) w is of type $\Box$.

    (c) $\phi_w(x_w)$ is $x_w =_v d$.

3. $w \leftarrow \circledast(v)$ creates a node that contains all finite subsets of $I(v)$ (see Fig. 18).

    (a) v is any node in the schema **S**.

    (b) w is of type $(\circledast, v)$.

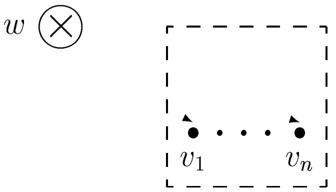

Figure 19: The algebraic operation $w \leftarrow \bigotimes(v_1, \ldots, v_n)$

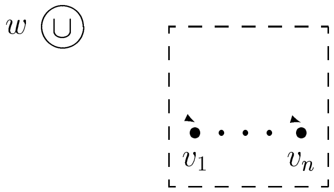

Figure 20: The algebraic operation $w \leftarrow \uplus(v_1, \ldots, v_n)$

   (c) $\phi_w(x_w)$ is T (i.e. always true).

4. $w \leftarrow \bigotimes(v_1, \ldots, v_n)$ creates a node that contains the Cartesian product $I(v_1) \times \cdots \times I(v_n)$ (see Fig. 19).
   (a) $v_1, \ldots, v_n$ are any n nodes in the schema $\mathbf{S}$.
   (b) w is of type $(\bigotimes, n, v_1, \ldots, v_n)$.
   (c) $\phi_w(x_w)$ is T.
5. $w \leftarrow \uplus(v_1, \ldots, v_n)$ creates a node that contains the disjoint union $I(v_1) \cup \cdots \cup I(v_n)$ (see Fig. 20).
   (a) $v_1, \ldots, v_n$ are n distinct nodes of the schema $\mathbf{S}$.
   (b) w is of type $(\uplus, n, v_1, \ldots, v_n)$.
   (c) $\phi_w(x_w)$ is T.

6.1.2. Selection Operators The LDM algebra has two selection operators.

1. The operation $w \leftarrow \sigma_{i\ \theta\ j}(v)$ is similar to the selection operation in the relational algebra. This operator selects those tuples in v whose i^{th} and j^{th} components are related by θ (see Fig. 21).
   (a) v is a node of $\mathbf{S}$ of type $(\bigotimes, n, v_1, \ldots, v_n)$ and $i\ \theta\ j$ is one of the relations $i \in j$, $i\ \pi_t\ j$, $i\ \rho\ j$, $i =_n j$ and $i =_v j$.
   (b) w is of type $(\bigotimes, n, v_1, \ldots, v_n)$.

(c) $\phi_w(x_w)$ is

$$(\exists x_v)(\exists y_{v_i})(\exists y_{v_j})(y_{v_i}\ \pi_{v_i}\ x_v \wedge y_{v_j}\ \pi_{v_j}\ x_v \wedge y_{v_i}\ \theta\ y_{v_j} \wedge x_v =_v x_w)$$

Alternatively, the selection condition may be of the form $i =_v d$ where d is a data value in D. Then $\phi_w(x_w)$ is $(\exists x_v)(\exists y_{v_i})(y_{v_i}\ \pi_{v_i}\ x_v \wedge y_{v_i} =_v d \wedge x_v =_v x_w)$.

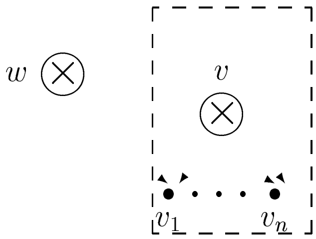

Figure 21: The algebraic operation $w \leftarrow \sigma_{i\ \theta\ j}(v)$

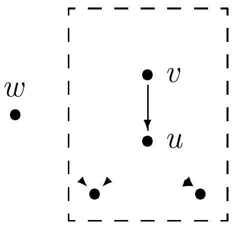

Figure 22: The algebraic operation $w \leftarrow \sigma_{\mathrm{in}}(u, v)$

2. $w \leftarrow \sigma_{\mathrm{in}}(u, v)$. Here u is a child of v, and w will contain those objects of $I(u)$ that actually appear in $I(v)$ (see Fig. 22). For example, if v is to type $\circledast$, each element of $I(v)$ will be a set of elements from $I(u)$. The result of $\sigma_{\mathrm{in}}(u, v)$ will select from $I(v)$ those elements that are in at least one of these sets.

   (a) u and v are nodes of **S** and u is a child of v.

   (b) w is of the same type as u and has the same children.

   (c) $\phi_w(x_w)$ depends on the type of v. Note that v cannot be of type $\square$ since it has a child u.

      i. If v is of type $\otimes$ with u as its i^{th} child then $\phi_w(x_w)$ is

$$(\exists x_u)(\exists x_v)(x_u =_v x_w \wedge x_u\ \pi_i\ x_v)$$

      (Strictly speaking, $\sigma_{\mathrm{in}}(u, v)$ is under-specified here, in case there are multiple edges from v to u, since we have to specify the edge to which we are referring. In this case we shall use the notation $\sigma_{\mathrm{in}}(u, v, i)$ to mean: use the i^{th} edge with tail v.)

ii. If v is of type $\circledcup$, then $\phi_w(x_w)$ is

$$(\exists x_u)(\exists x_v)(x_u =_v x_w \wedge x_u \ \rho \ x_v)$$

iii. If v is of type $\circledast$, then $\phi_w(x_w)$ is

$$(\exists x_u)(\exists x_v)(x_u =_v x_w \wedge x_u \in x_v)$$

6.1.3. Union, Difference and Projection

1. The union operator is similar to the relational union. The syntax we use is $w \leftarrow \cup(v_1, \ldots, v_n)$ (see Fig. 23).

   (a) $v_1, \ldots, v_n$ are n nodes of $\mathbf{S}$ that are of the same type and have the same children.

   (b) w has the same type and the same children as the v_i's.

   (c) $\phi_w(x_w)$ is $(\exists x_{v_1})(x_{v_1} =_v x_w) \vee \cdots \vee (\exists x_{v_n})(x_{v_n} =_v x_w)$.

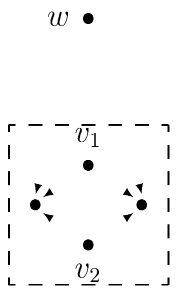

Figure 23: The algebraic operation $w \leftarrow \cup(v_1, v_2)$

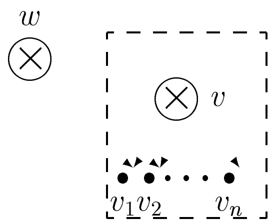

Figure 24: The algebraic operation $w \leftarrow \Pi_{\{v_1, v_2\}}(v)$

2. For difference we shall use infix notation, i.e., we shall write $w \leftarrow v_1 - v_2$ rather than $-(v_1, v_2)$.

   (a) v_1 and v_2 are nodes of $\mathbf{S}$ that are of the same type and have the same children.

(b) w has the same type and the same children as v_1 and v_2.

(c) $\phi_w(x_w)$ is $(\exists x_{v_1})(x_{v_1} =_v x_w) \wedge (\forall x_{v_2})(x_{v_2} \neq_d x_w)$.

3. The projection operation is similar to projection in the relational algebra. The syntax we use is $w \leftarrow \Pi_A(v)$, where A is an ordered multiset of edges with tail v.

   (a) v is a node of $\mathbf{S}$ of type $(\otimes, n, v_1, \ldots, v_n)$ and A is an ordered multiset of edges with tail v.

   (b) Let $A = \{e_1, \ldots, e_k\}$ where e_j is the edge (v, v_{i_j}). Then w is of type $(\otimes, k, v_{i_1}, \ldots, v_{i_k})$.

   (c) $\phi_w(x_w)$ is

$$(\exists x_v)(\exists x_{v_1}) \cdots (\exists x_{v_n})\Big(x_{v_{i_1}}\ \pi_1\ x_w \wedge \cdots \wedge (x_{v_{i_k}}\ \pi_k\ x_w) \wedge \\ x_v =_v (x_{v_1}, \ldots, x_{v_k})\Big)$$

   When it does not cause any ambiguity, we shall use a set A of nodes rather than of edges, as in Fig. 24.

6.2. Equivalence of the Logical and Algebraic Query Languages

We now use these algebraic operators to define an algebraic query language. An algebraic query is a sequence $\{\alpha_1, \ldots, \alpha_n\}$, where each α_i is an algebraic operator on the result of α_{i-1}. We would like to be able to show that this query language is equivalent to the logical query language. In other words, for each logical query on a schema $\mathbf{S}$, there should exist a sequence of algebraic operations, and vice versa, with the property that the schemas created by these two queries are identical, and for every database instance $\mathbf{I}$, the results are isomorphic relative to $\mathbf{S}$. Unfortunately, as the next example shows, this is not quite true.

Example 12: Let $\mathbf{S}$ consist of a node u of type $\Box$ and let $\mathbf{Q}$ be the logical query that adds a node v of type $\Box$ to $\mathbf{S}$. Let d_1 and d_2 be two data values, and let $\phi_v(x_v)$ be $(x_v =_v d_1 \vee x_v =_v d_2)$. The candidate values for v are then $\{d_1, d_2\}$. There is no algebraic query equivalent to $\mathbf{Q}$. If there was such a query, it would consist of one algebraic operation alone, since each operator adds a new node to the schema. By inspection we can see that no single algebraic operator is equivalent to **Q**.

How can we modify the definition to get an equivalent query? If $\mathbf{Q}_A$ is the algebraic query that consists of the operations $w_1 \leftarrow \Box\,(d_1)$, $w_2 \leftarrow \Box\,(d_2)$ and $v \leftarrow \cup(w_1, w_2)$, it is clear that the instance of v is what we are after. If we were then to restrict the result of the query to the schema that consists of the nodes u and v we get the instance we want. We have essentially used the two nodes w_1 and w_2 for temporary storage while computing the result of the query. In fact the same thing occurs in the relational model, since temporary relations are used there for subexpressions and then deleted at the end. It is therefore reasonable to expect the same thing to happen in the logical data model.

To be able to use temporary nodes, we extend the algebraic query language by adding a "delete" operator. This operator deletes a node from the schema and restricts the instance of the original schema to the new schema. We have to make sure that we never delete a node that is the child of some other node, since in that case the result would not be a legal schema. The operator that deletes the node v will be written $D(v)$.

Definition 22: Let **S** be an LDM schema with instance **I**. The algebraic operator $D(v)$ is legal when v is a node with no parent. The result of $D(v)$ is the schema $\mathbf{S}'$ that consists of deleting v from **S**, together with the instance that we get by restricting **I** to $\mathbf{S}'$.

In the algebraic query language we must take care not to delete database nodes, i.e., we must only allow the user to delete nodes that have been constructed by his query. We shall call the language with the deletion operator the *extended algebraic query language.*

Definition 23: Let **S** be an LDM schema. An *extended algebraic query* on **S** is a sequence $\mathbf{Q}_A = (\alpha_1, \ldots, \alpha_n)$ where each α_i is either

1. An operation of the form $w_i \leftarrow \beta_i(v_i^1, \ldots, v_i^{i_j})$, where β_i is an algebraic operator other than the deletion operator and $v_i^1, \ldots, v_i^{i_j}$ are either node of **S** or are nodes that were created by some previous β_j and have not been deleted.

2. The operator $D(v_i)$, where v_i is a node that was created by a previous algebraic operator in the sequence $\beta_1, \ldots, \beta_{i-1}$ and has not yet been deleted.

Definition 24: Let $\mathbf{Q}_A$ be an extended algebraic query on $\mathbf{S}$, and let $\mathbf{Q}_B$ be an extended algebraic query on the result of $\mathbf{Q}_A$. The query $\mathbf{Q}_B \circ \mathbf{Q}_A$ is the *composition* of $\mathbf{Q}_A$ and $\mathbf{Q}_B$, formed simply by concatenating the lists of algebraic operators.

Obviously, the delete operator itself is not equivalent to any logical query, since every logical query adds nodes to the schema. This by itself does not necessarily mean that we cannot find a logical query equivalent to any extended algebraic query. After all, an extended algebraic query does not delete database nodes, since the only nodes that are deleted are those that were constructed by previous algebraic operations. It might still be the case (as in Example 12) that there is an equivalent logical query that does not use temporary nodes. Nevertheless, in [Kup85] it is shown that the extended algebraic query language is strictly more powerful than the logical query language. In order to get an equivalent query language, we modify the logical language to include temporary nodes as well.

Definition 25: Let $\mathbf{S}$ be an LDM schema. An *extended logical query* on $\mathbf{S}$ is a tuple $\mathbf{Q} = \langle S_\mathbf{Q}, \Phi_\mathbf{Q}, \prec_\mathbf{Q}, D_\mathbf{Q} \rangle$ where

1. $\langle S_\mathbf{Q}, \Phi_\mathbf{Q}, \prec_\mathbf{Q} \rangle$ is a logical query on $\mathbf{S}$.
2. $D_\mathbf{Q}$ is the set of temporary nodes used in the query. $D_\mathbf{Q}$ is a subset of the query nodes $V_\mathbf{Q} - V$ that we can delete and still get an LDM schema. In other words, there is no edge with tail outside $D_\mathbf{Q}$ and head in $D_\mathbf{Q}$, i.e., if $(e_1, e_2) \in E_\mathbf{Q}$ and $e_2 \in D_\mathbf{Q}$, then $e_1 \in D_\mathbf{Q}$.

Definition 26: Let $\mathbf{Q}$ be the extended logical query $\langle S_\mathbf{Q}, \Phi_\mathbf{Q}, \prec_\mathbf{Q}, D_\mathbf{Q} \rangle$, and let $\mathbf{I}$ be an instance of $\mathbf{S}$. The result of this query consists of

1. The schema $\mathbf{S}^*_\mathbf{Q}$ consisting of
   (a) The nodes in $V_\mathbf{Q} - D_\mathbf{Q}$.
   (b) The relevant edges, i.e., all those edges of $\mathbf{S}_\mathbf{Q}$ whose head and tail are both in $V_\mathbf{Q} - D_\mathbf{Q}$.

(c) The restriction of the type assignment μ to $V_{\mathbf{Q}} - D_{\mathbf{Q}}$.

2. The result of $\mathbf{Q}$ on $\mathbf{I}$ is defined as follows. Let $\mathbf{I}_{\mathbf{Q}}$ be the result of $\langle \mathbf{S}_{\mathbf{Q}}, \Phi_{\mathbf{Q}}, \prec_{\mathbf{Q}} \rangle$ on $\mathbf{I}$. The result of $\mathbf{Q}$ on $\mathbf{I}$ is then the restriction of $\mathbf{I}_{\mathbf{Q}}$ to $\mathbf{S}^*_{\mathbf{Q}}$.

We now show that every extended algebraic query is equivalent to some extended logical query.

Lemma 14: Let $\mathbf{Q}_A = (\alpha_1, \ldots, \alpha_n)$ be an extended algebraic query on $\mathbf{S}$. There exists a safe extended logical query $\mathbf{Q}_L$ on $\mathbf{S}$ such that for every instance $\mathbf{I}$ of $\mathbf{S}$, the results of $\mathbf{Q}_A$ and $\mathbf{Q}_L$ on $\mathbf{I}$ are isomorphic relative to $\mathbf{S}$.

Proof: The schema of $\mathbf{Q}_L$ consists of all those nodes that are created by the operations in the query $\mathbf{Q}_A$. The set of temporary nodes $D_{\mathbf{Q}_L}$ is the set of nodes deleted in $\mathbf{Q}_A$, i.e., $\{v_i \mid \text{ The operator } \alpha_i \text{ is } D(v_i)\}$. Since we are only allowed to delete nodes that are not in $\mathbf{S}$ and that have no parent, it is easy to see that there is no edge whose head is in $D_{\mathbf{Q}_L}$ and whose tail is not. Each α_i that is not a delete operator must be of the form $w_j \leftarrow \beta_j(w_i^1, \ldots, w_i^{i_j})$. We define an order on the nodes of $V_{\mathbf{Q}} - V$ as follows: $w_i \prec w_j$ whenever $i < j$. $\phi_{w_i}(x_{w_i})$ is the formula that was used to define the operator β_j. It is easy to verify that the results of $\mathbf{Q}_A$ and $\mathbf{Q}_L$ on any instance $\mathbf{I}$ are indeed isomorphic. ∎

To show the converse, let $\mathbf{Q}_L$ be a logical query on $\mathbf{S}$. Let $\mathbf{I}$ be a fixed instance of $\mathbf{S}$ such that $\mathbf{Q}_L$ is safe on $\mathbf{I}$. The definition of $\mathbf{Q}_A$ will not depend on $\mathbf{I}$, but the results of $\mathbf{Q}_A$ and $\mathbf{Q}_L$ will only be isomorphic on those instances of $\mathbf{S}$ on which $\mathbf{Q}_L$ is safe. We keep $\mathbf{I}$ fixed so that we will be able to prove various lemmas about the results of the queries as we go along.

We first look at the case when $\mathbf{Q}_L$ is a simple query $\mathbf{Q}_w$. We start by creating a node w_{dom}, that contains the "domain" of w, i.e., all those objects that might be candidate values for w, if we were to ignore everything except the type of w and the fact that $\mathbf{Q}_L$ is safe on $\mathbf{I}$. We define w_{dom} as follows.

1. If w is of type $\Box$, let $v_1, \ldots, v_t$ be all the nodes in $\mathbf{S}$ that are of type $\Box$ and let $d_1, \ldots, d_k$ be the constants that occur in $\phi_w(x_w)$. Define

w_{dom} by the algebraic query:

$$\begin{aligned}
&s_1 \leftarrow \square(v_1)\\
&\quad\vdots\\
&s_t \leftarrow \square(v_t)\\
&s_{t+1} \leftarrow \square(d_1)\\
&\quad\vdots\\
&s_{t+k} \leftarrow \square(d_k)\\
&w_{\text{dom}} \leftarrow \cup(s_1, \ldots, s_{t+k})\\
&D(s_1)\\
&\quad\vdots\\
&D(s_{t+k})
\end{aligned}$$

2. If $\mu(w) = (\otimes, k, v_1, \ldots, v_k)$ define w_{dom} by $w_{\text{dom}} \leftarrow \otimes(v_1, \ldots, v_k)$.
3. If $\mu(w) = (\circledast, v)$ define w_{dom} by $w_{\text{dom}} \leftarrow \circledast(v)$.
4. If $\mu(w) = (\circledcup, k, v_1, \ldots, v_k)$ define w_{dom} by $w_{\text{dom}} \leftarrow \circledcup(v_1, \ldots, v_k)$.

Call this algebraic query $\mathbf{Q}_{\text{dom}}$.

Lemma 15:

1. The schema created by $\mathbf{Q}_{\text{dom}}$ is equal to the schema of $\mathbf{S}$ together with a node w_{dom} of the same type and with the same children as the node w in the original logical query $\mathbf{Q}_w$.
2. Let $\mathbf{I}_{\text{dom}}$ be the result of $\mathbf{Q}_{\text{dom}}$ on $\mathbf{I}$ and let $\mathbf{I}_w$ be the result of $\mathbf{Q}_w$ on $\mathbf{I}$. If *val* is a value of an object in $I_w(w)$, then *val* is also a value of an object in $\mathbf{I}_{\text{dom}}(w_{\text{dom}})$.

Proof: If w is of type $\otimes$, $\circledcup$ or $\circledast$, the lemma is obvious. If w is of type $\square$, the first part follows from the fact that all the nodes except w_{dom} that are created by $\mathbf{Q}_{\text{dom}}$ are also deleted by $\mathbf{Q}_{\text{dom}}$. The second part is an immediate consequence of Lemma 12 and the definition of $\mathbf{Q}_{\text{dom}}$. ∎

We may assume, if necessary by renaming some bound variables, that all the bound variables in the formula $\phi_w(x_w)$ are distinct. Let these variables be $x^1_{w_1}, \ldots, x^k_{w_k}$. The algebraic query $\mathbf{Q}_{\text{prod}}$ on the result of $\mathbf{Q}_{\text{dom}}$ consists of the operation

$$w_{\text{prod}} \leftarrow \bigotimes(w_1, \ldots, w_k, w_{\text{dom}})$$

For the purpose of defining $\mathbf{Q}_A$ we shall label the edges with tail w_{prod} as follows. The i^{th} edge with tail w_{prod} will be labeled $x^i_{w_i}$. These labels are used only to define the algebraic query, and are not themselves part of the query. Their purpose is to tell us which bound variable the edge corresponds to.

In certain cases, when we create a new node using some algebraic operation, the outgoing edges from the new node will inherit the labels of the corresponding edges whose head is one of the arguments of the operator. We only use this inheritance in cases when it is unambiguous, i.e., in cases when all the arguments have the same labeling. The operations for which labels will be inherited are $\sigma_{i\ \theta\ j}$, difference and union. When we use the projection operation the new edges will also inherit the labeling of the corresponding edges whose head is the argument of the projection.

Arrange all the well-formed subformulas of $\phi_w(x_w)$ in a list $\psi_1, \ldots, \psi_m$, where $\psi_m = \phi_w(x_w)$ and ψ_i precedes ψ_j whenever it is a subformula of ψ_j. For each such subformula, we define an extended algebraic query $\mathbf{Q}_{\psi_i}$ on the result of $\mathbf{Q}_{\psi_{i-1}}$. $\mathbf{Q}_{\psi_1}$ is a query on the result of $\mathbf{Q}_{\text{prod}}$. The node w_{ψ_i} is of type $(\bigotimes, k+1, w_{j_1}, \ldots, w_{j_k}, w_{\text{dom}})$, and contains, intuitively, those tuples $(l_1, \ldots, l_k, l_d)$ for which $\models_{\mathbf{I}_{\psi_i}} \psi_i(l_1, \ldots, l_k, l_d)$. The edges with tail w_ψ will be labelled with the variables that might be free in ψ—i.e., those that haven't yet been bound by ψ. The definition of $\mathbf{Q}_{\psi_i}$ is as follows.

1. ψ_i is $x^a_{w_a}\ \theta\ x^b_{w_b}$. $\mathbf{Q}_{\psi_i}$ is $w_{\psi_i} \leftarrow \sigma_{a\ \theta\ b}(w_{\text{prod}})$
2. ψ_i is $x^a_{w_a}\ \theta\ x_w$. $\mathbf{Q}_{\psi_i}$ is $w_{\psi_i} \leftarrow \sigma_{a\ \theta\ k+1}(w_{\text{prod}})$
3. ψ_i is $x_w\ \theta\ x_w$. $\mathbf{Q}_{\psi_i}$ is $w_{\psi_i} \leftarrow \sigma_{k+1\ \theta\ k+1}(w_{\text{prod}})$
4. ψ_i is $x^a_{w_a} =_v d$. $\mathbf{Q}_{\psi_i}$ is $w_{\psi_i} \leftarrow \sigma_{a =_v d}(w_{\text{prod}})$
5. ψ_i is $x_w =_v d$. $\mathbf{Q}_{\psi_i}$ is $w_{\psi_i} \leftarrow \sigma_{(k+1) =_v d}(w_{\text{prod}})$
6. ψ_i is $\psi_{j_1} \vee \psi_{j_2}$. Let A_1 be the (ordered multi)set of edges with tail $w_{\psi_{j_1}}$ that have the same label as some edge with tail $w_{\psi_{j_2}}$. Let A_2 be the

corresponding set of edges with tail $w_{\psi_{j_2}}$. $\mathbf{Q}_{\psi_i}$ is the following extended algebraic query

$$\begin{array}{l} s_1 \leftarrow \Pi_{A_1}(w_{\psi_{j_1}}) \\ s_2 \leftarrow \Pi_{A_2}(w_{\psi_{j_2}}) \\ w_{\psi_i} \leftarrow \cup(s_1, s_2) \\ D(s_1) \\ D(s_2) \end{array}$$

(s_1 and s_2 are different temporary nodes from those used above, and from similarly named nodes used below.) Note that the way we defined A_1 and A_2 guarantees that there is no ambiguity in labeling the edges of the result, at least as long as the labels of the edges in A_1 and A_2 are in the same order. We shall show later that this is indeed the case.

7. ψ_i is $\neg\psi_j$. Let A be the (ordered multi)set of edges with tail w_{dom} that have the same label as some edge with tail w_{ψ_j}. $\mathbf{Q}_{\psi_i}$ is the following extended algebraic query

$$\begin{array}{l} s_1 \leftarrow \Pi_A(w_{\psi_{\text{dom}}}) \\ w_{\psi_i} \leftarrow s_1 - w_{\psi_j} \\ D(s_1) \end{array}$$

As in the previous case we shall show that we can label the edges with tail w_{ψ_i} without any ambiguity.

8. ψ_i is $(\exists x^a_{w_a})(\psi_j)$. Let A be the (ordered multi)set of all edges with tail w_{ψ_j} except for the edge labeled $x^a_{w_a}$. We show below that there must be exactly one edge with this label. $\mathbf{Q}_{\psi_i}$ then consists of the algebraic operation $w_{\psi_i} \leftarrow \Pi_A(w_{\psi_j})$.

Lemma 16: Let $\psi = \psi_i$ be one of these well formed subformulas of $\phi_w(x_w)$. Let $x^{a_1}_{w_{a_1}}, \ldots, x^{a_j}_{w_{a_j}}$ be those variables in the above list that are not bound in ψ_i. Note that some of the $x^{a_i}_{w_{a_i}}$'s may not actually occur in ψ_i. Then w_{ψ_i} is of type $(\otimes, j+1, w_{a_1}, \ldots, w_{a_j}, w_{\text{dom}})$. Furthermore, the t^{th} edge with tail w_{ψ_i} has head w_{a_t} and is labeled with the variable $x^{a_t}_{w_{a_t}}$. As a consequence of this, all the labelings of edges are in the same order, and the assumptions that we made on the labelings when we defined the w_{ψ_i}'s hold.

Proof: The proof is a fairly straightforward induction using the definition of w_{ψ_i}. The tricky case is when ψ_i is $\psi_{j_1} \vee \psi_{j_2}$. Then the children of $w_{\psi_{j_1}}$ correspond to the bound variables of ϕ_w that are not bound in ψ_{j_1} and the children of $w_{\psi_{j_2}}$ to the bound variables of ϕ_w not bound in ψ_{j_2}. Since a variable is not bound in $\psi_{j_1} \vee \psi_{j_2}$ iff it is not bound in ψ_{j_1} and it is not bound in ψ_{j_2}, we see that the result does hold in this case[4]. ∎

Lemma 17: Let w_{ψ_i} be of type $(\bigotimes, j, w_{j_1}, \ldots, w_{j_k}, w_{\text{dom}})$. Let $\mathbf{I}_{\psi_i}$ be the result of $\mathbf{Q}_{\psi_i}$ on $\mathbf{I}$, let l_d be a member of $I_{\psi_i}(w_{\text{dom}})$ and let l_t be a member of $I_{\psi_i}(w_{j_t})$ for $t = 1, \ldots, k$. Then there exists an l in $I_{\psi_i}(w_{\psi_i})$ with $val(l) = (l_1, \ldots, l_k, l_d)$ if and only if $\models_{\mathbf{I}_{\psi_i}} \psi_i(l_1, \ldots, l_k, l_d)$. Intuitively, $(l_1, \ldots, l_k, l_d)$ is a candidate value iff it satisfies ψ_i.

Proof: A straightforward induction on the structure of ψ_i. ∎

The extended algebraic query $\mathbf{Q}_{\text{final}}$ on the result of $\mathbf{Q}_{\phi_w}$ consists of

$$\begin{array}{l} w_A \leftarrow \sigma_{\text{in}}(w_{\text{dom}}, w_\phi) \\ D(w_\phi) \\ \quad\vdots \\ D(w_{\psi_1}) \\ D(w_{\text{prod}}) \\ D(w_{\text{dom}}) \end{array}$$

We finally define the algebraic query $\mathbf{Q}_A$ as

$$\mathbf{Q}_{\text{final}} \circ \mathbf{Q}_\phi \circ \mathbf{Q}_{\psi_{m-1}} \circ \mathbf{Q}_{\psi_1} \circ \mathbf{Q}_{\text{prod}} \circ \mathbf{Q}_{\text{dom}}$$

Lemma 18: Let $\mathbf{I}_1$ be the result of $\mathbf{Q}_w$ on $\mathbf{I}$ and let $\mathbf{I}_2$ the result of the algebraic query $\mathbf{Q}_A$ on $\mathbf{I}$. Then $\mathbf{I}_1$ and $\mathbf{I}_2$ are isomorphic relative to $\mathbf{S}$.

Proof: First note that the schemas are equal. The only node created but not deleted by $\mathbf{Q}_A$ is the node w_A. This node is similar to the node w_{dom} and hence to w.

We have to show that the instances of w_A and w are isomorphic, i.e., that at the point in evaluating the queries that we compute the instances of these nodes, they have the same candidate values. We assume that we are at the point in the evaluation of $\mathbf{Q}_A$ just before the final round of deletions.

[4] Note that "not bound" is not the same as "free." A variable that is not bound in one of these formulas is either free in this formula, or does not appear in it at all

Let val be a candidate data value for w. Extend $\mathbf{I}$ to an instance $\mathbf{I}_w$ of $\mathbf{S}_{\mathbf{Q}_w}$ by defining $I_w(w) = \{l\}$ and $val(l) = val$. Then $\models_{\mathbf{I}_w} \phi_w(l)$. Let $\mathbf{I}_{\phi_w}$ be the result of $\mathbf{Q}_{\phi_w}$ on $\mathbf{I}$. By the second clause of Lemma 15, val is a candidate data value for w_{dom} and so for some l_d in $I_{\phi_w}(w_{\mathrm{dom}})$, $val(l_d) = val$. By Lemma 6, $\models_{\mathbf{I}_w} \phi_w(l)$ implies $\models_{\mathbf{I}_{\phi_w}} \phi_w(l_d)$, and therefore, by Lemma 17, for some l_ϕ in $I_{\phi_w}(w_{\mathrm{dom}})$, $val(l_\phi) = val$ is a candidate data value for w_A.

For the converse, suppose that val is a candidate data value for w_A. Let $\mathbf{I}_{\phi_w}$ be the result of $\mathbf{Q}_{\phi_w}$ on $\mathbf{I}$. Since val is a candidate data value for w_A, for some l_ϕ in $I_{\phi_w}(w_{\phi_w})$ and some l_d in $I_{\phi_w}(w_{\mathrm{dom}})$, $val(l_\phi) = (l_d)$ and $val(l_d) = val$. Since l_ϕ is in $I_{\phi_w}(w_{\phi_w})$, Lemma 15 implies that $\models_{\mathbf{I}_{\phi_w}} \phi_w(l_d)$. Restrict $\mathbf{I}_{\phi_w}$ to an instance $\mathbf{I}_{\mathrm{dom}}$ of the schema of $\mathbf{Q}_{\mathrm{dom}}$. Then $\models_{\mathbf{I}_{\mathrm{dom}}} \phi_w(l_d)$, and so by Lemma 6, $val(l_d) = val$ is a candidate data value for w. ∎

Since an arbitrary logical query can be viewed as a sequence of simple queries, we can easily extend the above construction to arbitrary queries by concatenating the algebraic queries for the individual simple queries. If we have an extended logical query we have to add deletion operations to delete those nodes in the delete set of the query. This yields the following theorem.

Theorem 19: The extended algebraic query language and the extended logical query language are equivalent, i.e., for every extended algebraic query on $\mathbf{S}$ there exists a safe extended logical query on $\mathbf{S}$ and for every extended logical query on $\mathbf{S}$ there exists an extended algebraic query on $\mathbf{S}$, such that both queries define the same schema and for every database instance $\mathbf{I}$ on which the logical query is safe, the results of both queries are isomorphic relative to $\mathbf{S}$. ∎

7. Concluding Remarks

We have described a new model of data, the Logical Data Model, that is designed to combine the advantages of the existing data models. On the one hand, it enables the database to describe more of the semantics of the data than is possible using the relational model of data. On the other hand, we do not lose the nice properties that relational databases have, in particular the ability to query the database using equivalent nonprocedural and procedural languages. The complexity of our query language is studied in [KV85, **?**, **?**].

Our work unifies and generalizes a long sequence of previous works on semantic data models [AB84, DB82, FK77, **?**, GDB82, Gra79, Har78, **?**, **?**, Kob80, Mak77, MP82, OY85, RR83, SP82, **?**, SS77, Tsi76]. The preliminary publication of our results in [KV84], stimulated later work, such [**?**, **?**, **?**]. All these models can be viewed as special cases of LDM, designed to make certain classes of queries easier to express, or more efficient to implement.

Our model incorporates two important *object-oriented* features: *object identity*, captured by the distinction between object names and object values, and *strong typing*, our types are built from the product, power, and union operations. It does not, however, incorporate other features considered to be important to object-oriented databases, such as "inheritance", "methods", "encapsulation" etc. (cf. [**?**]). An important feature of our model is the separation of schema and instance, unlike the models in [**?**, **?**].

While one of the feature of LDM is the ability to describe cyclic data, this feature is lacking in the query language. This brings up two questions: is it possible to eliminate cycles from databases? is it possible to add cycles to queries? The first question is addressed in [KV85]. More recently, [**?**] shows how to extend the query language to allow cycles in queries.

The query languages that we have described are based on the paradigm of first-order logic. It is by now recognized that in the framework of the relational model first-order languages are too weak for the task of database querying [AU79]. This motivated the study of more powerful query languages, based on the paradigm of logic programming [**?**, **?**, Rei78, Ull85]. More recently, similar query languages have been developed for models similar to LDM [**?**, **?**, **?**, **?**, **?**, **?**].

Acknowledgments

We are indebted to Jeff Ullman for some of the basic ideas underlying this work. We would also like to thank Rick Hull, Paris Kanellakis, and Dave Maier, for helpful discussions and suggestions, and the anonymous referees for suggestions for improvements.

References

[AB84] S. Abiteboul and N. Bidoit. Non first normal form relations to represent hierarchically organized data. In *Proc. Third Annual ACM Symposium on Principles of Database Systems*, pages 191–200, Waterloo, Ontario, 1984. ACM.

[Acz85] P. Aczel. Notes on non-well-founded sets. Unpublished Manuscript, 1985.

[AU79] A. V. Aho and J. D. Ullman. Universality of data retrieval languages. In *Conference Record of the Sixth Annual ACM Symposium on Principles of Programming Languages*, pages 110–120, 1979.

[Cod70] E. F. Codd. A relational model of data for large shared data banks. *Communications of the ACM*, 13(6):377–387, 1970.

[Cod79] E. F. Codd. Extending the database relational model to capture more meaning. *ACM Transactions on Database Systems*, 4(4):397–434, December 1979.

[DB82] U. Dayal and P. A. Bernstein. On the correct translation of update operations on relational views. *ACM Transactions on Database Systems*, 7(3):381–416, 1982.

[FK77] A. L. Furtado and L. Kerschberg. An algebra of quotient relations. In *Proc. ACM Int'l Conf. on Management of Data*, pages 1–8, Toronto, Ontario, 1977. ACM.

[GDB82] D. Gangopadhyay, U. Dayal, and J. C. Browne. Semantics of network data manipulation languages: An object-oriented approach. In *Proc. Eighth Intl. Conf. on Very Large Data Bases*. IEEE, 1982.

[Gra79] M. H. Graham. NETS: Operations and logic. In F. H. Lochovsky, editor, *A Panache of DBMS Ideas II*, pages 152–179. Tech. Report CSRG-101, Computer Systems Research Group, Univeristy of Toronto, 1979.

[Har78] W. T. Hardgrave. Ambiguity in processing boolean queries on tdms tree-structures: A study of four different philosophies. Technical Report IFSM TR-35, University of Maryland, 1978.

[HY82] R. Hull and C. K. Yap. The format model: A theory of database organization. In *Proc. First Annual ACM Symposium on Principles of Database Systems*, pages 205–211, Los Angeles, CA, 1982. ACM.

[Jac79] B. E. Jacobs. Application of database logic to database design. Technical Report TR–892, University of Maryland at College Park, 1979.

[Jac82] B. E. Jacobs. On database logic. *Journal of the ACM*, 29(2):310–332, 1982.

[Kob80] I. Kobayashi. An overview of the database mangement technology. Technical Report TRCS–4–1, Sanno College, Kanagawa 259–11, 1980.

[Kup85] G. M. Kuper. *The Logical Data Model: A New Approach to Database Logic.* PhD thesis, Stanford University, 1985.

[KV84] G. M. Kuper and M. Y. Vardi. A new approach to database logic. In *Proc. Third Annual ACM Symposium on Principles of Database Systems*, pages 86–96, Waterloo, Ontario, 1984. ACM.

[KV85] G. M. Kuper and M. Y. Vardi. On the expressive power of the logical data model. In *Proc. ACM Int'l Conf. on Management of Data*, pages 180–189, Austin, TX, 1985. ACM.

[Mak77] A. Makinouchi. A consideration on normal form of not-necessarily normalized relations in the relational data model. In *Proc. Third Intl. Conf. on Very Large Data Bases*, pages 447–453, Tokyo, Japan, 1977. IEEE.

[MP82] F. Manola and A. Pirotte. CQLF—A query language for CODASYL-type databases. In *Proc. ACM Int'l Conf. on Management of Data*, pages 94–103, Orlando, FL, 1982. ACM.

[OY85] Z. M. Ozsoyoglu and L.-Y. Yuan. A normal form for nested relations. In *Proc. Fourth Annual ACM Symposium on Principles of Database Systems*, pages 251–260, Portland, OR, 1985. ACM.

[Rei78] R. Reiter. Deductive question answering in relational databases. In H. Gallaire and J. Minker, editors, *Logic and Databases*, pages 147–177. Plenum Press, 1978.

[RR83] M. Rafanelli and F. L. Ricci. A data definition language for a statistical database. Technical Report TR-62, IASI-CNR, July 1983.

[SP82] H.-J. Scheck and P. Pistor. Data structures for an integrated data base management and information retrieval system. In *Proc. Fourth Intl. Conf. on Very Large Data Bases*. IEEE, 1982.

[SS77] J. M. Smith and D. C. P. Smith. Database abstractions: Aggregation and generalization. *ACM Transactions on Database Systems*, 2(2):105–133, 1977.

[TL82] D. C. Tsichritzis and F. H. Lochovsky. *Data Models*. Prentice Hall, Englewood Cliffs, NJ., 1982.

[Tsi76] D. C. Tsichritzis. LSL: A link and selector language. In *Proc. ACM Int'l Conf. on Management of Data*, pages 123–133, Washington, D. C., 1976. ACM.

[Ull82] J. D. Ullman. *Principles of Database Systems*. Computer Science Press, Rockville, MD, 1982.

[Ull85] J. D. Ullman. Implementation of logical query languages for databases. In *Proc. ACM Int'l Conf. on Management of Data*, Austin, TX, 1985. ACM.

[Var83] M. Y. Vardi. Review of [Jac82]. *Zentralblatt für Mathematik*, 497.68061, 1983.