GraphLog: a Visual Formalism for Real Life Recursion

Mariano P. Consens
consens@db.toronto.edu

Alberto O. Mendelzon
mendel@db.toronto.edu

Abstract

We present a query language called GraphLog, based on a graph representation of both data and queries. Queries are graph patterns. Edges in queries represent edges or paths in the database. Regular expressions are used to qualify these paths. We characterize the expressive power of the language and show that it is equivalent to stratified linear Datalog, first order logic with transitive closure, and non-deterministic logarithmic space (assuming ordering on the domain). The fact that the latter three classes coincide was not previously known. We show how GraphLog can be extended to incorporate aggregates and path summarization, and describe briefly our current prototype implementation.

1 Introduction

The literature on theoretical and computational aspects of deductive databases, and the additional power they provide in defining and querying data, has grown rapidly in recent years. Much less work has gone into the design of languages and interfaces that make this additional power available in a convenient form. We propose here a language called *GraphLog*, based on a graph representation of both databases and queries. Graphs are a very natural representation for data in many application domains; for example, transportation networks, project scheduling, parts hierarchies, family trees, concept hierarchies, and *Hypertext*. GraphLog has evolved from the earlier language G^+ proposed in [CMW88]; it differs from G^+ in its more general data model, use of negation, and computational tractability.

GraphLog queries ask for patterns that must be present or absent in the database graph. Each such pattern, called a *query graph*, defines a set of new edges (i.e., a new relation) that are added to the graph whenever the pattern is found. GraphLog queries are sets of query graphs, called *graphical queries*. An edge used in a query graph either represents a base relation or is itself defined in another query graph. GraphLog is well suited to a graphical interface in which users draw query graphs on a screen. A prototype implementation of such an interface is described in Section 5.

The simplicity and power of the language rest on two main principles. The first is that edges in a query graph match edges or paths in the database graph. Regular expressions are used to qualify these paths. The second is that there is no recursive definition of new edges in a GraphLog query;

The first author was partially supported by the PEDECIBA, United Nations Programme for the Development of Basic Sciences, Uruguay.
Published in *Proceedings of the Nineth ACM SIGACT–SIGMOD Symposium on Principles of Database Systems*, 1990, pp. 404–416.

any edge used in a query graph must have been defined elsewhere without directly or indirectly referring to the new edge being defined in the current query graph.

The design of GraphLog makes the deliberate choice of avoiding the full power of Horn clauses with stratified negation. It is therefore important to characterize the set of expressible queries. In achieving this characterization, we have as a side benefit clarified the relationship among three well-known query classes: queries expressible in stratified linear Datalog, queries computable in non-deterministic logarithmic space, and queries expressible with a transitive closure operator plus first-order logic, all of which turn out to coincide with the class expressible in GraphLog.

The fact that there is a close relationship between linear Datalog and transitive closure has become somewhat of a folk theorem in recent years. In [JAN87], it was shown how to evaluate a linear rule by a transitive-closure-like computation. The procedure is quite complex, due to the optimizations that the technique is intended to achieve. Negation is not mentioned. Ullman [Ull89] gives a construction to map a single linear rule into a transitive closure problem. He only considers rules with a single IDB predicate in the body, and no negation. The connection between linear programs and transitive closure is also considered in [CK86], and in fact the proof of their Theorem 2 can be used to derive one of our results when negation is not considered.

Our result establishes a simple translation procedure that takes into account stratified negation. Given any linear program, this procedure outputs a program in which all recursive rules are transitive closures. We then use Immerman's results on the power of the transitive closure operator to show that the queries computed by stratified linear Datalog programs (assuming an ordering on the domain) are exactly those computable in non-deterministic logarithmic space on the size of the database. Without an ordering of the domain, we show that linear Datalog programs have non-deterministic logarithmic space data complexity (i.e., L-DATALOG $\subset$ QNLOGSPACE), improving on results from [UvG86, Kan87] that showed that[1] L-DATALOG $\subset$ QNC.

These results reflect favourably on the design of GraphLog, since linear stratified Datalog is believed by many to express most "real life" recursive queries. We have also incorporated to the language the ability to compute aggregate functions and to summarize along paths. As a test case, we have considered in detail the application of GraphLog to one particular domain, *Hypertext* systems, with encouraging results, which are described in [CM89].

2 The GraphLog Query Language

Many databases can be naturally viewed as graphs. In particular, any relational database in which we can identify one or more sets of objects of interest and relationships between them can be represented by mapping these objects into nodes and relationships into edges.

Example 2.1: Figure 1 shows a graph representation of a flights schedule database. Each flight number is associated with the cities it connects (by the predicates from and to), as well as with the departure and arrival times of the flight (by the predicates departure and arrival, respectively). There is also a unary predicate capital that tells which cities are national capitals. □

[1] Actually, it was known that L-DATALOG $\subset$ QNC^2 from the fact that L-DATALOG $\subset$ STAGE$(\log n)$ and STAGE$(\log n)$ $\subset$ QNC^2. Our result shows that STAGE(TC-DATALOG) = STAGE(L-DATALOG).

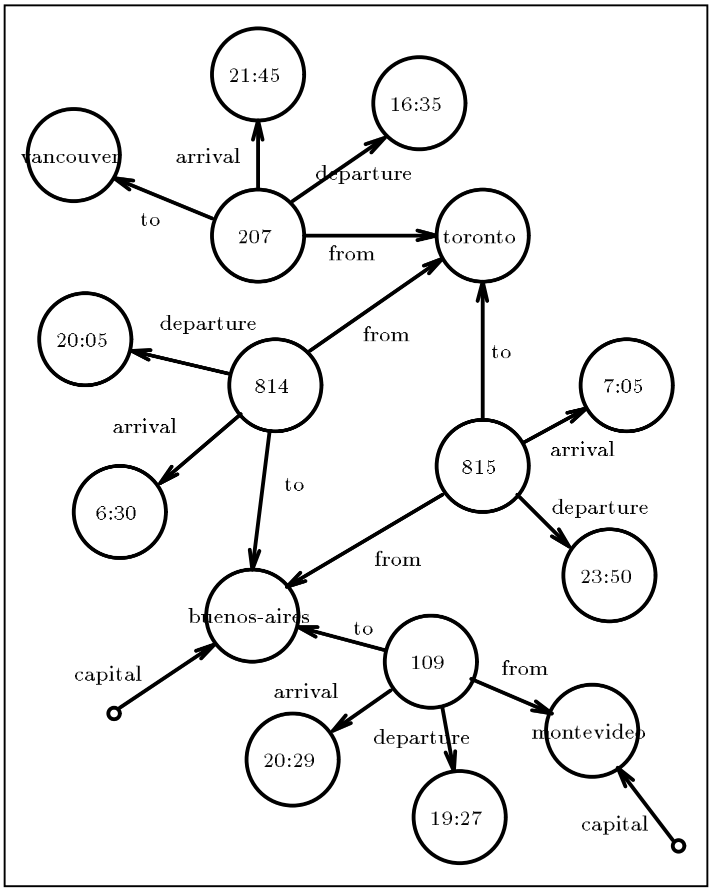

Figure 1: A graph representation of a flights schedule database.

In the example, we can think of labels such as from and to as edge types, carrying no additional information. In other cases, we would like additional attributes on both nodes and edges; for example, we might have chosen to represent a flight as a relationship between two cities that gives the departure and arrival times, labeling the edge with a literal of the form flight(21:45,23:15). In general, a tuple $P(a_1, \ldots, a_i, b_1, \ldots, b_j, c_1, \ldots, c_k)$ can be represented by an edge between nodes (a1,...,ai) and (b1,...,bj) labelled P(c1,...,ck). The precise definition of the kind of graphs that we are considering for representing databases is given below.

Definition 2.1: A *directed labeled multigraph* G is a septuple

$$(N, E, L_N, L_E, \iota, \nu, \epsilon)$$

where: N is a finite set of *nodes*; E is a finite set of *edges*; L_N is a set of *node labels*; L_E is a set of *edge labels*; ι, the *incidence function*, is a function from E to N^2 that associates with each edge in E a pair of nodes from N; ν, the *node labeling function*, is a function from N to L_N that associates with each node in N a label from L_N; ϵ, the *edge labeling function*, is a function from E to L_E that associates with each edge in E a label from L_E. $\square$

We would like to profit from a graph based representation of a database to express queries. In analogy with tableau queries, which define a relation by means of a tuple of "distinguished variables" that is in the answer whenever some "template" of tuples is present in the database, we want our expressions to describe a "graph pattern" that should be present for a "distinguished edge" to define a relation.

Definition 2.2: A *directed labeled multigraph with a distinguished edge* $G_{\epsilon(e)}$ is a ninetuple

$$(N, E, L_N, L_E, \iota, \nu, \epsilon, e, L_e)$$

where $(N, E \cup \{e\}, L_N, L_E \cup L_e, \iota, \nu, \epsilon)$ is a directed labeled multigraph, $\epsilon(e) \in L_e$ and $\epsilon(e') \in L_E$ for each e' in E. □

Below, we will formally define our "graph patterns" as directed labeled multigraphs with a distinguished edge, without isolated nodes, and having the following properties. The nodes are labeled by sequences of variables. Each edge is labelled by a literal (i.e. positive or negative occurrence of a predicate applied to a sequence of variables and constants) or by a *closure literal*, which is simply a literal followed by the positive closure operator. That is, if s is a literal, then s^+ is a closure literal. Closure literals may only appear between nodes labelled by sequences of the same length. There is a distinguished edge, which can only be labelled by a positive non-closure literal.

Definition 2.3: A *query graph* G_p is a directed labeled multigraph with a distinguished edge

$$(N, E, L_N, L_E, \iota, \nu, \epsilon, e, L_e)$$

where L_N is a set of sequences of variables, L_E is a set of literals and closure literals, L_e is a set of positive literals, there are no isolated nodes (i.e., for each $n \in N$ there exist $n' \in N, e' \in E$ such that either $\iota(e') = (n, n')$ or $\iota(e') = (n', n)$), and such that for each $e' \in E \cup \{e\}$ with $\iota(e') = (n_1, n_2)$, $|\nu(n_1)| = k_1$, $|\nu(n_2)| = k_2$, and $\epsilon(e') = s$, where s has l positions, the arity of s is $k_1 + k_2 + l$ and if $k_1 \neq k_2$, then s is not a closure literal. [2] □

Example 2.2: Figure 2 shows a query graph. There are several different kinds of literals labeling the edges. The literal not-desc-of(P2) is the (necessarily positive) literal labeling the distinguished edge. The distinguished edge of the query graph is graphically represented by a bold line. The literal descendant$^+$ labeling the edge between P1 and P3 is a positive closure literal (notice the dashed edge; edges labeled with closure literals are drawn as dashed lines). The literal $\neg$descendant$^+$ labeling the edge between P2 and P3 is a negative closure literal. Note that we are representing negative literals by crossing over the edges labeled with them, and showing the negative literals as if they were positive ones. Finally, person is a unary predicate.

Intuitively, this query graph expresses the query that returns a ternary predicate not-desc-of(P1,P3,P2) with the descendants P3 of person P1 who are not descendants of person P2. □

At the beginning of this section we mentioned our intentions of interpreting a query graph as a graph pattern that, when satisfied, defines a relation by means of its distinguished edge. We would

[2] Observe that we do not force a one-to-one mapping between nodes and node labels; i.e., the same sequence of variables can appear in more than one node. There are circumstances (e.g., a cluttered picture, faster editing of graphs), in which repeating node labels can improve readability. However, we believe that the one-one correspondence should be preferred: it is far more intuitive to identify nodes with variables.

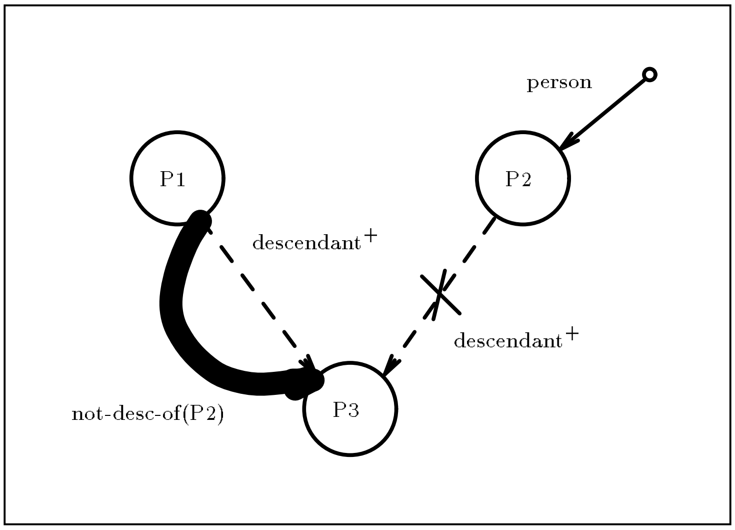

Figure 2: The descendants of P1 which are not descendants of P2.

like to read a query graph as follows: "if each of the edges in the graph pattern is present, then the relation defined by the distinguished edge holds."

This interpretation naturally corresponds to the way we read the rules of a logic program. The correspondence requires mapping each edge to an appropriate predicate. Mapping edges labeled with closure literals requires particular attention. We will match the edges of the query graph labeled with closure literals to paths in the graph of the database. Hence, the corresponding predicate will be defined by rules expressing the transitive closure of the predicate in the literal labeling the edge.

In light of the above discussion, we define a translation function that associates a stratified Datalog program with a query graph.

Definition 2.4: The *logical translation* function λ is a function from query graphs to logic programs given by $\lambda(G_{p_0}) = \mathcal{P}$ where

$$G_{p_0} = (N, \{e_1, \ldots, e_k\}, L_N, L_E, \iota, \nu, \epsilon, e_0, L_{e_0})$$

and such that the rule

$$s_0 \leftarrow s_1, \ldots, s_k. \tag{1}$$

is in $\mathcal{P}$, where, for $0 \leq i \leq k$, if $\epsilon(e_i) = s, \iota(e_i) = (n_1, n_2), \nu(n_1) = \overline{X_1}$ and $\nu(n_2) = \overline{X_2}$, then

1. if s is $p(\overline{X_3})$ (resp. $\neg p(\overline{X_3})$), then s_i is $p(\overline{X_1}, \overline{X_2}, \overline{X_3})$ (resp. $\neg p(\overline{X_1}, \overline{X_2}, \overline{X_3})$),
2. if s is $=$ (resp. $\neq$), then s_i is $\overline{X_1} = \overline{X_2}$ (resp. $\overline{X_1} \neq \overline{X_2}$)[3],
3. if s is $p(\overline{X_3})^+$ (resp. $\neg p(\overline{X_3})^+$), then s_i is $p'(\overline{X_1}, \overline{X_2}, \overline{X_3})$ (resp. $\neg p'(\overline{X_1}, \overline{X_2}, \overline{X_3})$) and the following two rules are also in $\mathcal{P}$:

$$p'(\overline{X}, \overline{Y}, \overline{W}) \leftarrow p(\overline{X}, \overline{Y}, \overline{W}). \tag{2}$$
$$p'(\overline{X}, \overline{Y}, \overline{W}) \leftarrow p(\overline{X}, \overline{Z}, \overline{W}), P'(\overline{Z}, \overline{Y}, \overline{W}). \tag{3}$$

[3]More precisely, the notation stands for a sequence of equality (resp. inequality) atoms, one for each pair of variables in the same component of each sequence. Note, however, that equality atoms are seldom (if ever) used.

```
not-desc-of(P1,P3,P2) ← descendant-tc(P1,P3),
                        ¬ descendant-tc(P2,P3),
                        person(P2).

descendant-tc(X,Y) ← descendant(X,Y).
descendant-tc(X,Y) ← descendant(X,Z), descendant-tc(Z,Y).
```

Figure 3: The descendants of P1 which are not descendants of P2, in Datalog.

where $|\overline{X}| = |\overline{Y}| = |\overline{Z}| = |\overline{X_1}| = |\overline{X_2}|$, $|\overline{W}| = |\overline{X_3}|$ and there are no repeated variables in $|\overline{X}|$, $|\overline{Y}|$, $|\overline{Z}|$, $|\overline{W}|$.

and no other rules are in $\mathcal{P}$. □

Example 2.3: The program listed in Figure 3 is the result of applying the logical translation function λ to the query graph of Figure 2.

The closure literals descendant$^+$ are translated to an IDB predicate descendant-tc(X,Y) . One more rule defining not-descendant-of(P1,P2,P3) has one literal corresponding to each edge in the query graph. These literals appear negated, if the edge label is a negative literal, or they are IDB's defined by other rules, like descendant-tc(X,Y), if the edge label is a closure literal. □

A query graph corresponds to a logical rule, perhaps with some other rules expressing a slight generalization of transitive closure. The next step to specify a query is, of course, to express a set of rules by means of a set of graphs.

Definition 2.5: A *graphical query* $\mathcal{G}$ is a finite set of query graphs whose edge labels contain two classes of predicate symbols: *IDB predicates*, denoted $p, p_1, p_2, \ldots$, which are the ones that appear in the label of the distinguished edge of some query graph; and *EDB predicates*, denoted $q, q_1, q_2, \ldots$, which are the ones that do not appear in the label of the distinguished edge of any query graph. □

As with logical programs, we will associate with a graphical query a graph with information about its structure.

Definition 2.6: The *dependence graph* of a graphical query $\mathcal{G}$ is a directed graph whose nodes are the IDB and EDB predicates that appear in the edge labels of the query graphs in $\mathcal{G}$ and such that there is an edge from p_j (resp. q_j) to p_i iff there is a query graph G_{p_i} in $\mathcal{G}$ whose distinguished edge is labeled p_i and which has p_j (resp. q_j) labeling some non-distinguished edge of G_{p_i}. □

The definition of the logical translation function λ (defined for query graphs) is extended to operate on graphical queries by simply taking the union of the rules generated for each query graph. The

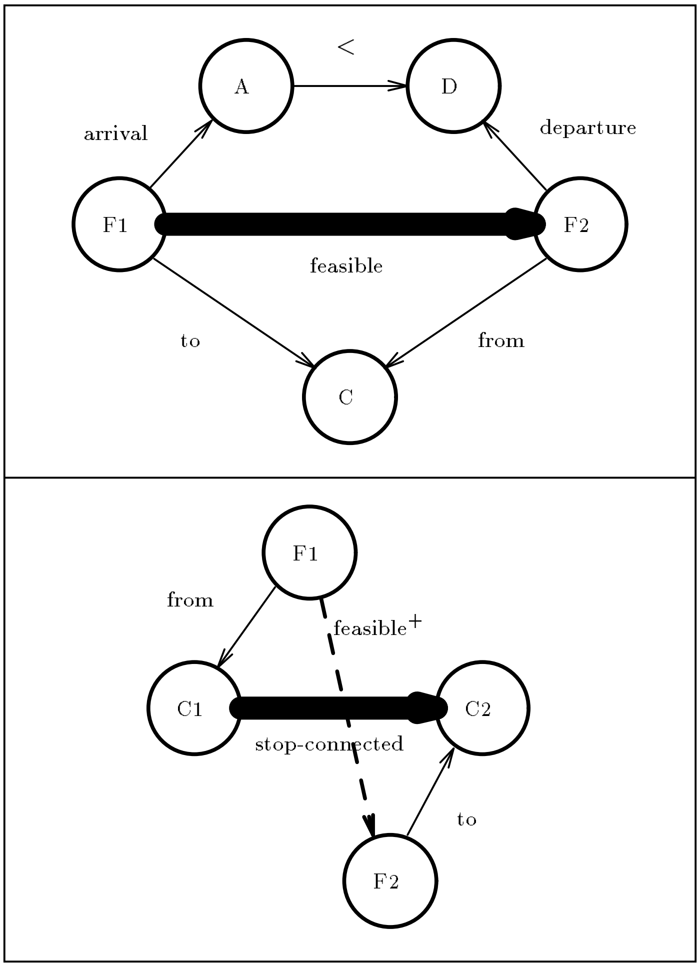

Figure 4: Feasible flight connections.

semantics of a graphical query is determined by the usual semantics for the associated set of stratified Datalog rules.

We allow as expressions of the GraphLog query language only those graphical queries with an acyclic dependence graph. Note that, although we disallow explicit recursion, recursion is nevertheless implicit in the use of closure literals.

Definition 2.7: *Graphlog* is the query language defined by the set of graphical queries $\mathcal{G}$ whose dependence graph is acyclic. The meaning of a graphical query $\mathcal{G}$ is the meaning of the program $\lambda(\mathcal{G})$ under stratified Datalog semantics. $\Box$

Example 2.4: Figure 4 shows a graphical query with two query graphs (each query graph is contained in a separate region within the box of its graphical query). The query applies to the database of Figure 1. The first query graph defines a predicate feasible(F1,F2) between two flights whenever the first flight arrives before the departure of the second. The second defines a predicate stop-connected(C1,C2) between two cities whenever there is a sequence of at least two feasible flights between them.

If we wanted to include directly connected cities, we would have to draw another query graph. Below we will consider means of avoiding drawing several query graphs in situations like this.

□

The language can be made considerably more concise, without changing the semantics, by generalizing literals and closure literals to arbitrary regular expressions. We do this step by step as follows. Each of the new operators introduced is definable in terms of the basic language and is added only for convenience.

If we want an edge in a query graph to represent either predicate p_1 or p_2 we use the operator "|", denoting the *alternation* of the predicates p_1 and p_2 by $p_1|p_2$. Note that care must be taken not to use variables that do not appear in both p_1 and p_2, referred to as *ghost variables*, elsewhere in the query graph, because they "vanish" from the IDB predicate that replaces the use of alternation. The expression $p_1|p_2$ constitutes the *scope* of the ghost variables in it. Therefore, a ghost variable must never occur outside its scope.

Another useful shortcut is to avoid drawing several edges and nodes to represent a path along which the variables labeling the nodes will not be used elsewhere. We would like to simply write the sequence of predicates that appear along the path in just one edge. In order to do this we introduce the *composition* of the predicates p_1 and p_2, denoted by p_1p_2.

We will also introduce an operator that "changes the direction of the arrow" in an edge of a query graph, the *inversion* of the predicate p_1, denoted $-p_1$. The usefulness of the inversion operator can be appreciated when combined with composition.

Our last addition is particularly useful when used together with closure. If we have a closure literal $p(X)^+$ labeling an edge of a query graph, it stands for a path in the graph representation of the database along which the ground literals $p(a)$ always will have the same value a, for some a such that an appropriate valuation θ has $\theta(X) = a$. If we want to have a sequence of ground literal labels whose values are not all the same along the path, but can be arbitrary, then we have to project out the component with variable X in predicate p. We want to avoid this extra query graph. The *underscore* (_) will stand in place of the variables that we want to project out.

All the constructs we have considered so far are considerably more useful when used in combination with each other. We will define below the set of expressions that results from their combined usage.

Definition 2.8: A *path regular expression* (*p.r.e.*, for short) is an expression generated by the grammar

$$E \leftarrow S; (E)^+; -(E); \neg(E); (E|E); (EE)$$

where S stands for any literal. □

Two new operators can be defined in term of the ones we have already seen: the *Kleene closure* of a p.r.e. E, denoted $(E)^*$, which is equivalent to the expression $(= |(E)^+)$, and the *optional* operator (?), that when applied to a p.r.e. E yields $(E)?$, which is equivalent to the expression $(= |E)$.

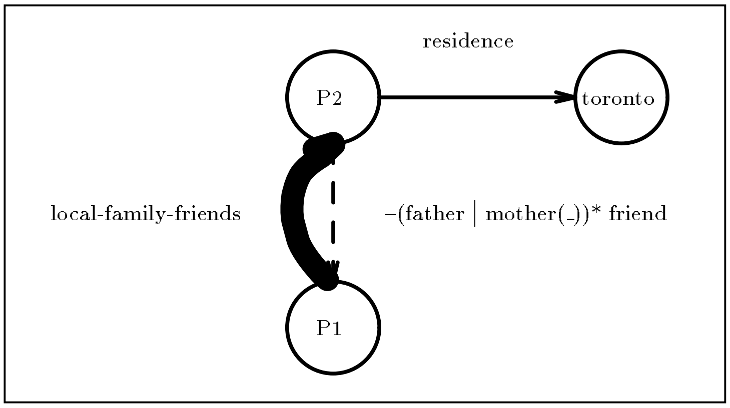

Figure 5: Finding the local family friends.

There are two aspects of p.r.e.'s that deserve attention. The first one is that restrictions apply in the use of ghost variables, the other is that the presence of negation allows universal quantification of paths within one query graph.[4]

From now on we will consider as GraphLog expressions the graphical queries that contain query graphs with p.r.e. in labels, and/or use the underscore in literals; we know these extensions do not add anything to the language, but convenience and succinctness.

Example 2.5: Suppose we are interested in finding among our friends, and the friends of our ancestors, people who live in Toronto There is a residence(P,L) predicate meaning that person P lives in city L. There is no parent predicate, but two predicates: father(P1,P2) representing that P1 is the father of P2, and mother(P1,P2,H) representing that P1 is the mother of P2 and giving also the name of the hospital H at which the relationship started.

Figure 5 shows a graphical query with only one query graph having two nodes. Without p.r.e.'s, it would have been necessary to use three query graphs, one of them with four nodes.

Notice that we used the underscore in the position corresponding to the hospital in the mother relation. Had we used a variable H in that position, the variable H would have been a ghost variable and we would have asked for either ancestors that are fathers or ancestors that are mothers all of which had their descendants in the same hospital. □

Example 2.6: Figure 6 represents a query in a software development environment. Nodes represents software modules, functions, or libraries. The predicate in-module(F,M) means that function F belongs to module M. Predicate calls-local(F1,F2) means that function F1 calls some function F2 defined locally, i.e. within the same module. Predicate calls-extn(F1,F2) means F1 calls external function F2. Finally, in-library(F,L) means that function F is in library L. The query shown in Figure 6 defines predicate self-used(M), meaning that module M uses directly or indirectly the async-io library and M calls itself indirectly through some other modules. □

[4] Note that, with the translation to Datalog that we have been considering so far, safety of the equivalent logic program requires restricting the occurrence of negation to the outermost subexpression of a p.r.e.

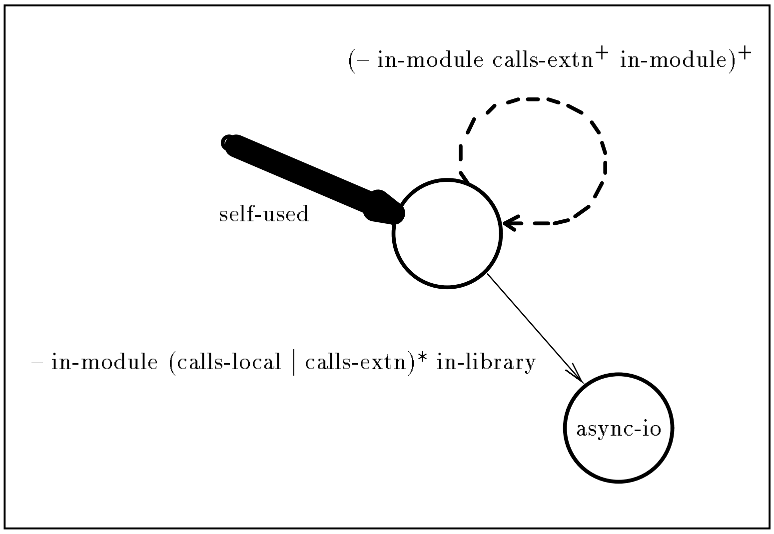

Figure 6: Circularly used modules invoking code from the "async-io" library.

3 Expressive Power

In this section we present results on the expressive power of GraphLog. A more detailed exposition can be found in [Con89]. We first introduce the transitive closure queries and some of their properties.

Definition 3.1: A *transitive closure formula* is a formula $TR\phi(\overline{x}; R)$, where $\phi(\overline{x}; R)$ is a domain relational calculus query such that $\overline{x}$ is an sequence of variables of even length and R is free in $\phi(\overline{x}; R)$ but bound in $TR\phi(\overline{x}; R)$ by the *transitive closure operator* T. The meaning of $TR\phi(\overline{x}; R)$is given by the transitive closure of $\phi(\overline{x}; R)$. □

The relational calculus extended with transitive closure formulas is denoted TC.

There is one relationship between complexity classes and sets of queries expressed by a language that is particularly relevant to our work. Consider the language PTC obtained from TC by allowing only positive (i.e. not within any negation signs) applications of the transitive closure operator. The following result, which requires the presence of an order relation on the domain, is from [Imm87].

Lemma 3.1: QNLOGSPACE $=$ PTC$^{<}$

The set of queries expressed by formulas of the form $\overline{z}TR\phi(\overline{x}; R)$ (where $\overline{z}$ is a vector of not necessarily distinct free variables that are substituted for the components of the transitive closure), such that $\phi(\overline{x}; R)$ is in E (where E is the set of existential first order queries), is denoted TE. In the presence of an order ($<$) in the domain and of two constants in the language, 0 and m, denoting resp. the first and last (in $<$) values of the domain[5], it was shown in [Imm87] that PTC$^{c,<}$ $=$ TE$^{c,<}$.

[5]Notice that $x = 0$ iff $\neg\exists y(y < x)$, which is a first order formula. Hence, by introducing a FO formula we can get rid of the constants 0 and m.

A surprising result in [Imm88a, Imm88b] is that $\text{PTC}^{<} = \text{TC}^{<}$, that is, $\text{PTC}^{<}$ is closed under complement. Together with Lemma 3.1 it shows that nondeterministic logspace is closed under complement, a formerly long-standing open problem in complexity theory, that, for instance, answers whether the context sensitive languages are closed under complement [AU79]. We collect in the next theorem the consequences of this result.

Theorem 3.1: [Imm88a, Imm88b] $\text{QNLOGSPACE} = \text{TC}^{c,<} = \text{TE}^{c,<}$

When we are not provided with an order relation, a result in [CH82] shows that the QLOGSPACE query that tests whether the size of the database domain is even is not a transitive closure query. Hence, we can conclude:

Lemma 3.2: $\text{TC} \subset \text{QNLOGSPACE}$

Now we present the definition of several special cases of Datalog programs.

Definition 3.2: A *linear* logic program is one in which each rule has at most one recursive subgoal.[6]

A *TC logic program* is a linear program in which each recursive IDB predicate p is the head of exactly two rules of the form:

$$\begin{array}{lcl} r_1 & : & p(X_1,\ldots,X_n,Y_1,\ldots,Y_n) \leftarrow p_0(X_1,\ldots,X_n,Y_1,\ldots,Y_n). \\ r_2 & : & p(X_1,\ldots,X_n,Y_1,\ldots,Y_n) \leftarrow p_0(X_1,\ldots,X_n,Z_1,\ldots,Z_n), p(Z_1,\ldots,Z_n,Y_1,\ldots,Y_n). \end{array}$$

Rules r_1 and r_2 are referred to as *TC rules.* □

The set of queries expressed by linear (resp. TC) Datalog programs will be denoted by L-DATALOG (resp. TC-DATALOG).

The parallel complexity of logic programs was studied in [UvG86, Kan87]. It was shown there that $\text{L-DATALOG} \subset \text{QNC}$ and that there are P-complete problems that are expressible in DATALOG. Therefore, linear Datalog programs do not express all DATALOG queries, unless $\text{NC} = \text{PTIME}$ (a very unlikely fact).

When we only consider stratified linear (resp. TC) logic programs as expressions we get a restriction of S-DATALOG denoted SL-DATALOG (resp. STC-DATALOG). The following relation between the transitive closure queries and the stratified TC Datalog queries holds:

Lemma 3.3: $\text{TC} = \text{STC-DATALOG}$

To study the expressive power of GraphLog, we first relate it to the stratified Datalog sublanguages we have been considering.

[6] These are called piecewise linear programs in [Ull89], where the term linear logic program is used to refer to programs in which each rule has at most one IDB subgoal.

Lemma 3.4: TC = STC-DATALOG $\subseteq$ GRAPHLOG $\subseteq$ SL-DATALOG $\subseteq$ S-DATALOG

If we denote by MGRAPHLOG the subset of GRAPHLOG whose graphical queries do not have negated literals in any edge of the query graphs, we have the following Corollary.

Corollary 3.1: TC-DATALOG $\subseteq$ MGRAPHLOG $\subseteq$ L-DATALOG $\subseteq$ DATALOG

Our main contribution in this section is to improve on the results of Lemma 3.4 and present several consequences. To this end we will give an algorithm that translates any stratified linear Datalog program into an equivalent stratified TC Datalog program. The idea behind the translation comes from an informal description of the evaluation of a single linear rule by a transitive closure computation given in [Ull89].

We will make use of constants in the translation process to represent a "signature" for predicates. Signatures allow the codification of several predicates into a wider one that carries the information of all the predicates; the signature tells which tuples in the wider predicate are from each of the original predicates.

The reason for introducing constants is that they simplify the signature mechanism. We emphasize that constants are not necessary at all. Signature techniques that do not make use of constants (but at the expense of increased complexity) have been developed and presented elsewhere: [CH85] presents a technique using inequalities and [Shm87] develops a signature mechanism for Datalog without equalities nor inequalities.

In light of the above discussion, we will present both the algorithm and the results following it without paying attention to the presence of constants.

Algorithm 3.1: Translation of SL-DATALOG into STC-DATALOG.

INPUT: A stratified linear Datalog program $\mathcal{P}_p$.

OUTPUT: An equivalent stratified TC Datalog program $\mathcal{P}'_p$.

METHOD: The first step is to determine the dependence graph of $\mathcal{P}_p$. Let $S_1, \ldots, S_{l'}$ be the strongly connected components (SCC's) of $\mathcal{P}_p$. Then choose predicates $e_1, \ldots, e_{l'}$ and $t_1, \ldots, t_{l'}$ that are not in $\mathcal{P}_p$.

The remaining procedure for finding $\mathcal{P}'_p$ is given in Figure 7. We make use of $n+1$ distinct constants $c, c_1, \ldots, c_n$, where n is the maximum of the arities of the IDB's of $\mathcal{P}_p$. The notation c^k stands for a sequence of k constants c. $\square$

Example 3.1: Figure 8 shows the well-known "same generation" example. When given as input to Algorithm 3.1 the result is the program in Figure 9.

The algorithm creates an edge from the start node (c,c,c) to the nodes reachable by the initialization rule for sg as rule r'_2 in the procedure of Figure 7. Note that the constant sg is the signature for the

for each SCC S_l of $\mathcal{P}_p, 1 \leq l \leq l'$ **do**
  if there are no recursive rules in S_l **then**
    for each rule r whose head is a predicate in S_l **do** add r to $\mathcal{P}'_p$
  else begin
    let m be the maximum arity of the predicates $p_1, \ldots, p_n$ in S_l
    for each recursive rule r_1 in $\mathcal{P}_p$
      $r_1 : p_i(X_1, \ldots, X_{n_i}) \leftarrow p_j(Y_1, \ldots, Y_{n_j}), s_1, \ldots, s_k.$
      do add the rule r'_1 to $\mathcal{P}'_p$
      $r'_1 : e_l(Y_1, \ldots, Y_{n_j}, c_j^{m-n_j+1}, X_1, \ldots, X_{n_i}, c_i^{m-n_i+1}) \leftarrow s_1, \ldots, s_k.$
    for each non-recursive rule r_2 in $\mathcal{P}_p$
      $r_2 : p_i(X_1, \ldots, X_{n_i}) \leftarrow s_1, \ldots, s_k.$
      do add the rule r'_2 to $\mathcal{P}'_p$
      $r'_2 : e_l(c^m, X_1, \ldots, X_{n_i}, c_i^{m-n_i+1}) \leftarrow s_1, \ldots, s_k.$
    add the TC rules for predicate t_l to $\mathcal{P}'_p$
      $t_l(\overline{X}, \overline{Y}) \leftarrow e_l(\overline{X}, \overline{Y}).$
      $t_l(\overline{X}, \overline{Y}) \leftarrow e_l(\overline{X}, \overline{Z}), t_l(\overline{Z}, \overline{Y}).$
    for each predicate p_i in $S_l, 1 \leq i \leq n$
      do add the rule r'_3 to $\mathcal{P}'_p$
      $r'_3 : p_i(\overline{X}) \leftarrow t_l(c^m, \overline{X}, c_i^{m-n_i+1}).$
  end

Figure 7: Translation of SL-DATALOG into STC-DATALOG.

```
sg(X,X) ← person(X).

sg(X,Y) ← parent(X,Z), sg(Z,W), parent(Y,W).
```

Figure 8: The "same generation" query, in linear Datalog.

```
e(Z,W,sg,X,Y,sg) ← parent(X,Z),parent(Y,W).

e(c,c,c,X,X,sg) ← person(X).

t(X1,X2,X3,Y1,Y2,Y3) ← e(X1,X2,X3,Y1,Y2,Y3).
t(X1,X2,X3,Y1,Y2,Y3) ← t(X1,X2,X3,Z1,Z2,Z3), t(Z1,Z2,Z3,Y1,Y2,Y3).

sg(X,Y) ← t(c,c,c,X,Y,sg).
```

Figure 9: The "same generation" query, in TC Datalog.

predicate sg(X,Y) both in predicates e and t. Then, the edges corresponding to the linear recursive rule for the sg predicate are added to the program as rule r'_1 in the procedure of Figure 7. The rule defining predicate sg in the TC Datalog program (i.e., the one introduced as as rule r'_3 in the procedure of Figure 7) as well as the TC rules for predicate t express that whenever there is a path from initial values for sg to some node (X,Y), then (X,Y) is in sg. □

The correctness of Algorithm 3.1 can be proved by showing the equivalence of the input and output logic programs by an induction on the iterations of the naive evaluation within each strongly connected component of the dependency graphs of the programs.

Theorem 3.2: Algorithm 3.1 correctly translates any SL-DATALOG program into an equivalent STC-DATALOG program in time polynomial in the size of the input.

Corollary 3.2: Algorithm 3.1 correctly translates any L-DATALOG expression into an equivalent TC-DATALOG program.

Theorem 3.3: TC = STC-DATALOG = GRAPHLOG = SL-DATALOG

Proof: The existence of Algorithm 3.1 shows that SL-DATALOG $\subseteq$ STC-DATALOG. The results of Lemma 3.4 conclude the proof. □

As a consequence of the above theorem and of Lemma 3.2, we can bound the data complexity of GraphLog.

Lemma 3.5: GRAPHLOG $\subset$ QNLOGSPACE $\subseteq$ QNC

The result of Theorem 3.3 can be specialized for monotone queries.

Corollary 3.3: TC-DATALOG = MGRAPHLOG = L-DATALOG

Proof: The result is a consequence of the property of Algorithm 3.1, presented in Corollary 3.2, together with Corollary 3.1. $\square$

We are finally in a position to improve on the data complexity result for linear Datalog.

Lemma 3.6: L-DATALOG $\subset$ QNLOGSPACE

Proof: From Lemma 3.3 we have that L-DATALOG = TC-DATALOG. By definition TC-DATALOG $\subset$ STC-DATALOG and Lemma 3.3 shows that STC-DATALOG = TC. Considering the result of Lemma 3.2 that proves that TC $\subset$ QNLOGSPACE, we conclude that L-DATALOG $\subset$ QNLOGSPACE. $\square$

The results on complexity can be tightened in the presence of an order relation.

Lemma 3.7: $\text{TC}^{<}$ = $\text{STC-DATALOG}^{<}$ = $\text{GRAPHLOG}^{<}$ = $\text{SL-DATALOG}^{<}$ = QNLOGSPACE

Proof: By Lemma 3.1 we know that QNLOGSPACE = $\text{PTC}^{<}$. By definition PTC $\subseteq$ TC. Together with the result of Lemma 3.2, we conclude that QNLOGSPACE = $\text{TC}^{<}$. Theorem 3.3, which is not affected by the presence of an order relation, completes the proof of the chain of equalities in the hypothesis. $\square$

An interesting consequence from Theorem 3.3 and Theorem 3.1 is that any graphical query in $\text{GRAPHLOG}^{c,<}$ has an equivalent one in which only one edge has a closure literal label.

We know that there is no algorithm that given any linear Datalog program will find an equivalent TC Datalog program with only one application of transitive closure: there exists a property (having a "one-sided" equivalent program) that is undecidable for the former class but decidable for the latter [Nau87]. On the other hand stratified linear programs (with constants and an order relation) collapse into equivalent programs with only one application of transitive closure, by the same argument applied in the previous paragraph.

The diagram of Figure 10 summarizes the relationships between the expressive power of the query languages we have considered. The sets of queries expressed by relational calculus, fixpoint formulas and Datalog with inflationary semantics, are denoted FO, FP and $\text{DATALOG}^{\neg}$, respectively. The relative height within the picture represents the expressive power of the query languages shown (omitting technical details like presence of an order relation to simplify the diagram), but the sets of queries at the right hand side of the diagram are monotone, hence incomparable with the ones at the left hand side. Our main result has been to prove that the languages enclosed within each of the two large ellipses have equivalent expressive power.

4 Aggregation and Summarization

To justify our claim of "real life" queries in the title, we need to include two additional features: aggregation and path summarization.

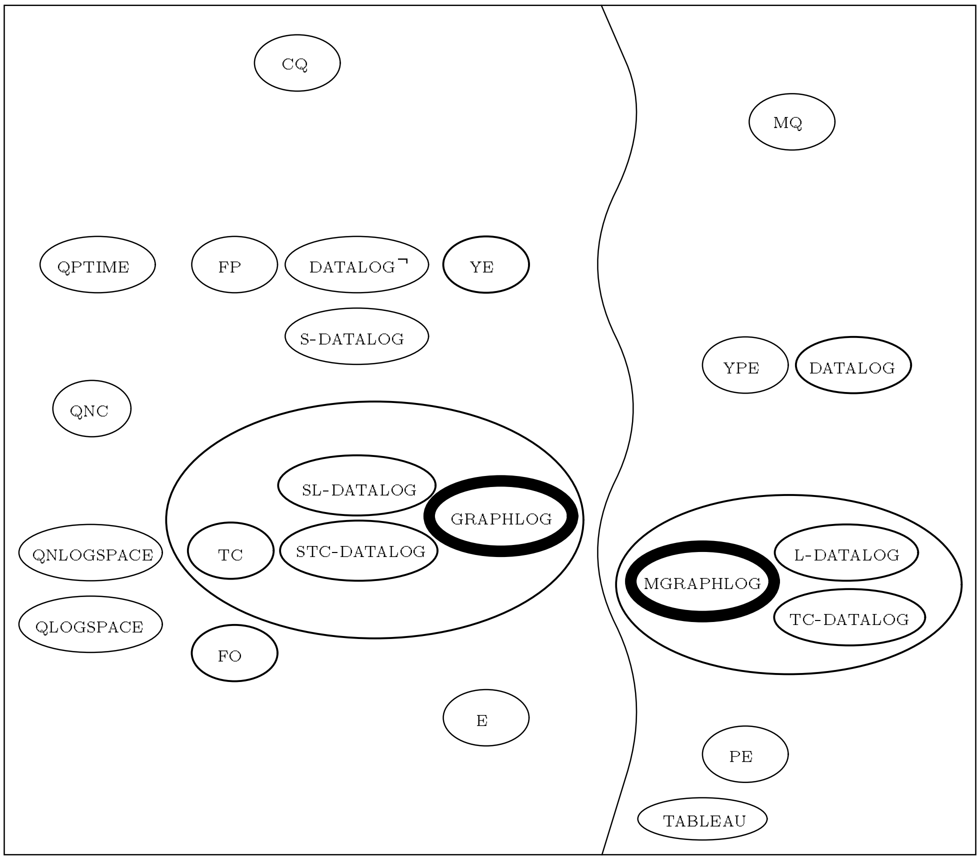

Figure 10: Relative expressive power of the query languages considered.

For introducing aggregation in GraphLog, we have defined an extension to Datalog that incorporates aggregate functions. We have proved that our extension captures the class of first order queries with aggregates of [Klu82]. We could have defined GraphLog with aggregates by translating it to a query language based on logic programming with sets, like LDL [TZ86]. LDL provides the facilities necessary for defining aggregation and does so within the logic programming framework. The problem is that logic programming with sets has (arbitrary) exponential data complexity (see [Bee88]). Our proposal instead retains polynomial time data complexity. However, the expressive power results of Section 3 do not apply when aggregate operators are considered.

There are several kinds of applications that require not only aggregation of sets of values appearing in edge labels, but also the capability to summarize information along paths. For example: "find the length of a shortest path between two nodes". Our approach integrates aggregation and path summarization uniformly. Since space does not allow a complete description, we illustrate with an example.

Example 4.1: Consider a task scheduling database storing data about which tasks affect which others, represented by the predicate affects(T1,T2), and about the duration and scheduled start of each task in the predicates duration(T,D) and scheduled-start(T,S). Durations and scheduled starts are both measured in days since some initial day 0.

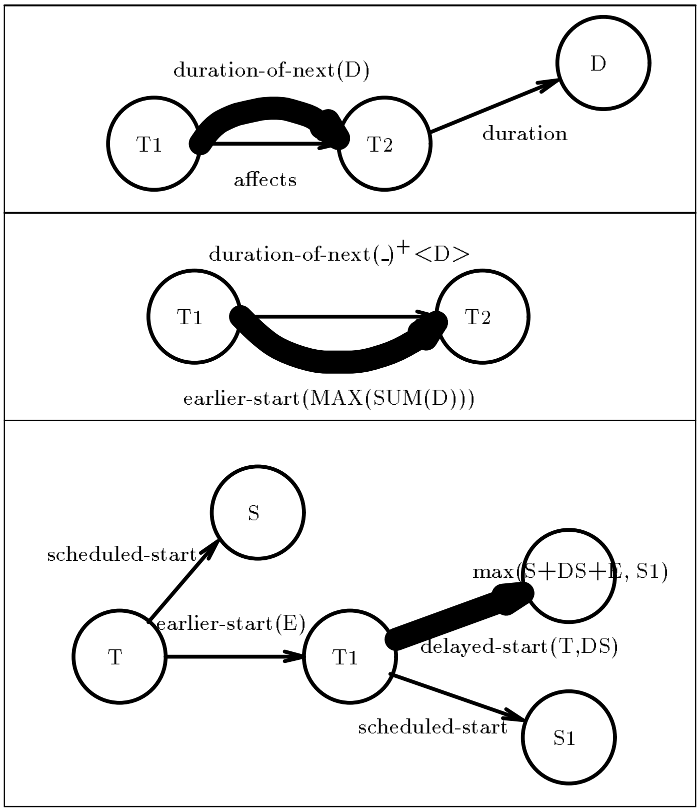

Figure 11: How a delay DS in task T would affect other tasks.

Figure 11 shows how to define in GraphLog a predicate delayed-start(T,DS) that answers the question: "how a would delay DS in task T affect other tasks?" The first query graph simply "moves" the duration of a task T2 to a new edge defined from any task T1 which affects T2 to T2. The second query graph defines a predicate earlier-start(E) from T1 to T2, where E is the longest sum of durations along all paths from T1 to T2. In the last query graph, the new start time from task T1 when task T is delayed by DS is defined by a simple calculation. □

5 Prototype Implementation

The original effort consisted in the specialization of a Smalltalk-80™ graph editor product (NodeGraph-80 [Ada87]) for editing query graphs and displaying database graphs. The resulting editor supports graph "cutting and pasting", as well as text editing of node and edge labels, node and edge repositioning and reshaping, scrolling over large graphs, storage and retrieval of graphs as text files, etc.

Once the graph editor was available, the query evaluation component was developed to support G^+ edge queries. These are simple queries containing two nodes with one edge connecting them; the edge may be labelled by an arbitrary regular expression. The algorithms used to search the database for answers are discussed in [MW89]. The user interface lets the user display the database

Figure 12: Displaying the answer of a GraphLog query.

graph (or part of it) in a window and the query graph in another. The answers may be displayed by highlighting qualifying paths directly on the database graph, or viewing them one by one in a separate window, or by turning their union into a new graph which can then itself the queried. The latter possibility supports iterative filtering of large and complex graphs.

The current prototype handles arbitrary GraphLog queries, not including aggregation and summarization. The screendump in Figure 12. shows three queries on a database of flights, where the nodes are cities and the edges are flights. There is one binary predicate for each airline; for example, the edge labelled AA from Buenos Aires to Lima means there is a flight between those two cities on Aerolíneas Argentinas. The three queries are in the three small windows at the top of the display.

The large window displays the result of the leftmost query: define a loop labelled RT-scale going from a city back to itself if the city is a scale on a sequence of Canadian Pacific flights from Rome to Tokyo. The result is being displayed by highlighting on the database window all instances of the new edge.

The G^+/GraphLog system graphs are held in main memory, as Smalltalk-80$^{\text{TM}}$ objects. However, the system has an interface for processing G^+/GraphLog queries on top of the Neptune hypertext front-end to the Hypertext Abstract Machine (HAM) [DS86]. The HAM is a general-purpose, transaction-based, multi-user server for a hypertext storage system. Using this interface, queries on large graphs may be posed.

6 Conclusions

We have described the GraphLog query language and characterized its expressive power. In doing so, we established the equivalence in expressive power of GraphLog, stratified linear Datalog, nondeterministic logarithmic space, and transitive closure. Our results imply that GraphLog is in QNC, hence amenable to efficient parallel implementations. Furthermore, implementations can benefit from the existing work on transitive closure computation and linear Datalog optimization (see [Ull89] for references).

An interesting research direction is the application of GraphLog to data models with complex objects and object identity. Our complexity results suggest that GraphLog may provide a good trade-off between computational complexity and expressive power. Current proposals (as in [Bee88]) either require exponential time or fail to express transitive closures. An exception is the polynomial time restricted language of [AK89]; however, this proposal uses inflationary semantics for negation, while we use the more natural and simpler stratified semantics.

References

[Ada87] Sam S. Adams. *NodeGraph-80 Version 1.0.* Knowledge Systems Corporation, 1987.

[AK89] Serge Abiteboul and Paris C. Kanellakis. Object identity as a query language primitive. Technical Report 1022, INRIA, April 1989.

[AU79] A.V. Aho and J.D. Ullman. Universality of data retrieval languages. *Proc. 6th ACM Symp. on Principles of Programming Languages*, pages 110–120, 1979.

[Bee88] Catriel Beeri. Data models and languages for databases. *Proc. 2nd Int. Conf. on Database Theory, Lecture Notes in Computer Science 326*, pages 19–40, 1988.

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

[CH85] A.K. Chandra and D. Harel. Horn clause queries and generalizations. *J. Logic Programming*, 2(1):1–15, 1985.

[CK86] Stavros S. Cosmadakis and Paris C. Kanellakis. Parallel evaluation of recursive rule queries. In *Proceedings of the Fifth ACM SIGACT-SIGMOD Symposium on Principles of Database Systems*, pages 280–293, 1986.

[CM89] Mariano Consens and Alberto Mendelzon. Expressing structural hypertext queries in GraphLog. In *Proceedings of the Second ACM Hypertext Conference*, pages 269–292, 1989.

[CMW88] I.F. Cruz, A.O. Mendelzon, and P.T. Wood. G^+: Recursive queries without recursion. In Larry Kerschberg, editor, *Proceedings of the Second International Conference on Expert Database Systems*, pages 355–368, 1988.

[Con89] Mariano P. Consens. Graphlog: "real life" recursive queries using graphs. Master's thesis, Department of Computer Science, University of Toronto, 1989.

[DS86] N. Delisle and M. Schwartz. Neptune: A hypertext system for cad applications. In *Proceedings of ACM-SIGMOD 1986 International Conference on Management of Data*, pages 132–142. SIGMOD, 1986.

[Imm87] Neil Immerman. Languages that capture complexity classes. *SIAM Journal on Computing*, 16(4):760–778, 1987.

[Imm88a] Neil Immerman. Descriptive and computational complexity. Technical report, Department of Computer Science, Yale University, New Haven, 1988.

[Imm88b] Neil Immerman. Nondeterministic space is closed under complementation. In *Third Structure in Complexity Theory Conference*, 1988.

[JAN87] H.V. Jagadish, R. Agrawal, and L. Ness. A study of transitive closure as a recursive mechanism. In *Proceedings of ACM-SIGMOD 1987 Annual Conference on Management of Data*, pages 331–344. SIGMOD, 1987.

[Kan87] P.C. Kanellakis. Logic programming and parallel complexity. In J. Minker, editor, *Foundations of Deductive Databases and Logic Programming*, pages 547–586. Morgan Kaufmann Publishers, Inc., 1987.

[Klu82] Anthony Klug. Equivalence of relational algebra and relational calculus query languages having aggregate functions. *Journal of the ACM*, 29(3):699–717, 1982.

[MW89] A.O. Mendelzon and P.T. Wood. Finding regular simple paths in graph databases. In *Proc. 15th International Conference on Very Large Data Bases*, pages 185–194, 1989.

[Nau87] J. Naughton. One-sided recursions. In *Proceedings of the Sixth ACM SIGACT-SIGMOD Symposium on Principles of Database Systems*, pages 340–348, 1987.

[Shm87] O. Shmueli. Decidability and expressiveness aspects of logic queries. In *Proceedings of the Sixth ACM SIGACT-SIGMOD Symposium on Principles of Database Systems*, pages 237–249. Assoc. for Comp. Machinery, 1987.

[TZ86] S. Tsur and C. Zaniolo. LDL: a logic-based data-language. In *Proceedings of the Twelfth International Conference on Very Large Data Bases*, 1986.

[Ull89] J.D. Ullman. *Principles of Database and Knowledge-Base Systems*, volume 2. Computer Science Press, Potomac, Md., 1989.

[UvG86] J.D. Ullman and A. van Gelder. Parallel complexity of logical query programs. *Proc. 27th Ann. Symp. on Foundations of Computer Science*, pages 438–454, 1986.