On the Expressive Power of the Logical Data Model

Preliminary Report

GABRIEL M KUPER[1]
Department of Computer Science
Stanford University
Stanford, California

MOSHE Y VARDI[2]
Center for Study of Language and Information
Stanford University
Stanford, California

Abstract

In this paper we study the expressive power of the logical data model *LDM*, introduced in [KV84] We show that even though the logical data model is semantically powerful, it is not overly powerful so as to be intractable. We demonstrate it from three aspects First, we study the complexity of checking integrity constraints, and we show that it is no more difficult than checking integrity constraints in the relational model Secondly, we show that the logic of the model is essentially first-order That means, for example, that one can use a standard theorem-prover either in the database design process or for deductive query answering Finally, we prove the surprising result that the ability to define cycles does not, in a certain sense, add any power to the model Thus, any cyclic schema can be converted to an "equivalent" acyclic schema

[1] Work supported by AFOSR grant 80 0212

[2] Part of the work reported here was done while this author was visiting IBM Research Laboratory at San Jose, California Research at Stanford University was supported by a gift from System Development Foundation

1 Introduction

Until recently most work on database theory has focused on the relational model [Co70], mainly due to its elegance and mathematical simplicity compared to the other models Some of this work has pointed out various disadvantages of the relational model, among them its lack of semantics and the fact that it forces the data to have a flat structure that the real data does not always have [Co79] [ScSw75] [SmSm77]

Several recent papers have addressed this problem by trying to find a more semantically motivated data model, see for example [HM78] and [Shi78] These models, however, do not have the solid mathematical foundation that the relational model has More recently, two general mathematical frameworks were introduced Jacobs [Ja82] describes *database logic*, a mathematical model for databases that generalizes the three principal data models the relational, hierarchical and network models Also, Hull and Yap [Hu82] describe the *format model*, which generalizes the relational and hierarchical models

In [KV84] we pointed out some shortcomings of these models First, both these mod-

els are unsatisfactory in their ability to logically restructure data, i e , the ability to query the database While Hull and Yap ignore the issue of a data sublanguage, Jacobs' treatment is an overkill - his query language enables one to write noncomputable queries [Va83] 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 [Ul82] Note that virtual records introduce cyclicity not only in the schema level but also at the instance level

As an alternative we proposed the *logical data model (LDM)* [KV84] The logical data model unifies and generalizes the three principal data models, including the ability to use virtual records The essential feature of the model is the separation between the *data space* and the *address space* The model also consists of a logic, in which integrity constraints can be specified, and a query facility, consisting of equivalent procedural (calculus-like) and nonprocedural (algebra-like) query languages An attractive feature of this query facility is that answers to queries do not have to be flat, i e , relations, but can have semantically motivated structure as well Thus, for example, the answer to a query over a network database can also have a network structure

In this paper we discuss the expressive power of the logical data model Two components of the model seem to be very powerful the ability to form sets of data and the ability to define cycles Giving the model too much power might be counterproductive, it is the power of database logic [Ja82] that makes it noniecursive [Va83] We show here that even though the logical data model is semantically powerful, it is not overly powerful so as to be intractable We demonstrate this from three aspects First, we study the complexity of checking integrity constraints, and show that it is no more difficult than checking integrity constraints in the relational model (as studied in [Va82]) Secondly, we show that the logic of the model is essentially first-order That means, for example, that one can use a standard theorem-prover either in the database design process [BBG78] [MMSU81] or for deductive query answering [NG78] [Re84]. Finally, we prove the surprising result that the ability to define cycles does not, in a certain sense, add any power to the model Thus, any cyclic schema can be converted to an "equivalent" acyclic schema

The paper is organized as follows Section 2 contains a short description of the logical data model For a fuller exposition the reader is referred to [KV84] Section 3 examines the complexity of checking integrity constrains in the model Section 4 shows that LDM-logic is essentially first-order Finally, Section 5 describes how to eliminate cycles in LDM-schemas

2 The Logical Data Model

2.1 Schemas and Instances

In the *logical data model* (LDM), *schemas* are directed graphs, with a *type* associated with each node The possible types are *basic type*, *composition*, and *collection* (there is a fourth type, *classification*, which we ignore here for the sake of brevity)

1 *Basic type*, written □ Nodes of this type contain the data stored in the database

2 *Composition*, written ⌓ Nodes of this type contain tuples whose components are taken from the successors of the node

3 *Collection*, written ○ Nodes of this type contain sets, all of whose elements are taken from the node's successor

An *instance* of a schema is an assignment of a set of values to each node We use the notation $I(u)$ for the set of values assigned to the node u by the instance I $I(u)$ consists of a set of *l-values*, with corresponding *r-values* Intuitively, r-values constitute the data space, and l-values constitute the address space The instance of a node consists of set of l-values, with an r-value assigned to each of them Formally, the *l-values* are elements of a fixed set L (usually taken to be the set of natural numbers) We require that the instances of distinct nodes be disjoint We also have a function r on L, that assigns r-values to

these l-values, and we require that the r-values be of the correct form, depending on the type of the node, as follows

If $l \in I(v)$ then

1 if v is of type $\square$, then $r(l)$ must be in the data domain D,

2 if v is of type $\frown$ with children $v_1, \ldots, v_n$, then $r(l)$ must be a tuple $(l_1, \ldots, l_n)$ where for each I, $l_i \in I(v_i)$, and

3 if v is of type $\bigcirc$, and w is its child, then $r(l)$ must be a subset of $I(w)$

2.2 Logic

Let S be a schema Each variable over S has a fixed *sort*, where the sorts are nodes of S The sorts restrict the possible values the variable may take For example, if x is a variable of sort v, then x can take only values in $I(v)$ We shall usually subscript the variable with its sort, e g , x_v We also have countably many constants, ranging over the domain D

Definition 1 An *atomic formula* over S is one of the following

1 $x_v \; \pi_t \; y_w$, meaning that the l-value of x_v is the t^{th} component of the r-value of y_w

2 $x_v \in y_w$, meaning that the l-value of x_v is a member of the r-value of y_w

3 $x_v =_l y_w$, meaning that the l-values of x_v and y_w are equal

4 $x_v =_r y_w$, meaning that the r-values of x_v and y_w are equal

5 $x_v =_r d$, where d is a constant, meaning that the r-value of x_v is d

In each of these atomic formulas, v and w are required to be of types for which the formula is meaningful For example, $x_v \in y_w$ can be used only when w is of type $\bigcirc$ and v is its child We then define well-formed formulas over S in the usual way $\models_I \phi(l_1, \ldots, l_n)$ means that ϕ is satisfied by $l_1, \ldots, l_n$ in the instance I, and is defined in the usual way

3 The Complexity of Integrity Checking

In this section we investigate the complexity of checking integrity That is, we have integrity constraints that are sentences in LDM-logic, and a database is "legal" if and only if it satisfies the constraints Following [Va82], we use two measures of complexity, *data complexity* and *expression complexity* Intuitively, data complexity is the complexity of testing satisfaction of a fixed sentence, in terms of the size of the database Expression complexity, on the other hand, is the complexity of testing satisfaction of sentences on a fixed database, in terms of the length of the sentences (To measure complexity we clearly have to assume that all instances are finite)

More formally, the data complexity of LDM-logic is the complexity of the sets

$$Gr(S, \phi) = \{I \mid I \text{ is an instance of } S \text{ and } \models_I \phi\},$$

where S is a schema and ϕ is a sentence over S The expression complexity of LDM-logic is the complexity of the sets

$$Gr'(S, I) = \{\phi \mid \models_I \phi\},$$

where S is a schema and I is an instance of S Note that $Gr(S, \phi)$ is a set of instances, while $Gr'(S, I)$ is a set of sentences

Theorem 1:

1 For every schema S and every sentence ϕ over S, the set $Gr(S, \phi)$ is in LOGSPACE

2 For every schema S and every instance I of S, the set $Gr'(S, I)$ is in PSPACE

3 There is a schema S and an instance I of S such that the set $Gr'(S, I)$ is logspace complete in PSPACE ∎

Thus the data complexity of LDM-logic is LOGSPACE and the expression complexity of LDM-logic is PSPACE Since analogous results hold for the relational model, we see that integrity checking in the logical data model is not more difficult than in the relational model.

4 LDM-Logic is First-Order

In this section we show that LDM-logic is essentially first-order, that is, it has a proof theory, it is compact and it satisfies a Lowenheim-Skolem theorem. We prove this by reducing LDM-logic to a many-sorted first-order logic with equality. We mention in contrast that Jacobs' database logic [Ja82] is inherently a higher-order logic that does not have any of the above properties.

Let $\mathcal{L}$ be an LDM-logic over a schema S. We shall construct a corresponding many-sorted first-order logic with equality $\mathcal{L}'$ as follows. $\mathcal{L}'$ will have a sort v for each node v of the schema, and one special sort d, which corresponds to the domain D from which the data are taken. $\mathcal{L}'$ has variables ranging over all the sorts, except for the special sort d.

$\mathcal{L}'$ has the following relation and function symbols. For each node w in S of type $\bigcirc$ with child v, $\mathcal{L}'$ has a binary relation symbol $\in_w$ between elements of sorts v and w. For each node w of type $\bigcirc\!\!\!\!-$ with v as its i^{th} child, $\mathcal{L}'$ has a function symbol $\pi_{w,i}$ from sort w to sort v. Intuitively, $\pi_{w,i}$ maps l-values to the i^{th} components of their r-values. For each node v of type $\square$, $\mathcal{L}'$ has a function symbol f_v from sort v to sort d. Finally, the constants of $\mathcal{L}$ are also constants of $\mathcal{L}'$ of sort d.

Given an $\mathcal{L}$-formula ϕ we convert it to an $\mathcal{L}'$-formula $F(\phi)$, as follows:

1. $F(x_v \pi_i y_w)$ is $x_v = \pi_{w,i}(y_w)$
2. $F(x_v \in y_w)$ is $x_v \in_w y_w$ (we use infix notation)
3. $F(x_v =_l y_v)$ is $x_v = y_v$
4. $F(x_v =_r y_w)$ depends on the type of v and w. For example, if v and w are of type $\square$, then $F(x_v =_r y_w)$ is $f_v(x_v) = f_w(y_w)$
5. $F(x_v =_r c)$ is $f_v(x_v) = c$

The mapping of nonatomic $\mathcal{L}$-formulas to $\mathcal{L}'$-formulas is straightforward. Given an instance I of S, we can construct a structure $F(I)$ of $\mathcal{L}'$ as follows. The domain corresponding to each sort v, except for d, will be the set $I(v)$. The domain corresponding to the sort d will be the set D. The interpretation of $\pi_{w,i}$ will map each $l \in I(w)$ to the i^{th} component of its r-value, and the interpretation of f_v will map each $l \in I(v)$ to its r-value. Finally, the interpretation of $\in_w$ will be a subset of $I(v) \times I(w)$, where v is the child of w, such that the pair (l_1, l_2) is in this relation iff $l_1 \in r(l_2)$.

Theorem 2: For any $\mathcal{L}$-sentence ϕ, we have $\models_I \phi \Leftrightarrow \models_{F(I)} F(\phi)$ ∎

We now want to define a mapping from the first-order logic $\mathcal{L}'$ to the LDM-logic $\mathcal{L}$. Each atomic formula in the first-order logic $\mathcal{L}'$ must be either of the form $t_1 = t_2$ or $t_1 \in_w t_2$. We shall only show here how to deal with the latter case, the former is dealt with in a similar way.

In the latter case, t_2 must be of sort w, and t_1's sort must be w', the child of w. Since each f_v maps a term to one of sort d, and there are no function symbols from sort d, the only possible form the terms t_1 and t_2 can have is

$$t_1 = \pi_{u_1,i_1} \quad \pi_{u_{n-1},i_{n-1}} \pi_{u,i_n}(x_u),$$

and

$$t_2 = \pi_{v_1,j_1} \quad \pi_{v_{m-1},j_{m-1}} \pi_{v,j_m}(y_v)$$

We introduce new variables $z^1_{w'}, z^1_{u_1}, \ldots, z^1_{u_{n-1}}$ and $z^2_w, z^2_{v_1}, \ldots, z^2_{v_{m-1}}$, and convert $t_1 \in_w t_2$ into the $\mathcal{L}$-formula

$$(\exists z^1_{w'})\ldots(\exists z^2_{v_{m-1}})(z^1_{w'} \pi_{i_1} z^1_{u_1} \wedge \ldots \wedge z^1_{u_{n-1}} \pi_{i_n} x_u \wedge z^2_w \pi_{j_1} z^2_{v_1} \wedge \ldots \wedge z^2_{v_{m-1}} \pi_{j_m} y_v \wedge z^1_{w'} \in z^2_w)$$

We then convert nonatomic $\mathcal{L}'$-formulas into $\mathcal{L}$-formulas in the obvious way. If ϕ is a formula in $\mathcal{L}'$, we shall use the notation $L(\phi)$ for the resulting $\mathcal{L}$-formula. We then convert any structure I of $\mathcal{L}'$ into an instance $L(I)$ of S, such that the following theorem holds.

Theorem 3: For any $\mathcal{L}'$-sentence ϕ, we have $\models_I \phi \Leftrightarrow \models_{L(I)} L(\phi)$ ∎

We note that if ϕ is an $\mathcal{L}$-sentence, then $L(F(\phi))$ is logically equivalent in $\mathcal{L}$ to ϕ. Similarly, if ϕ is an $\mathcal{L}'$-sentence, then $F(L(\phi))$ is logically equivalent to ϕ.

From the above two theorems, the following results about LDM-logic follow.

Corollary 4: (Validity) Let ϕ be a LDM-sentence Then ϕ is valid if $F(\phi)$ is valid, and vice-versa ∎

Corollary 5· (Compactness) Let Σ be a set of LDM-sentences over a schema S Then Σ is satisfiable iff every finite subset of Σ is satisfiable ∎

Corollary 6. (Lowenheim-Skolem) Let Σ be a set of LDM-sentences over a schema S If S is satisfiable, then it is satisfiable by a countable model ∎

While the latter two corollaries are of theoretical interest, the Validity Corollary has also a practical significance It implies that with the appropriate interface we can use a standard theorem-prover either in the database design process or for deductive query processing (see [BBG78] [MMSU81] [NG78] [Re84])

5 Elimination of Cycles

In the logical data model cycles can exists not only in the schemas but also in the data, e g , if $r(l) \in l$ for some l-value l, then we have a cycle in the data This feature of the model has its price For example, as explained in [Ku85], it is not clear how to construct the cycles in the first place Indeed, the query languages defined in [KV84] cannot construct new cycles Furthermore, in updating a cyclic database one cannot deal with one node at a time, rather one has to update all the nodes on the cycle simultaneously. Thus, in view of this price, we would like to study the expressive power of cycles Namely, are there applications that cannot be modelled without cycles? Consider the following example

Example 1. Fig 1 shows an example of a cyclic schema of a a database that stores the information about procedure calls in a program Elements in $I(v)$ represent procedures, elements in $I(u)$ represent procedure names, and elements in $I(w)$ represent sets of procedures Thus, if $x \in I(v)$ and $r(x) = (y, z)$, then $r(y)$ is the name of the procedure x, and $r(z)$ is the set of procedures called from x Note that if a procedure calls itself, then we have a cycle in the

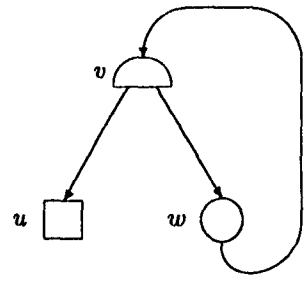

Figure 1 Cyclic Schema

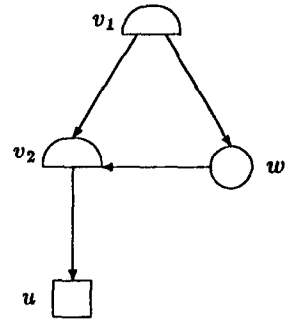

Figure 2 Equivalent Acyclic Schema

data An acyclic schema that intuitively seems to "capture the same information" is shown in Fig 2 Elements in $I(v_2)$ represent procedure entities, elements in $I(u)$ represent procedure names, elements in $I(w)$ represent sets of procedures, and elements in $I(v_1)$ represent the relationship "procedure calls procedures" ∎

To formalize the idea of "capturing the same information," we use the following definition, which is closely related to the notion of query-equivalence of [Hu84]

Definition 2: Let S and T be schemas Then T *dominates* S if there is a mapping f of instances of S to instances of T, such that for each query Q_1 on S there is a query Q_2 on T such that $Q_1(I)$ is isomorphic to $Q_2(f(I))$ for all instances I of S We say that S and T are

equivalent if each of them dominates the other

The definition of equivalence depends, of course, upon one's choice of a query language In [KV84], we defined two query languages for the logical data model, one procedural and the other nonprocedural, and proved their equivalence We recall the definition of the nonprocedural language A *query* on a schema S consists of

1 An acyclic schema S'

2 A collection of formulae, one (ϕ_v) for each node v of S' The formula ϕ_v must satisfy

   (a) There is only one free variable in ϕ_v, and it is of sort v

   (b) All other variables are bound, and their sorts are either nodes of S, or are descendants of v (Intuitively, ϕ_v describes what data should be at node v, in terms of the instances of nodes that have already been constructed)

In [KV84] we showed how to construct the result of the query "bottom-up," and proved that the result is unique up to isomorphism

It can be shown that the two schemas in Example 1 are equivalent, provided that the relationship represented by v_1 is functional, i e , for every procedure there is a unique set of procedures that it calls We now describe a general transformation from cyclic schemas to equivalent acyclic schemas The general idea is to break cycles by creating composition nodes that represent the cyclic relationships

When we try to break cycles in arbitrary cyclic schemas, we notice that in several pathological cases it will not work First, the cycle has to contain a ⌓-node at which to break it Even if the cycle has a ⌓-node, this node may have only one child, which would leave us with a childless ⌓-node after breaking the cycle What all these examples have in common is that in each case the schema relates l-values to l-values, and never relates them to the actual data Intuitively, pure relationships between l-values do not correspond to anything in the "real world" This motivates the following definition.

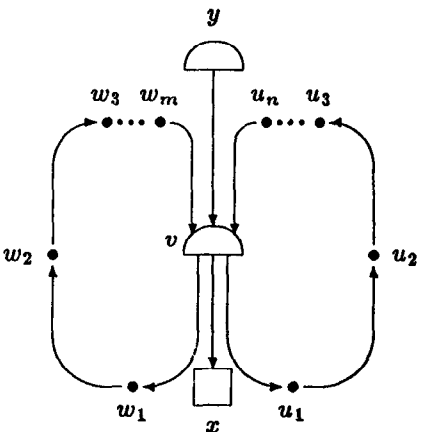

Figure 3 Cycles Through v.

Definition 3: A schema S is called *well-formed* if from each node in the schema, there is a path to a node of type □

Our transformation will apply only to well-formed schemas Let S be a well-formed cyclic schema It is easy to see that S must have a node v of type ⌓ that is in at least one cycle, and has at least one child that is not in any cycle (for example, see Fig 3) We break all the cycles that go through the node v as follows (see Fig 4) We replace the ⌓-node v by two nodes v_1 and v_2 All incoming edges to v in S except those in the cycles now enter v_1 All outgoing edges from v except those in the cycles now leave from v_2

Lemma 7: Let S be a well-formed cyclic schema If we break a cycle in S as described above, the resulting schema is also well-formed ∎

This lemma shows that we can repeatedly break cycles in the schema, at each step getting a new well-formed schema

Lemma 8: If S is well-formed, then repeatedly breaking the cycles, no matter in which order nodes at which to break the cycles are selected, eventually yields an acyclic schema T. ∎

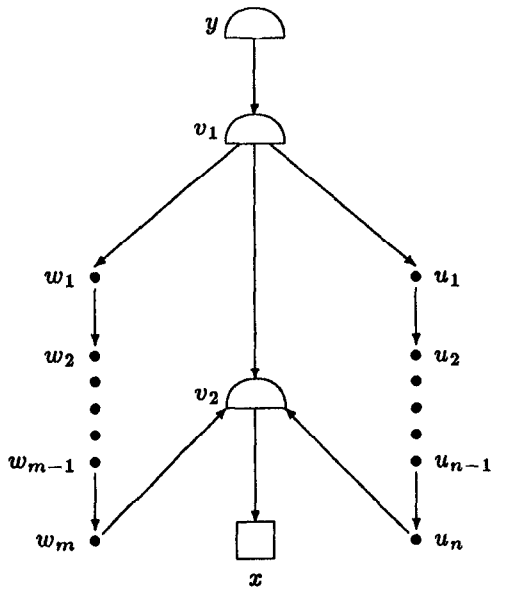

Figure 4 After Breaking the Cycles

We now have to show how to convert an instance I of S into an instance $f(I)$ of T Assume that we got from S to T by breaking one cycle C as above (the general construction if there is more than one cycle will follow easily from this). L-values in instances of nodes different from v, that neither have v as parent or as child are unchanged Each l-value in $I(v)$ is replaced by a pair of l-values, one in $f(I)(v_1)$ and the other in $f(I)(v_2)$, the second being the child of the first We then modify the l-values in the parents and children of v, in a fairly straightforward way

Lemma 9: Let S, T and f be as above Then, for each query Q_1 on S, there is a query Q_2 on T such that $Q_1(I)$ is isomorphic to $Q_2(f(I))$ for all instances I of S Thus, T dominates S ∎

To show that S and T are equivalent, we have to show that S dominates T This is not, however, true in general The problem is that the relationships that represent the cycles have to be functional That is, each l-value in $f(I)(v_2)$ is the child of exactly one l-value in $f(I)(v_1)$ This motivate the following definition

Definition 4: A *constrained* schema is a pair (S, Φ), where S is a schema and Φ is a sentence over S An instance I of S is an instance of (S, Φ) if $\models_I \Phi$.

It turns out that it is quite easy to capture the functionality constraint by constrained schemas Thus rather then transform S to T, we transform it into (T, Φ) for an appropriate Φ

Theorem 10: For any well-formed schema S, there exists an equivalent acyclic constrained schema (T, Φ) ∎

The reader may ask why we have bothered to introduce cycles into the logical data model if they do not add any expressive power The answer to that is two-fold First, without introducing cycles we could not prove that they do not add any expressive power Secondly, we have only shown that, according to a *certain* measure, cycles do not add any expressive power But it is not clear that this measure is the ultimate one We believe that the issue of cycles deserves further study

6 Acknowledgments

We are grateful to R Hull for stimulating discussions In particular, he raised the question whether cycles add expressive power to our model We'd like to thank R Fagin and J D. Ullman for their helpful comments on a previous draft of this paper We also appreciate comments by one of the reviewers

References

[BBG78] C Beeri, P A Bernstein, N Goodman "A Sophisticate's Introduction to Database Normalization Theory," *Proc 4th Intl Conf on Very Large Data Bases*, Berlin, 1978, pp 113–124

[Co70] E F Codd "A Relational Model of Data for Large Shared Data Banks," *Comm ACM*, **13** 6, 1970, pp 377–387

[Co79] E F Codd "Extending the Database Relational Model to Capture more Meaning," *ACM Trans on*

Database Systems, 4 4, 1979, pp 397–434

[HM78] M Hammer, D McLeod "Database Description with SDM—A Semantic Database Model," *ACM Trans on Database Systems*, 3 3, Sept 1978, pp 201–222

[Hu82] R Hull, C K Yap "The Format Model A Theory of Database Organization," *Proc 1st ACM Symp on Principles of Database Systems*, 1982, pp 205–211

[Hu84] R Hull "Relative Information Capacity of Simple Relational Database Schemata, " *Proc 3rd ACM Symp on Principles of Database Systems*, 1984, pp 97 -109

[Ja82] B E Jacobs "On Database Logic," *J ACM*, **29** 2, 1982, pp 310–332

[Ku85] G M Kuper *"The Logical Data Model,"* Ph D Dissertation in preparation, Dept of Computer Science, Stanford University, 1985

[KV84] G M Kuper, M Y Vardi "A New Approach to Database Logic," *Proc. 3rd ACM Symp on Principles of Database Systems*, 1984, pp 86 96.

[MMSU81] D Maier, A O Mendelzon, F. Sadri, J D Ullman "Adequacy of Decompositions of Relational Databases," in *Advances in Database Theory* (H. Gallaire, J Minker, and J M Nicolas, eds), Plenum Press, 1981, pp. 101- 114

[NG78] J M Nicolas, H Gallaire "Database-Theory vs Interpretation," in *Logic and Databases* (H Gallaire and J. Minker, eds), Plenum Press, 1978, pp 33–54.

[Re84] Reiter, R , "Towards a Logical Reconstruction of Relational Database Theory," in *On Conceptual Modelling Perspectives from Artificial Intelligence, Databases, and Programming Languages* (M L Brodie, J Mylopoulos, and J Schmidt, eds), Springer-Verlag, 1984, pp 191–233

[ScSw75] H A Schmidt, J R Swenson "On the Semantics of the Relational Data Model," *Proc ACM Int'l Conf on Management of Data*, San Jose, 1975, pp 211–233

[Shi78] D Shipman "The Functional Data Model and the Data Language DAPLEX," *ACM Trans on Database Systems*, **6** 1, 1981, pp 140–173

[SmSm77] J M Smith, D C P Smith "Database Abstractions Aggregation," *Comm. ACM*, **20** 6, 1977, pp 405–413.

[Ul82] J D Ullman *Principles of Database Systems*, Computer Science Press, 1982

[Va82] M Y Vardi "The Complexity of Relational Query Languages," *Proc. 14th ACM Symp on the Theory of Computing*, 1982, pp 137–146

[Va83] M Y Vardi "Review of [Ja82]," *Zentralblatt fur Mathematik*, 497.68061, 1983