IDLOG: Extending the Expressive Power of Deductive Database Languages

Yeh-Heng Sheng
Department of Computer Science
The State University of New York at Stony Brook
Stony Brook, New York 11794
(ysheng@sbcs sunysb edu)

Abstract

The expressive power of pure deductive database languages, such as *DATALOG* and *stratified DATALOG*$^{\neg}$, is limited in a sense that some useful queries such as functions involving *aggregation* are not definable in these languages Our concern in this paper is to provide a uniform logic framework for deductive databases with greater expressive power It has been shown that with a linear ordering on the domain of the database, the expressive power of some database languages can be enhanced so that some functions involving aggregation can be defined Yet, a direct implementation of the linear ordering in deductive database languages may seem unintuitive, and may not be very efficient to use in practice We propose a logic for deductive databases which employs the notion of "identifying each tuple in a relation" Through the use of these *tuple-identifications*, different linear orderings are defined as a result This intuitively explains the reason why our logic has greater expressive power The proposed logic language is *non-deterministic* in nature However, non-determinism is not the real reason for the enhanced expressive power A deterministic subset of the programs in this language is *computational complete* in the sense that it defines all the *computable deterministic queries* Although the problem of deciding whether a program is in this subset is in general undecidable, we do provide a rather general sufficient test for identifying such programs Also discussed in this paper is an extended notion of queries which allows both the input and the output of a query to contain *interpreted constants* of an infinite domain We show that extended queries involving aggregation can also be defined in the language

1 Introduction

The expressive power of pure deductive database languages, such as DATALOG and stratified DATALOG$^{\neg}$, is limited in a sense that some useful queries such as functions involving aggregation are not definable in these languages It has been shown that the class of DATALOG queries is equivalent to the class YE^{+} of queries definable by a *fixpoint operator* applied to a *positive existential query* [CH82], and the class of stratified DATALOG$^{\neg}$ queries is a strict subset of the class FP of *fixpoint queries* [Dah87, KP88] The class FP is a subset of the class RQ of *while queries* [CH80] However, there are many reasonable queries which are not in RQ As a typical example, consider the query EVEN which returns TRUE if the size of the input database domain is even, and returns FALSE otherwise (the input database contains no relations) The query EVEN is not in RQ [CH80] Other query classes defined by procedural database languages have also been considered One example is the class of *bounded loop queries* which although subsumes EVEN, but the bounded loop construct still lacks the ability to define the query EQUAL which checks the equality of the sizes of two relations [Cha88] Chandra [Cha81] has shown that even by adding *counters* (natural number variables) to RQL, i e the language defining RQ, RQ still does not subsume the query EVEN Although RQL can be expanded by adding *generic variables* [Cha81], the notion of *wide relations* employed in the extended language makes the query language potentially inefficient, and often unintuitive[1] for use in practice [Cha88] On the other hand, a quick remedy for the expressive power of FP is to add a *linear ordering* on the domain of the database With the linear ordering, FP is identical to the class of *polynomial-*

[1] Abiteboul and Vianu [AV87, AV88] have shown that the requirement for wide relations can be dispensed with by adding *invented values* which are new elements that do not appear in the database or in the computation until the time they are created The languages considered in this paper do not involve this concept

time queries [Saz80, Var82, Imm86] and RQ is the same as the class of *polynomial-space queries* [Var82] Therefore, both of them subsume the query EVEN (EVEN is a polynomial-time query) Yet, as argued in [Cha88], the linear ordering may seem unintuitive for the programmer, and may not be very efficient to use in practice Accordingly, a direct implementation of the linear ordering in deductive databases may not be favorable

As for complex-object languages, COL with *untyped sets* under both *stratified semantics* and *inflationary semantics* has been shown to be computational complete in [HS89] There are also some algebra-based and calculus-based complex-object languages which are computational complete as shown in [HS89] In this paper, however, we consider languages with only atomic objects We propose a logic, IDLOG (LOGic with tuple-IDentifications), for deductive databases which employs the notion of "identifying each tuple in a relation" Our logic is *nondeterministic*[2] in a sense that we may get different answers for the same query according to different assignments of tuple-identifications (tid's for short) in each relation However, non-determinism is not the real reason for the enhanced expressive power The class of *computable deterministic queries* is subsumed by the class of *deterministic queries* defined by a subset of IDLOG programs Although the problem of deciding whether a program is in this subset is in general undecidable, we do present a rather general sufficient test for identifying such programs The resulting class of queries defined by programs recognized by this test still subsumes the queries EVEN and EQUAL Furthermore, if we allow both the input and the output of a query to contain *interpreted constants*, such as natural numbers, we can define more general queries involving aggregation Aggregate functions are useful in database systems, especially in statistical databases In [Klu82], relational algebra and relational calculus are extended to include aggregate functions, which are then extended in [OOM87] to relations with set-valued attributes Aggregate functions in deductive databases have also been considered in languages with complex objects in [AB88] and [Che88] In these languages, aggregate functions are used as built-in functions In this paper, our concern, however, is to provide a uniform logic framework for deductive databases with greater expressive power, so that, as a consequence, general aggregate functions can be defined IDLOG queries can be evaluated by a slight modification of some existing strategies for evaluating logic program queries For deterministic IDLOG queries, the evaluation can be in general more efficient

The rest of this paper is organized as follows In Section 2 IDLOG is presented, the expressive power of IDLOG is explored in Section 3, Section 4 shows how IDLOG defines general queries involving aggregation in IDLOG, the evaluation of IDLOG queries is briefly discussed in Section 5, Section 6 concludes the paper

2 IDLOG

In this section, we first define the notion of *ID-relations*, and the syntax and semantics of IDLOG Then the intended semantics of IDLOG programs is given which is a natural extension of the perfect-model semantics of logic programs [Prz88b, Prz88a]

2.1 ID-relations

In this paper, we only consider *flat relations*, i e relations containing no complex objects such as sets and lists Function symbols are excluded as well[3] Consider a countably infinite set U, called the *universal domain* An *uninterpreted domain* (u-domain for short) is a finite subset of U, and the *interpreted domain* is the set N of natural numbers 0, 1, 2, Suppose $m, n \geq 0$ The *types* defined in the following are 0, 1 sequences An *extended relation* (*relation* for short) of *type* $s_1 \quad s_m$ with u-domain D is a (possibly infinite) subset of $D_1 \times \quad \times D_m$ such that for $i = 1, m$, $D_i = D$ if $s_i = 0$, and $D_i = \mathsf{N}$ if $s_i = 1$ Suppose for $j = 1, n$, r_j is a finite relation of type a_j with u-domain D An *extended database* (*database* for short) of type $(a_1, \quad, a_n)$ with u-domain D is of the form $(D, r_1, \quad, r_n)$

Now we define the notion of *ID-relations* Intuitively, an ID-relation of a relation r is obtained from r by appending each tuple in r with its tid which is determined according to the way tuples in r are grouped Let r be an n-ary relation of type a, and r_c denote a selection of tuples in r which satisfy the condition c Let also v be a tuple of distinct attribute numbers in $\{1, \quad, n\}$ An ID-relation of r on v is an $(n+1)$-ary relation of type $a \quad 1$ with the same u-domain as r, and each of its tuples is composed of a tuple in r and a tid which is assigned w r t the corresponding sub-relation grouped by attributes in v Suppose $f_r^c \quad r_c \rightarrow \{0, 1, \quad, |r_c| - 1\}$ is a one-to-one and onto function for each relation r and each condition c Then an ID-relation of r on v is an $(n+1)$-ary

[2] The non-deterministic properties of our logic is discussed more thoroughly in [She90]

[3] Although we are primarily interested in programs with no general function symbols, the IDLOG programs we define here can be naturally extended to allow them

relation defined by

$$\{t'd \quad t' \in r, t'[v] = t, f_r^{v=t}(t') = d\}$$

Note that when v is an empty tuple, the above reduces to

$$\{t'd \quad t' \in r, f_r^{\text{true}}(t') = d\},$$

i e , each tuple of r is assigned a unique tid

Example 1 Let $r = \{(a, c), (a, d), (b, c)\}$, then the ID-relations of r on the attribute tuple (1) are
$\{(a, c, 1), (a, d, 0), (b, c, 0)\}$, and
$\{(a, c, 0), (a, d, 1), (b, c, 0)\}$ ∎

2 2 Languages and Interpretations

Consider a two-sorted first-order logic language with sorts u and i Constants of sort u are elements of U and constants of sort i are 0, 1, Each *interpretation I* of this language contains two *universes*, namely u-universe and i-universe, where the latter is intended to be the set N Equality symbols (for both sorts) are assumed in this language We also assume the existence of standard *arithmetic predicates*, such as +, −, *, / (of sort (i, i, i), and < (of sort (i, i)) Sorts of predicates are also written as 0, 1 sequences denoting attributes of sort u and sort i respectively For convenience, we will not mention the sorts of variables and predicates if they can be inferred from the context An *IDLOG language* L is a such language with additional ID-predicates Let v be a (possibly empty) tuple of distinct attributes in $\{1, \quad, n\}$ For each ordinary n-ary predicate symbol p of sort a, an *ID-version* of p on v is a predicate symbol of sort a 1, denoted by $p[v]$ The set of all *ID-predicates* in an IDLOG language L is the set of all ID-versions of ordinary predicates in L Hereinafter, unless stated otherwise, we will ignore arithmetic predicates for they have respective intended meanings

IDLOG interpretations are extensions of first-order logic interpretations An IDLOG interpretation I, in addition to the usual assignment of a relation to each ordinary predicate, assigns to each ID-predicate $p[v]$ an ID-relation of p^I on v, where p^I is the relation assigned by I to the predicate p An atom (resp a literal) is called an *ID-atom* (resp an *ID-literal*) if it contains an ID-predicate, and a *non-ID-atom* (resp a *non-ID-literal*) otherwise The *predicate symbol of a non-ID-literal* A is the ordinary predicate in A, while the *predicate symbol of an ID-literal* B is the ID-predicate in B An *IDLOG clause* is a sentence of the (Skolem) form $(\forall^{s_1} X_1 \quad \forall^{s_k} X_k)(A \leftarrow B_1 \wedge \quad \wedge B_n)$, where B_j's are any literals, and A is a non-ID-atom containing no equality or arithmetic predicates And an *IDLOG program* is a finite set of IDLOG clauses Throughout this paper, we assume each use of an arithmetic predicates is *safe* [Zan86] which is guaranteed through a sufficient condition for each literal l containing an arithmetic predicate in the body of a clause r, a "sufficient" number of arguments of l must be *bound* in the body of the same clause, i e , each of them either is a constant or appears in a positive literal containing no arithmetic predicates in the body of r For example, in the following program,

q(a, 1)
$p1$(X, N) ← q(X, N), +(N, L, M)
$p2$(X, N) ← q(X, N), +(L, M, N)

the first occurrence of + is not allowed, since there are infinitely many solutions for the equation 1 + L = M, the second occurrence of + is allowed since there are only finite number of solutions for the equation L + M = 1 Thus, for the arithmetic predicate +, allowed combinations of bound (b) and unbound (n) arguments are "bbb", "bbn", "bnb", "nbb", and "nnb" (note that this is only a sufficient condition for ensuring safety) Similarly, we can define the sufficient numbers for all other arithmetic predicates

A *universal interpretation* $\mathcal{U}$ of an IDLOG program P is an IDLOG interpretation of P which assigns to each ordinary predicate in P of sort $s_1 \quad s_n$ the set $U_1 \times \quad \times U_n$ such that for each i, U_i = the u-universe of $\mathcal{U}$ if $s_i = 0$, and $U_i = \mathbb{N}$ otherwise Note that there is no other restriction on the assignment for ID-predicates in a universal interpretation Every universal interpretation of an IDLOG program P is also a model of P

Consider an IDLOG program P The Herbrand universe of P, is the union of N and the set of all constants of sort u in P (if there is no constant of sort u in P then a new constant will be added) Henceforth, when mentioning formulae, we will assume they are of the right sorts Let H_P be the set of all predicates in P and their corresponding ID-versions or ordinary predicates The *Herbrand base* of P consists of ground atoms constructed from all predicates in H_P, and constants in the Herbrand universe of P A *Herbrand interpretation* of P is a subset of the Herbrand base of P which satisfies the requirement of being an IDLOG interpretation, i e relations assigned to ordinary predicates and their ID-versions stand in right relationship

2.3 Perfect-Model Semantics of IDLOG Programs

Recently, finding an appropriate declarative semantics for *logic programs* (DATALOG¬ with function symbols) has attracted much attention [ABW88,

GL88, Prz88b, Prz88a, VG88, VGRS88] In the following, we will give an intended semantics for IDLOG programs which is a natural extension of the *perfect-model semantics* for logic programs [Prz88b, Prz88a]

We first discuss the inappropriateness of minimal-model semantics for *definite programs* in IDLOG, i e , positive IDLOG programs containing no ID-literals Consider the definite program P given by

p(a)
q(b)

The following are both minimal models of P

$M_1 = \{p(a), p[](a, 0), p[1](a, 0), q(b), q[](b, 0), q[1](b, 0)\}$
$M_2 = \{p(a), p(b), p[](b, 0), p[](a, 1), p[1](a, 0), p[1](b, 0), q(b), q[](b, 0), q[1](b, 0)\}$

Apparently, M_2 contains p(b) which is not what we really want The problem comes from the requirement that tid's are continuous and begin with zero, which can force some unnecessary non-ID-atoms to be in the minimal models Thus, the minimal-model semantics is not appropriate for even definite programs in IDLOG However, as shown below, if we regard each ordinary predicate as having a "higher priority" than all of its ID-versions, perfect-model semantics can be suitably defined for definite programs In fact, as shown in Theorem 1, the perfect-model semantics for definite programs in IDLOG is a natural extension of the one for definite programs in standard (first-order) logic

Before defining perfect models of IDLOG programs, we first extend the priority relations $<$ and $\leq$ for logic programs to IDLOG programs as follows Let P be an IDLOG program Suppose p, q, s, and t are predicates in H_P The priority relation $<$ and the auxiliary relation $\leq$ for P are defined by the following rules

- p $<$ q, if p is an ID-version of q,
- p $<$ q, if p is in the head and q is the predicate of a negative literal in the body of a clause in P,
- p $\leq$ q, if p $=$ q, or if p is in the head and q is the predicate of a positive literal in the body of a clause in P

We also assume that $\leq$ is *transitive* and $<$ is related to $\leq$ in the expected way

- if p $\leq$ q and q $\leq$ s, then p $\leq$ s,
- if p $\leq$ q and q $<$ s (resp t $<$ p) then p $<$ s (resp t $<$ q),
- if p $<$ q then p $\leq$ q

Notice that the priority relation $<$ defined above is transitive as well

Suppose that M and N are two distinct models of an IDLOG program We say that N is *preferable* to M (written $N \ll M$), if for every predicate p for which the set $\mathsf{p}^N - \mathsf{p}^M$ is not empty, there is a predicate q in P such that p $<$ q and the set $\mathsf{q}^M - \mathsf{q}^N$ is not empty A model M of P is *perfect* if there are no models preferable to it We write $M \preceq N$, if $M = N$ or $M \ll N$ It is observed that for each IDLOG program P, a perfect model of P is also a minimal model of P, but even for definite programs, a minimal model is not necessarily a perfect model As an example, the model M_1 is preferable to M_2, and, therefore, M_2 is not a perfect model However, the set of all perfect models of a definite program P (in IDLOG) can be easily characterized by the unique minimal model of P in standard logic

Theorem 1 *Suppose P is a definite program An IDLOG interpretation M is a perfect model of P iff the set of all non-ID-atoms in M is the unique minimal model of P in standard logic* ∎

Yet, as we know, a logic program may not have a perfect model [Prz88b], to avoid the same situation, we naturally extend the definition of *stratification* for logic programs to IDLOG programs

Definition 1 *An IDLOG program P is* stratified *if and only if we can decompose the set H_P into disjoint sets H_0, H_1, , H_n, called* strata, *so that*

1 *for each ID-version $p[v]$ of an ordinary predicate p in P $p[v]$ belongs to $\cup \{H_j \mid p \in H_i \ \& \ j > i\}$, and*
2 *for every clause $C \leftarrow A_1, \ldots, A_a, \neg B_1, \ldots, \neg B_b$ in P, where A_i's and B_j's are atoms, we have*
   - *for all i, $\mathsf{stratum}(A_i) \leq \mathsf{stratum}(C)$, and*
   - *for all j, $\mathsf{stratum}(B_j) < \mathsf{stratum}(C)$,*

where $\mathsf{stratum}(X) = k$ if the predicate symbol of X belongs to H_k Any particular decomposition $\{H_0, H_1, \ldots, H_n\}$ of H_P satisfying the above conditions is called a stratification *of P* ∎

Proposition 1 *For each stratified IDLOG program P, there exists a stratification H_0, H_1, , H_{2n+1} such that each stratum in $\{H_0, H_2, \ldots, H_{2n}\}$ contains only ordinary predicates, and for $i = 0, \ldots, n$, each stratum H_{2i+1} contains all the ID-versions of each ordinary predicate in H_{2i}*

Now, we have the following main result about perfect models of IDLOG programs

Theorem 2 *For every model N of a stratified IDLOG program P, there exists a perfect model M of P such that $M \preceq N$. Thus, every stratified IDLOG program has at least one perfect model.*
∎

The corresponding result in [Prz88b] for stratified logic programs can be thought of as a special case of the above theorem. The perfect-model semantics for IDLOG programs can therefore be similarly defined. To overcome the so called *universal-query* problem [Prz88a], the above result can be extended to general interpretations along the lines of [Prz88a].

3 The Expressive Power of IDLOG

In this section, we first define IDLOG queries. Then we discuss the expressive power of IDLOG programs. A sub-class of stratified IDLOG programs defines the class of all computable deterministic queries.

3.1 IDLOG Queries

The following definition of queries is an extension of the one defined in [CH80, Cha81]. Some terminologies are borrowed from [HS89] and [AV88].

An *elementary* relation type is a relation type containing no 1's. Suppose for $i \geq 0$, a_i is an elementary relation type. A *(non-deterministic) query* $\mathbf{f}$ of type $\bar{a} = (a_1, \ldots, a_n) \rightarrow a_0$ is a partial function giving, for each database of type $\bar{a}$, an output (if any) which is a non-empty set of finite relations of type a_0 such that the u-domain of each relation in the output is the same as the u-domain of the input database. When an input database $\mathbf{r}$ contains only one relation r and the u-domain of $\mathbf{i}$ is immaterial to the context, we will write r instead of $\mathbf{r}$. Each relation in the output of a query $\mathbf{f}$ with input $\mathbf{i}$ is called a *maybe-answer* of $\mathbf{f}(\mathbf{r})$. A query is said to be *deterministic* if for each appropriate input, the output (if any) contains only one maybe-answer. Let C be a finite subset of U. A query $\mathbf{f}$ of type $\bar{a} \rightarrow a_0$ is *C-generic* if for each database $\mathbf{r}$ of type $\bar{a}$ and for each relation r of type a_0, $r \in \mathbf{f}(\mathbf{i})$ iff $\sigma(r) \in \mathbf{f}(\sigma(\mathbf{r}))$ for each permutation σ over U with $\forall x \in C, \sigma(x) = x$ (σ is naturally extended to relations and databases). A query $\mathbf{f}$ is *computable* iff it is C-generic (for some C) and the input-output relation $\{(\mathbf{r}, r) \mid r \in \mathbf{f}(\mathbf{r})\}$ is a recursively enumerable set. Note that in particular, when $\mathbf{f}$ is deterministic, $\mathbf{f}$ is computable iff it is C-generic and *partial recursive*.

In this paper, we will mainly focus on computable deterministic queries. A more thorough discussion of computable non-deterministic queries defined by IDLOG programs appears in [She90].

Suppose P is a stratified IDLOG program. An *input predicate* of an IDLOG program P is an ordinary predicate q such that q does not appear in the head of any clause in P, and q appears in P or an ID-version of q appears in P, an ordinary predicate p is an *output predicate* of P if p appears in the head of some clause in P. Note that the sort of each input predicate and output predicate is required to be elementary, and the arithmetic predicates are not counted either as input predicates or as output predicates. Now, assume $D = \{d_1, \ldots, d_m\}$, and p_1 (of sort a_1), $\ldots$, p_n (of sort a_n) are input predicates of P. Let $\mathbf{r} = (D, r_1, \ldots, r_n)$ be a database of type $\bar{a} = (a_1, \ldots, a_n)$. Let also udom be a predicate symbol which represents the u-domain of the input database. The program $P \cup \{p_j(t) \;\; 1 \leq j \leq n \;\&\; t \in r_j\} \cup \{\mathsf{udom}(d_i) \;\; i = 1, m\}$ is called a *database program* denoted by $(P, \mathbf{i})$. Throughout this paper, we assume, in addition to the usual axioms of standard logic, the following axioms [Rei83] for database programs:

- *Equality Axioms*,
- *Unique Name Axioms* for each $i \neq j$, $d_i \neq d_j$,
- *Domain Closure Axiom* $\forall^{\mathsf{u}} X \; (X = d_1 \vee \cdots \vee X = d_m)$

Let PERF_Q be the set of all perfect (Herbrand) models of the database program $Q = (P, \mathbf{r})$. Then for each output predicate q of sort a_0, P defines an *IDLOG query* $\mathbf{q}$ of type $\bar{a} \rightarrow a_0$ as follows

$$\mathbf{q}(\mathbf{r}) = \begin{cases} \{q^I \mid I \in \mathsf{PERF}_Q\}, & \text{if } \cup \mathsf{PERF}_Q \text{ is finite} \\ \text{undefined}, & \text{o/w} \end{cases}$$

Example 2 The following IDLOG program P defines the same queries **man** and **woman** that are defined by the disjunctive program $man(\mathrm{X}) \vee woman(\mathrm{X}) \leftarrow person(\mathrm{X})$ (based on the perfect-model semantics of disjunctive programs defined in [Prz88b]).

$sex(\mathrm{X}, 0)$
$sex(\mathrm{X}, 1)$
$man(\mathrm{X}) \leftarrow person(\mathrm{X}), sex[1](\mathrm{X}, 0, 0)$
$woman(\mathrm{X}) \leftarrow person(\mathrm{X}), sex[1](\mathrm{X}, 1, 0)$

The queries **man** and **woman** defined by P are as follows. Suppose $r = \{(a), (b)\}$, and the input database is $(\{a, b\}, r)$. Now, the interpretation for *man* and *woman* in each perfect model of the corresponding database program is either one of the following

$\{man(a), man(b)\}$,
$\{man(a), woman(b)\}$,
$\{woman(a), man(b)\}$,
$\{woman(a), woman(b)\}$

Therefore, $\mathbf{man}(r) = \mathbf{woman}(r) = \{\emptyset, \{(a)\}, \{(b)\}, \{(a), (b)\}\}$ ∎

3.2 Expressive Power

An IDLOG program P is said to be *ID-independent* w r t an output predicate q in P if and only if the query $\mathbf{q}$ defined by P is deterministic The set of all ID-independent programs is not restrictive in terms of expressive power In fact,

Theorem 3 *The class of IDLOG queries defined by stratified ID-independent IDLOG programs is equivalent to the class of computable deterministic queries* ∎

We show in the following how the queries EVEN and EQUAL are defined in IDLOG Let $\mathbf{i} = (D)$, a database containing no relations The query EVEN of type $() \rightarrow 0$ can be represented as[4]

$$\text{EVEN}(\mathbf{r}) = \begin{cases} \emptyset, & \text{if } |D| \text{ is even} \\ D, & o/w \end{cases}$$

The following ID-independent program defines EVEN (there may be some more efficient programs defining EVEN)

$t0(\text{M}) \leftarrow \text{udom}[](\text{N}, \text{M})$
$count\_\text{udom}(\text{M}) \leftarrow t0(\text{M}), \neg t0(\text{N}), +(\text{M}, 1, \text{N})$
$t(0, 0) \leftarrow t0(0)$
$t(\text{L1}, 1) \leftarrow t(\text{L}, 0), t0(\text{L1}) \ +(\text{L}, 1, \text{L1})$
$t(\text{L1}, 0) \leftarrow t(\text{L}, 1), t0(\text{L1}), +(\text{L}, 1, \text{L1})$
$\text{EVEN}(\text{X}) \leftarrow t(\text{L}, 0), count\_\text{udom}(\text{L}), \text{udom}(\text{X})$

Now consider the query EQUAL of type $(0, 0) \rightarrow 0$

$$\text{EQUAL}((D, r, s)) = \begin{cases} (r - s) \cup (s - r), & \text{if } |r| = |s| \\ \emptyset, & o/w \end{cases}$$

Let $count\_pr$ and $count\_ps$ be the predicates which compute (one less than) the size of pr and the size of ps respectively The following ID-independent program defines EQUAL

$\text{EQUAL}(\text{X}) \leftarrow pr(\text{X}), \neg ps(\text{X}), count\_pr(\text{M}), count\_ps(\text{M})$
$\text{EQUAL}(\text{X}) \leftarrow ps(\text{X}), \neg pr(\text{X}), count\_pr(\text{M}), count\_ps(\text{M})$

We verify here that however, in IDLOG, the infinite interpreted domain is not the main cause for the enhanced expressive power Theorem 4 below shows the limitation of stratified DATALOG$^\neg$ in the two-sorted first-order logic defined before

Theorem 4 *EVEN cannot be defined by any stratified DATALOG$^\neg$ program in the two-sorted first-order logic* ∎

Theorem 5 further shows that the non-determinism provided by disjunctive programs does not contribute much to the expressive power either

Theorem 5 *EVEN cannot be defined by any stratified disjunctive DATALOG$^\neg$ program in the two-sorted first-order logic* ∎

Therefore, adding ID-predicates in deductive databases has its own value in enhancing the expressive power Yet, the enhanced expressive power has its own price

Theorem 6 *In general, the problem of deciding whether an IDLOG program is ID-independent w r t a query is undecidable* ∎

However, in the next section, we give a general sufficient test for identifying ID-independent IDLOG programs All the ID-independent programs in this paper can be recognized by this test

3.3 A Sufficient Test for ID-independent Programs

As shown before, the problem of deciding whether an IDLOG program is ID-independent w r t a query is in general undecidable Fortunately, a rather general sufficient test exists for identifying ID-independent IDLOG programs The idea is to first adorn the given program Our algorithm is basically an extension of the one in [RBK88] used for adorning *existential arguments* The input of the algorithm is a stratified IDLOG program (actually, a portion of the program related to the query predicate), and the output is an adorned version of the input For each ID-literal l containing an ID-predicate $p[v]$, the last argument (containing the tid) of l is called the *ID-argument* of l, and the other arguments not in v are called *target arguments* of l The purpose of this adornment process is to test whether there exists an ID-literal l in the input program whose ID-argument is somehow related to any of its target arguments If there is no such ID-literal, then the program is ID-independent w r t the query predicate (Theorem 7)

For each predicate p and each string a of d's and n's, p^a is called an *adorned version* of p In an *adorned version* $\text{p}^{a_1 \ldots a_n}(X_1, \ldots, X_n)$ (resp $\neg\text{p}^{a_1 \ldots a_n}(X_1, \ldots,$

[4] The same query is called ODD in [CH8r]

X_n)) of a literal p(X_1, , X_n) (resp ¬p(X_1, , X_n)), each argument X_i is said to be *adorned as* a_i (which is either d or n) Note that each argument adorned as d by this algorithm is not necessarily an existential argument A clause r is *related to* an output predicate q in the same program P if the head predicate of r appears either in a clause defining q or in a clause related to q (a recursive definition) The *program portion* related to q in P, denoted by P/q, is the set of all clauses in P which are related to q A *pivot* is an adorned version of a predicate prefixed with a positive sign (+) or a negative sign (−) Suppose P is a stratified IDLOG program, and q is one of P's output predicates The algorithm is as follows

Algorithm 1
[Input] P/q
[Output] Q

(1) Initially, S only contains an *unmarked* pivot $+q^{n\ n}$

(2) For each unmarked pivot $\pi = \varepsilon p^a$ in S (ε is either + or −),

- if p is an ordinary predicate, then for each clause that has p in its head, generate an adorned version of that clause with the head predicate p^a as described in ($\star$) below and add it to Q, then we *mark* the pivot π in S,
  - ($\star$) In choosing an adornment for a literal in the body, an argument of a positive literal or a negative ID-literal is adorned as d if the variable in it does not occur anywhere else in the clause, except possibly in an argument in the head adorned as d All other arguments are adorned as n The adorned version of a clause may generate additional unmarked pivots After the adorned version of each clause is created, for each adorned predicate t^c in the body of an adorned clause, $S = S \cup \{+t^c\}$ if t^c appears in a positive literal, and $S = S \cup \{-t^c\}$ if t^c appears in a negative literal (both $+t^c$ and $-t^c$ are unmarked)
- if p is an ID-predicate, say $p[v]$, then for each clause that has p in its head, generate an adorned version of that clause with the head predicate p^b as described in ($\star$) and add it to Q, where b is obtained from a by dropping the last character in a if ε is +, and b is an adornment of all d's if ε is −, then we mark the pivot π in S

∎

The algorithm always terminates since the number of adorned predicates is finite The following theorem provides a sufficient test for identifying ID-independent IDLOG programs

Theorem 7 *Suppose P is a stratified* IDLOG *program and q is an output predicate of P Assume for each ID-literal l in any clause in Q, either the ID-argument of l is adorned as* d *or all the target arguments of l are adorned as* d *Then P is ID-independent w r t q* ∎

Note that the algorithm will adorn as d the only target argument (the first argument) of the ID-literal udom[](N, M) in the program defining EVEN in the previous subsection Therefore, the program is an ID-independent program recognized by this test, and so is the program defining EQUAL However, there may exist some programs which define EVEN (or EQUAL) but cannot be recognized by this test

4 Extended IDLOG Queries

So far, we have seen some functions involving aggregation such as EVEN and EQUAL The input and output of these functions are required not to contain interpreted constants In practical database systems, more general queries involving aggregation are usually used as built-in functions In the following, we show that *aggregate operators*, a general notion of queries involving aggregation, can also be efficiently defined in IDLOG

4.1 Extended Queries

As discussed in [CH80], to answer queries involving aggregation such as "sum up the salaries of all employees" or "how many students are there in a certain class", we need to consider a potentially infinite domain with some intended meaning This is one of our motivations for defining an extended notion of queries Without loss of generality, we assume N to be the interpreted domain An *extended query* is a query which allows the input and output to contain numbers Now the definition of C-generic extended queries is an extension of the one for queries such that the mapping σ is also an identity on N A deterministic extended query **f** is computable iff it is C-generic (for some C) and partial recursive

Theorem 8 *Each deterministic extended query defined by a stratified ID-independent IDLOG program is computable* ∎

4.2 Aggregate Operators

A *multiset*, denoted by {* *}, is a collection of objects that are not necessarily distinct We define aggregate functions on multisets so that duplicates can be counted in evaluation (multisets occur naturally as a result of applying projection operators) An *aggregate function* takes a finite multiset as an argument and produces a single simple value as a result Considered is this paper are the following aggregate functions sum, count, minimum, maximum, even, and odd (the aggregate function average can be simulated through encodings) Suppose u is a tuple of numbers in $\{1, \ldots, n\}$ Then $r[u]$ (resp $t[u]$) denotes the projection of the n-ary relation r (resp the n-ary tuple t) on attributes in u Assume f is an aggregate function, $a = s_1 \ldots s_n$ is a 0, 1 sequence, $b \in \{1, \ldots, n\}$, and $v = (v_1, \ldots, v_m)$ is a tuple of attribute numbers in $\{1, \ldots, n\}$ An *aggregate operator* $<\mathrm{f}, v, b>^a$ is a query of type $(a) \rightarrow s_{v_1} \ldots s_{v_m}$ 1 (the superscript a will be omitted if it is immaterial to the context) For each finite relation r, $<\mathrm{f}, v, b>(r)$ is defined by the following SQL [Ce76, Ull82] query

SELECT v, f(b)
FROM r
GROUP BY v

That is, if r is non-empty, then for each aggregate function f,

$<\mathrm{f}, v, b>(r) = \{tc \mid c = f(s),$ *where* $s =$
$\{* \ t'[b] \mid \exists t' \in r[v \ b] \ s \ t \ t'[v] = t \ *\} \}$

When v is an empty tuple, the above definition reduces to

$<\mathrm{f}, (), b>(r) = \{c \mid c = f(s),$ *where* $s =$
$\{* \ a \mid \exists t \in r \ s \ t \ t[b] = a \ *\} \}$

In the above definitions, the grouping operation is performed only on existing tuples in the argument relation, i e the aggregate function f in the definition gets no empty-set arguments We extend the definition to empty sets by assuming that $<\mathrm{f}, v, b>(\emptyset) = \emptyset$

Example 3 Let $r = \{(a, d, 200), (a, e, 300), (b, f, 500), (c, d, 200), (c, f, 200)\}$ Then $<\mathrm{sum}, (2), 3>(r) = \{(d, \mathrm{sum}(\{* \ 200, 200 \ *\})), (e, \mathrm{sum}(\{* \ 300 \ *\})), (f, \mathrm{sum}(\{* \ 500, 200 \ *\}))\} = \{(d \ 400), (e, 300), (f, 700)\}$ ∎

4 3 Defining Aggregate Operators in IDLOG

Before discussing how IDLOG defines these aggregate operators, we first look at the *hardness* of aggregate operators Let COUNT (resp SUM, etc) denote the class of all aggregate operators of the form $<\mathrm{count}, v, b>$ (resp $<\mathrm{sum}, v, b>$, etc) Among the aggregate operator classes defined before COUNT and SUM are the *hardest* in a sense that if we can define (each aggregate operator in) one of them in a language which is at least as powerful as stratified DATALOG$^\neg$ programs (in a two sorted logic), then in this language we can also define all classes of aggregate operators considered in this paper On the other hand, if a language cannot define the class EVEN (or ODD) then, usually, it cannot define either COUNT or SUM Thus, EVEN can serve as an initial test of whether a language can define all classes of aggregate operators This somehow explains the reason why the queries EVEN and EQUAL defined before have been commonly used for the purpose of exploring the expressive power of various database languages

We say that a class F of aggregate operators subsumes another class G of aggregate operators (w r t stratified DATALOG$^\neg$ programs), written as G $\sqsubseteq$ F, if assuming each aggregate operator in F can be defined by a stratified DATALOG$^\neg$ program, then we can define all aggregate operators in G by stratified DATALOG$^\neg$ programs as well It is easy to see that MAX and MIN can be defined by stratified DATALOG$^\neg$ programs, EVEN $\sqsubseteq$ COUNT , and EVEN and ODD subsumes each other Moreover,

Theorem 9

1 COUNT $\sqsubseteq$ SUM
2 SUM $\sqsubseteq$ COUNT
3 COUNT $\not\sqsubseteq$ EVEN ∎

Therefore, COUNT and SUM are the hardest aggregate operator classes

Now let p be a ternary predicate The following program defines the aggregate operator $<\mathrm{count}, (1), 2>$

$r(\mathrm{X}, \mathrm{A}) \leftarrow p[1](\mathrm{X}, \mathrm{Y}, \mathrm{Z}, \mathrm{A})$
$<\mathrm{count}, (1), 2>(\mathrm{X}, \mathrm{N}) \leftarrow r(\mathrm{X}, \mathrm{M}), \neg r(\mathrm{X}, \mathrm{N}),$
$+(\mathrm{M}, 1, \mathrm{N})$

Accordingly, each aggregate operator can be defined in IDLOG

5 Evaluation of IDLOG queries

There is a number of strategies for evaluating logic queries bottom up (see [BR86] for a survey) Some of these strategies, such as *System Graphs* [KL85, KL86] can be extended to evaluate IDLOG queries The basic idea is to add some mechanism for generating instances of ID-predicates based on the generated instances of their corresponding ordinary predicates ac-

cording to different assignments of tid's For deterministic IDLOG queries, the evaluation process can be in general more efficient Indeed, they are defined by stratified ID-independent IDLOG programs The ID-independence allows us to efficiently evaluate these queries Since each ordinary predicate in a stratified IDLOG program has a priority higher than all its ID-versions and different assignments of tid's do not affect the output, we need to consider only one particular assignment function We can thus choose the one which assigns tid's to tuples of p in accordance with the order they were "created" when p was being evaluated

To answer queries such as "does some tuple satisfy certain properties" or "retrieve a tuple which satisfies certain properties", top-down evaluation (see also [BR86] for a survey) seems more appropriate Besides, in IDLOG, one can also ask *maybe queries* (queries which only ask for a maybe-answer)

Example 4 Suppose there are two binary relations *dept* and *manager* (D, E) $\in$ *dept* iff E is an employee in the department D, and (D, M) $\in$ *manager* iff M is the manager of the department D The following can be thought of as a program defining the maybe query "give the set containing the manager and (any) one employee of each department"

$$dept\_manager(\mathrm{M}, \mathrm{E}) \leftarrow dept(\mathrm{D}, \mathrm{E}), manager(\mathrm{D}, \mathrm{M})$$
$$ans(\mathrm{M}, \mathrm{E}) \leftarrow dept\_manager[1](\mathrm{M}, \mathrm{E}, 0)$$

∎

For queries defined by ID-independent IDLOG programs (w r t the query predicate), maybe queries coincide with usual queries Although not presented in this paper, we note that a resolution-based proof procedure (which is an extension of SLS-resolution [Prz88a]) for evaluating maybe queries has been developed [She90]

6 Conclusion

We presented a non-deterministic deductive database language, a deterministic subset of this language defines all the computable queries More complexity issues and optimizations on the evaluation of queries are to be explored in the future

Acknowledgements

The author is greatly indebted to Michael Kifer for valuable discussions and suggestions which considerably improved this paper The author wishes to thank Yehoshua Sagiv and David Scott Warren for their insightful remarks Thanks are also due to the anonymous referees for helpful comments on the draft of this paper

References

[AB88] S Abiteboul and C Beeri On the power of languages for the manipulation of complex objects *Manuscript*, 1988

[ABW88] K R Apt, H A Blair, and A Walker Towards a theory of declarative knowledge *Foundations of Deductive Database and Logic Programming (ed)*, 1988

[AV87] S Abiteboul and V Vianu A transaction language complete for database update and specification *Proceedings of ACM Symposium on Principles of Database Systems*, 260–268, March 1987

[AV88] S Abiteboul and V Vianu Procedural and declarative database update language *Proceedings of ACM Symposium on Principles of Database Systems*, 240–250, March 1988

[BR86] F Bancilhon and R Ramakrishnan An amateur's introduction to recursive query processing strategies *Proceedings of ACM SIGMOD*, 1986

[Ce76] D D Chamberlin and et al Sequel 2 a unified approach to data definition, manipulation, and control *IBM Journal of Research*, 20(6) 560–575, November 1976

[CH80] A K Chandra and D Harel Computable queries for relational databases *Journal of Computer and System Sciences*, 21(2) 156–178, October 1980

[CH82] A K Chandra and D Harel Structure and complexity of relational queries *Journal of Computer and System Sciences*, 25(1) 99–128, August 1982

[CH85] A K Chandra and Harel Horn clause queries and generalizations *Journal of Logic Programming*, 1 1–15, 1985

[Cha81] A K Chandra Programming primitives for database languages *Proceedings of ACM Symposium on Principles of Programming Languages*, 50 62, 1981

[Cha88] A K Chandra Theory of database queries *Proceedings of ACM Symposium on Principles of Database Systems*, 1–9, March 1988

[Che88] L Chen Extension of datalog with aggregate functions *IV journees bases de Donnees Avancees*, 1988

[Dah87] E Dahlhaus Skolem normal forms concerning the least fixpoint In E Borger, editor, *Computation Theory and Logic, Lecture Notes in Computer Science 270*, pages 101–106, Springer-Verlag, 1987

[GL88] M Gelfond and V Lifschitz The stable model semantics for logic programming *Proceedings of the Fifth Logic Programming Symposium*, 1070–1080, 1988

[HS89] R Hull and J Su Untyped sets, invention, and computable queries *Proceedings of ACM Symposium on Principles of Database Systems*, 347–359, March 1989

[Imm86] N Immerman Relational queries computable in polynomial time *Proceedings of 15th ACM Symposium on Theory of computation*, 347–354, April 1986

[KL85] M Kifer and E L Lozinski *Query optimization in deductive databases* Technical Report 85/16, Computer Science, SUNY at Stony Brook, Stony Brook, NY 11794, 1985

[KL86] M Kifer and E L Lozinski *A framework for an efficient implementation of deductive databases* Technical Report 86/4, Computer Science, SUNY at Stony Brook, Stony Brook, NY 11794, February 1986

[Klu82] A Klug Equivalence of relational algebra and calculus query languages having aggregate functions *JACM 29 3*, 1982

[KP88] P G Kolaitis and C H Papadimitriou Why not negation by fixpoint? *Proceedings of ACM Symposium on Principles of Database Systems*, 231–239, March 1988

[OOM87] G Ozsoyoglu, Z M Ozsoyoglu, and V Matos Extending relational algebra and relational calculus with set-valued attributes and aggregate functions *ACM TODS 12 4*, 1987

[Prz88a] T C Przymusinski On the declarative and procedural semantics of logic programs *Journal of Automated Reasoning*, 1988

[Prz88b] T C Przymusinski On the declarative semantics of deductive databases and logic programs In J Minker, editor, *Foundations of Deductive Databases and Logic Programming*, Morgan Kaufmann, Los Altos, CA, 1988

[RBK88] R Ramakrishnan, C Beeri, and R Krishnamurthy Optimizing existential Datalog queries 1988 Draft

[Rei83] R Reiter Towards a logical reconstruction of relational database theory In M L Brodie, J Mylopoulos, and J Schmidt, editors, *On Conceptual Modelling Perspectives from Artificial Intelligence, Databases and Programming Languages*, Springer-Verlag, 1983

[Saz80] V Sazonov Polynomial computability and recursivity in finite domains *Elektronische Informationverarbeitung und Kybernetik*, 16 319–323, 1980

[She90] Y Sheng The expressive power of deductive database languages with tuple-identifications 1990 in preparation

[Ull82] J D Ullman *Principles of Database Systems* Computer Science Press, second edition, 1982

[Var82] M Y Vardi The complexity of relational query languages *Proceedings of 14th ACM Symposium on Theory of computation*, 137–146, May 1982

[VG88] A Van Gelder Negation as failure using tight derivations for general logic programs In J Minker, editor, *Foundations of Deductive Databases and Logic Programming*, pages 149–176, Morgan Kaufmann, Los Altos, CA, 1988

[VGRS88] A Van Gelder, K A Ross, and J S Schlipf *Well-founded semantics for general logic programs* Technical Report UCSC-CRL-88-16, University of California, Santa Cruz, University of California, Santa Cruz, CA 95064, March 1988

[Zan86] C Zaniolo Safety and compilation of non-recursive horn clauses *Proceedings of of the 1st Expert Database Conf*, 1986