Low Complexity Aggregation in GraphLog and Datalog

Mariano P. Consens
consens@db.toronto.edu

Alberto O. Mendelzon
mendel@db.toronto.edu

Computer Systems Research Institute
University of Toronto
Toronto, Canada M5S 1A4

Abstract

We present constructs for computing aggregate functions over sets of tuples and along paths in a database graph.

We show how Datalog can be extended to compute a large class of queries with aggregates without incurring the large expense of a language with general set manipulation capabilities. In particular, we aim for queries that can be executed efficiently in parallel, using the class NC and its various subclasses as formal models of low parallel complexity.

Our approach retains the standard relational notion of relations as sets of tuples, not requiring the introduction of multisets. In the case where no rules are recursive, the language is exactly as expressive as Klug's first order language with aggregates. We show that this class of non-recursive programs cannot express transitive closure (unless LOGSPACE=NLOGSPACE), thus providing evidence for a widely believed but never proven folk result. We also study the expressive

This work has been supported by the Information Technology Research Centre of Ontario and the Natural Science and Engineering Research Council of Canada. The first author was also supported by the PEDECIBA – United Nations Program for the Development of Basic Sciences, Uruguay.

A preliminary version of this paper was published in the Proceedings of the Third International Conference on Database Theory, Lecture Notes in Computer Science Nr. 470, Springer-Verlag, pages 379-394, 1990.

power and complexity of languages that support aggregation over recursion.

We then describe how these constructs, as well as manipulating the length of paths in database graphs, are incorporated into our visual query language GraphLog. While GraphLog could easily be extended to handle all the queries described above, we prefer to restrict the language in a natural way to avoid explicit recursion; all recursion is expressed as transitive closure. We show that this guarantees all expressible queries are in NC. We analyze other proposals and show that they can express queries that are logspace-complete for P and thus unlikely to be parallelizable efficiently.

1 Introduction

First order query languages, like the relational algebra, are never used as practical query tools; instead, they are augmented in practice by various features, as embodied in commercial query languages like *SQL*. One such feature that seems fundamental is the ability to collect sets of tuples and compute *aggregate functions* (sum, average, minimum, maximum, etc.) over them. Until very recently database theory had not paid much attention to this capability, perhaps because it does not seem to affect the expressive power and complexity of queries in an essential way. One exception is the work of Klug [Klu82], who formalized extensions of relational algebra and calculus with aggregates and showed their equivalence. Ozsoyoglu *et al.* [OOM87] extended aggregate functions to relations with set-valued attributes. Both Cooper [Coo80] and Sheng [She90] have been concerned with the relative power of different aggregate operators added to the same language.

More recently, Mumick *et al.* [MPR90], Ganguly *et al.* [GGZ91, GZG92], Lefebvre [Lef91], and Sudarshan and Ramakrishnan [SR91] discuss extensions of Datalog with aggregates and the associated optimization problem. Similarly, Beeri *et al.* [BRSS92], Kemp and Stuckey [KS91], Ross and Sagiv [RS92], and Van Gelder [VG92] propose alternate semantics for aggregates in logic programs and deductive databases. Some authors have considered aggregation on top of general set manipulation in the context of logic languages or complex object models [AB88, TZ86]. However, we are interested in being able to express aggregation without incurring the large expense of a language with general set manipulation capabilities.

In this paper we show that, even for first order queries, the presence of aggregate operators affects the complexity and expressive power of query languages. In particular, we show by complexity theoretic arguments that adding aggregate operators to a first order language does not make it possible to express transitive closure, unless LOGSPACE = NLOGSPACE. This result, while not difficult, closes a gaping hole in the application of database theory to practice. It is considered folklore that SQL cannot express transitive closure, but this has never been proven before.

However, our main interest is in what happens when we incorporate aggregation to languages that go beyond the power of first order queries, such as *Datalog*. In this paper we make several theoretical contributions to this issue. Motivated by the practical goal of providing a convenient visual notation for expressing complex aggregative queries, we present the aggregation and summarization facilities of *GraphLog*, a visual query language based on graph pattern matching [Con89, CM90b]. From a theoretical point of view, we analyze the complexity and expressive power of GraphLog with aggregates and compare them to recent proposals for incorporating aggregates into Datalog. We use parallel complexity theory [Kan88] to distinguish various classes of queries according to whether they are efficiently parallelizable. In particular, we use the class NC of problems computable in polylogarithmic time by a polynomial number of processors, and its subclasses [Coo85], as formal models of efficiently parallelizable problems.

GraphLog is 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. From a broader perspective, 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 the objects to nodes and the relationships to edges. In the case of semantic and object-oriented databases, there is a natural mapping of objects to nodes and attributes to edges.

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. An implementation of such an interface is presented in [CM90a].

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 [CM90b], we gave such a characterization by proving that GraphLog without aggregates, stratified linear Datalog, and first-order logic augmented with a transitive closure operator all have the same expressive power. It follows (using a result by Immerman) that, in the presence of an order relation on the domain, GraphLog queries express exactly the queries computable in non-deterministic logarithmic space. In this paper we present results on the expressive power of GraphLog with aggregates.

The organization of the paper is as follows. In Section 2 we add aggregation to Datalog. Our first extension only allows aggregating over non-recursive predicates; semantic difficulties arise when trying to do aggregation over recursion. We then discuss two ways of getting around these difficulties. The first one is to restrict programs to be *monotone* with respect to aggregation, following [MPR90]. The second is to consider only rules that implement transitive-closure-like computations over closed semirings [CN89, Ull89].

In Section 3 we present the GraphLog query language. We then discuss how aggregates are incorporated into GraphLog, first non-recursively, as above, and then recursively by adding the notion of *path summarization*. Aggregation and path summarization in GraphLog allows us both to express closed semiring computations and to manipulate the length of paths in the database graph. By incorporating these capabilities into the language, we provide a visual notation for expressing and solving graph traversal problems (see [MS90] for a survey on query language support for graph traversal operations). We conclude in Section 4.

2 Aggregation in Datalog

In this paper, S-DATALOG is the language of function-free Horn clauses extended with stratified negation. We follow standard terminology (see e.g. [Ull88]) for its syntax and semantics. We identify several syntactic restrictions of S-DATALOG. SNR-DATALOG is the restriction to the case where all rules are non-recursive. STC-DATALOG, defined formally in [CM90b], is the language in which all recursive rules are transitive closure rules. Finally, SL-DATALOG is the language in which all recursive rules are *linear*, that is, they contain exactly one predicate mutually recursive with the head.

2.1 Non-recursive aggregation

In this subsection we add aggregate operators to Datalog and give semantics to this extension using naive evaluation by stages together with the application of Klug's aggregate operation, following [Con89]. Our first extension is a rather conservative one, in which we keep recursion completely separate from aggregation. In the next subsections we discuss what happens when we try to allow aggregation to interact with recursion.

We start with the set of *aggregate functions* $\mathrm{AGG} = \{\mathrm{MIN}, \mathrm{MAX}, \mathrm{EVEN}, \mathrm{COUNT}, \mathrm{SUM}\}$. The aggregate function EVEN returns 0 if the count of values aggregated is even and 1 otherwise, while the remaining aggregate functions have the expected meaning. Additional aggregate functions (e.g., AVG, VAR, STDV) can be defined in terms of the ones in AGG if the query language supports arithmetic operations.

Definition 2.1:[Klu82] The *aggregate operation* $\gamma_{f_A(X)}(r)$ is the relation

$$\gamma_{f_A(X)}(r) \quad = \quad \{t : t \text{ is a } XA\text{-tuple}, t[X] \in \pi_X(r), t[A] = f_A(\sigma_{X=t[X]}(r))\}$$

over XA, whenever XA is a subset of the scheme of r, $f \in \mathrm{AGG}$, and $f_A(r)$ yields the f aggregated value of all values $t[A], t \in r$, including repetitions.

The *relational algebra with aggregates* is the algebra obtained by extending the relational algebra with the aggregate operation. □

We consider databases over the domains D (an uninterpreted domain) and N (the natural numbers). Variables ranging over D and N belong to the uninterpreted and interpreted sorts, respectively. We assume that the values from N present in the active domain must have a logarithmic representation in the size of the database description. This is a natural restriction. We want to avoid integers that are too large compared with the database, but at the same time we want them to be large enough to hold a count (or the sum) of the elements present in the database.

Definition 2.2: A *term* is one of a constant, a variable, or an aggregate function $f \in \mathrm{AGG}$ applied to a variable X of the interpreted sort, the latter denoted $f(X)$ and called an *aggregate term.*

An *aggregate atom* is an atomic formula of the form $p(t_1, \ldots, t_n)[\overline{X}]$, where $t_1, \ldots, t_n$ are terms and at least one $t_i, 1 \leq i \leq n$, is an aggregate term.

An *aggregate rule* is a rule with an aggregate atom in its head. $\square$

Definition 2.3: A *stratified Datalog program with non-recursive aggregation* is a set of general rules and aggregate rules stratified w.r.t negation and such that the predicates in the head of aggregate rules (called *aggregate predicates*) are non-recursive[1].

The semantics of stratified Datalog programs with non-recursive aggregation are given by the application of Algorithm 2.1 (given below), which finds the perfect model of the program[2].

The set of queries expressed by stratified Datalog programs with non-recursive aggregation is denoted S-DATALOG$^{\mathrm{AGG}}$. $\square$

Before presenting the evaluation procedure that assigns meaning to stratified Datalog programs with non-recursive aggregation, let consider an example.

Example 2.1: Consider a database with relations payroll(E,S) and works-for(E,D) representing that employee E earns salary S and employee E works for department D, respectively.

If the (one rule) Datalog program with aggregation

$$\text{personnel-budget(D,SUM(S))} \leftarrow \text{works-for(E,D),payroll(E,S).}$$

is given as input to Algorithm 2.1, the output will be the relation personnel-budget(D,T) with the total amount T paid to employees working for department D. $\square$

[1] Requiring aggregate predicates to be non-recursive is completely equivalent to "stratifying" w.r.t. aggregation. The simplicity of the condition is due to the fact that aggregation occurs in the heads of rules, instead of in the bodies (as it is the case in [MPR90]).

[2] We omit presenting the model-based version of the semantics of stratified Datalog programs with non-recursive aggregation (see [MPR90, KS91, RS92, VG92]), since our results are based on the equivalent operational semantics given by Algorithm 2.1.

Algorithm 2.1: Evaluation of stratified Datalog with non-recursive aggregation.

INPUT: A stratified Datalog program with aggregation $\mathcal{P}_p$ and a finite database $d = (r_1, \ldots, r_m)$ over D and N.

OUTPUT: A finite relation r_0 over D and N; the result of the query expressed by $\mathcal{P}_p$ evaluated on d.

METHOD: The algorithm proceeds as in the naive evaluation by stages (see [Ull88]). First, initialize the relations corresponding to the EDB predicates of $\mathcal{P}_p$ with d. Then, for each strongly connected component of the dependence graph of $\mathcal{P}_p$, proceeding from the lowest to the highest strata, repeatedly evaluate (until no more changes occur) the relation corresponding to each predicate p_i, by unioning the relations defined by all the rules with head p_i. The only new consideration is how the relation defined by an aggregate rule is computed.

Given an aggregate rule

$$r' : p'(\ldots, f^1(Y_1), \ldots, f^n(Y_n), \ldots) \leftarrow s_1, \ldots, s_k.$$

where $f^1(Y_1), \ldots, f^n(Y_n)$ are the aggregate terms occurring in the head of r', the relation defined by the rule r' is given by the evaluation of the (relational algebra with aggregates) expression

$$\gamma_{f^1_{A_1}(X)}(r) \bowtie \ldots \bowtie \gamma_{f^n_{A_n}(X)}(r)$$

where r is the relation over all attributes corresponding to the variables in the body as in the naive evaluation by stages, X is the set of attributes corresponding to the variable occurrences in the head of p' not appearing in aggregate terms, and $A_1, \ldots, A_n$ are the attributes corresponding to the variable occurrences $Y_1, \ldots, Y_n$.

Once all the IDB predicates are evaluated by the above procedure, the relation r_0 is given by the value of the relation corresponding to the carrier p. $\square$

The queries expressed by stratified non-recursive (resp. transitive closure, linear) Datalog with non-recursive aggregation are denoted SNR-DATALOG$^{\mathrm{AGG}}$

(resp. STC-DATALOG$^{\text{AGG}}$, SL-DATALOG$^{\text{AGG}}$). The results below characterize the expressive power of stratified Datalog with non-recursive aggregation.

We first show that the expressive power of SNR-DATALOG$^{\text{AGG}}$ equals the expressive power of first order queries with aggregates, denoted FO$^{\text{AGG}}$, as defined in [Klu82] by an equivalent algebra and calculus with aggregates.

Lemma 2.1: FO$^{\text{AGG}}$ = SNR-DATALOG$^{\text{AGG}}$

Proof: The proof consists in showing that any algebra with aggregates query is a SNR-DATALOG$^{\text{AGG}}$ query and vice versa. Since stratified non-recursive Datalog expresses exactly the first order queries (see [Ull88]), and both languages are closed under composition, we only need to show that the expression of the algebra with aggregates $\gamma_{f_A(X)}(r)$ has an equivalent stratified Datalog program with aggregation, and that any aggregate rule has an equivalent expression of the algebra with aggregates; the desired result follows by structural induction on both languages.

Any aggregate rule has an equivalent expression of the algebra with aggregates resulting from applying Algorithm 2.1. Reciprocally, it is easy to check that Algorithm 2.1 has $\gamma_{f_A(X)}(r)$ as the output (for predicate p) corresponding to the program

$$p(X_1, \ldots, X_k, f(Y)) \leftarrow q.$$

where q is the EDB predicate corresponding to r, the variables $X_1, \ldots, X_k$ correspond to the set of attributes X, and the variable Y corresponds to the attribute A. □

To separate the class of queries FO$^{\text{AGG}}$ from STC-DATALOG$^{\text{AGG}}$, we use complexity theoretic arguments. Let QC denote the set of queries in complexity class C (i.e., those for which the decision problem of membership of a tuple in the answer of a fixed query is in class C, where the input size is obtained from a standard encoding of the input database and output tuple). The complexity classes that we consider are described by Cook [Coo85].

To present the complexity results in a more general framework we consider the sets AGG_{C} consisting of the aggregate functions such that the decision problem associated with the evaluation of the corresponding aggregate operations is in complexity class C. We have that the set AGG of aggregate functions that we listed in Section 2.1 is a proper subset of $\text{AGG}_{\text{LOGSPACE}}$ (abbreviated AGG_{L}, and similarly, $\text{AGG}_{\text{NLOGSPACE}}$ is shortened to AGG_{NL}).

The first lemma is a straightforward extension of a well-known result (see [Imm87]).

Lemma 2.2: $\text{FO}^{\text{AGG}_\text{L}} \subseteq \text{QLOGSPACE}$

Our next result is proved by slight modification of the proofs given in [Con89] that shows that STC-DATALOG and SL-DATALOG have precisely the same expressive power, and that their data complexity is non-deterministic logarithmic space (proved using a result in [Imm87]).

Lemma 2.3: $\text{STC-DATALOG}^{\text{AGG}_\text{NL}} = \text{SL-DATALOG}^{\text{AGG}_\text{NL}} \subseteq \text{QNLOGSPACE}$

The next lemma follows from the previous lemmas and the fact that transitive closure is logspace-complete for NLOGSPACE.

Lemma 2.4: $\text{FO}^{\text{AGG}_\text{L}} \subset \text{STC-DATALOG}^{\text{AGG}_\text{L}}$
(unless LOGSPACE = NLOGSPACE)

By the same argument we can conclude that $\text{FO}^{\text{AGG}_\text{L}}$ does not express transitive closure unless LOGSPACE = NLOGSPACE.

It is important to recognize that not all aggregate functions have the same impact on the expressive power and data complexity of query languages with aggregates. It is immediate that a first order language with an order relation on the domain (denoted $\text{FO}^<$) can express all first order aggregate queries when we restrict the aggregate functions to be MAX and MIN, that is $\text{FO}^< = \text{FO}^{\{\text{MAX},\text{MIN}\}}$. Some aggregate functions can be expressed in terms of others, but not vice versa: $\text{FO}^{\text{COUNT}} \subset \text{FO}^{\text{SUM}}$ (from [Coo80]), $\text{S-DATALOG}^{\text{EVEN}} \subset \text{S-DATALOG}^{\text{COUNT}}$, and $\text{S-DATALOG}^{\text{COUNT}} = \text{S-DATALOG}^{\text{SUM}}$ (both from [She90]).

A somewhat surprising result follows from Immerman's corollary in [Imm87] that *parity* of the input is not expressible in $\text{FO}^<$ (also extended with a predefined relation).

Lemma 2.5: $\text{FO}^< \subset \text{FO}^{\text{EVEN}}$

The following result, by Immerman, characterizes $\text{FO}^<$ in terms of the complexity class AC^0. This class characterizes those decision problems solvable by Alternating Turing Machines requiring logarithmic space and constant time (or equivalently, problems recognizable by a uniform sequence of polynomial size, unbounded fan-in boolean circuits of constant depth).

Lemma 2.6:[Imm88a] $\text{FO}^{<} = \text{QAC}^0$

Therefore, we can see that some aggregate functions (EVEN in particular) produce an increase in the complexity of aggregate queries, not just their expressive power. This increase is limited as the next lemma shows. The bound follows from membership of the sum of n integers of n bits each in NC^1 (see [Coo85]). The complexity class NC^1 denotes the class of problems solvable by a uniform circuit family of depth logarithmic in the number of input bits.

Lemma 2.7: $\text{FO}^{\text{SUM}} \subseteq \text{QNC}^1$

2.2 Monotone Aggregation

Would it not be the case that giving as input to Algorithm 2.1 aggregate rules with recursive predicates in the head the resulting language will have increased expressive power? The program in Example 2.2 will convince the reader that such an extension would lead to the possibility of writing non-terminating programs.

Example 2.2: It is not difficult to realize that giving the program below (which is not a valid $\text{S-DATALOG}^{\text{AGG}}$ program) as input to Algorithm 2.1 will start a non-terminating procedure.

$$\begin{array}{l} \mathsf{p(X)} \leftarrow \mathsf{q(X)}. \\ \mathsf{p(SUM(X))} \leftarrow \mathsf{p(X)}. \end{array}$$

□

However, combining recursion with aggregation does not always yield an infinite result. Mumick, Pirahesh and Ramakrishnan [MPR90], give a semantic condition and an equivalent syntactic restriction on the occurrences of aggregation across recursion which warrant the existence of a perfect model. For simplicity, and without loss of generality, assume that each aggregate predicate appears only in the head of one aggregate rule. The semantic condition requires all non-aggregate rules r not to invalidate earlier tuples added to the head of r by Algorithm 2.1 when tuples are added to a non-aggregate predicate in the body of r, or when tuples are added to the relation defined by the body of the aggregate rule defining an aggregate predicate in the body of r. The programs satisfying the above restrictions are called *monotonic programs*. We can place similar restrictions in our Datalog programs with aggregates and define a superset of S-DATALOG$^{\text{AGG}}$ denoted by M-DATALOG$^{\text{AGG}}$. The semantics of M-DATALOG$^{\text{AGG}}$ programs is still given by Algorithm 2.1. The corresponding extension to SL-DATALOG$^{\text{AGG}}$ is denoted by ML-DATALOG$^{\text{AGG}}$.

Below we give an example from [MPR90] of a query expressible by a monotone program.

Example 2.3: Consider a relation owns(C1,C2,S) representing that company C1 directly owns S% of the stock of company C2. The solution to the *corporate takeover* query is a relation bought(C1,C2) representing that company C1 has bought company C2, which is true whenever C1 controls more than 50% of the stock of C2. A company C1 controls itself and also controls stock controlled by any other company C1 has bought.

The ML-DATALOG$^{\text{AGG}}$ program that expresses the corporate takeover query is given below.

bought(C1,C1) $\leftarrow$ owns(C1,C2,S).
bought(C1,C2) $\leftarrow$ control(C1,C2,S), S $>$ 50.
control(C1,C2,SUM(S)) $\leftarrow$ bought(C1,C3), owns(C3,C2,S).

□

Mumick *et al.* do not discuss the complexity of this query. The next lemma characterizes it.

Lemma 2.8: Corporate takeover is logspace-complete for PTIME.

Proof: By a reduction from *path systems accessibility* [GJ79]. An instance of the path system accessibility problem consists of a finite set N of nodes, a ternary relation R on N, and two sets S, T of source and terminal nodes. A node x is accessible if it is in S or there exist accessible nodes y, z such that the triple (x, y, z) is in R. The path system accessibility problem is defined by the question: is there an accessible terminal node?

Given an instance of the problem above, an instance of corporate takeover can be constructed (in logarithmic space), as follows. Create one company for each node in N and one company for each triple in R. For each triple

$t = (x, y, z)$ in R, company t owns ϵ% of company x (where $\epsilon = 50/n$, and n is the number of triples with first component x), company y owns 50% of company t, and company z owns 50% of company t. Note that counting the triples required to determine ϵ can be done in logarithmic space, and that ϵ can be encoded with a logarithmic number of bits. The construction is completed by adding a company s_0 that owns 100% of every company in S and 50% of every company in $N - S$. Then, there is an accessible terminal node iff one of the companies in T is controlled by s_0. $\square$

It follows from this result that SL-DATALOG$^{\mathrm{AGG}}$NL cannot express corporate takeover unless NLOGSPACE = PTIME. Therefore, we conclude

Lemma 2.9: SL-DATALOG$^{\mathrm{AGG}}$NL $\subset$ ML-DATALOG$^{\mathrm{AGG}}$NL
(unless NLOGSPACE = PTIME)

It is worth pointing out that the formulation of corporate takeover used by Mumick *et al.* requires the use of multisets, while ours does not. Multisets are supported by commercial query languages, like SQL. However, the inclusion of multisets in recursive query languages has a considerable impact on the cost of evaluating queries. For example, the multiset transitive closure (as defined in [MPR90]) of an arbitrary graph is infinite, and the same query for acyclic graphs may yield an exponential number of answers.

There are other differences between our approach and the one presented by Mumick *et al.*, although when multisets are not considered, their stratified and monotone classes of programs are equivalent to our S-DATALOG$^{\mathrm{AGG}}$ and M-DATALOG$^{\mathrm{AGG}}$ classes, respectively. At the syntactic level we use aggregate rules, while they have *group-by* subgoals. We give semantics to our stratified Datalog programs with aggregates in a simple way by using an extension of the naive algorithm, but this in no way precludes the use of an extended semi-naive algorithm for the evaluation of monotone programs.

2.3 Closed Semirings

Another class of programs which involve recursion through aggregation, and yet Algorithm 2.1 finds a perfect model for them, is illustrated by the following two examples.

Example 2.4: The relation shortest-path(X,Y,SL) defined by the following Datalog program with aggregation finds the length SL of the *shortest path* between pairs of cities X and Y, where the direct distances L between cities X0 and Y0 are given by the relation distance(X0,Y0,L). The auxiliary predicate lengths(X,Z,Y,U) means there exists a path of length U=SL+L from X to Y through Z such that the length of the shortest path from X to Z is SL, and the direct distance between Z and Y is L.

```
lengths(X,X,X,0) ← distance(X,Y,L).
lengths(X,X,X,0) ← distance(Y,X,L).
lengths(X,Z,Y,U) ← shortest-path(X,Z,SL),distance(Z,Y,L), U=SL+L.
shortest-path(X,Y,MIN(L)) ← lengths(X,Z,Y,L).
```

□

Example 2.5: The solution to the *parts explosion* query consists of finding a relation total-quantity(P,S,T), where T is the total number of parts S that takes to make one unit of part P, given a relation subpart(P0,S0,N) listing the number N of subparts S0 of each part P0. The Datalog program with aggregation given below expresses the parts explosion query. The auxiliary predicate contains(P,Q,S,N) means that part P's usage of part Q contributes N units to P's total usage of part S, where S is a direct subpart of Q. To guarantee termination in case of a cyclic graph, we check for cycles by using the IDB predicate tc-subpart, the positive transitive closure of the EDB predicate subpart.

```
contains(P,P,P,1) ← subpart(P,S,N).
contains(P,P,P,1) ← subpart(S,P,N).
contains(P,Q,S,U) ← total-quantity(P,Q,T),subpart(Q,S,N), U=T×N.
contains(P,Q,S,+∞) ← total-quantity(P,Q,T), tc-subpart(Q,Q),subpart(Q,S,N).
total-quantity(P,S,SUM(N)) ← contains(P,Q,S,N).
```

□

The above examples are particular cases of an algebraic structure with two operators $\oplus$ and $\otimes$ known as a *closed semiring* (see [Ull89]). Shortest path corresponds to $(\oplus, \otimes) = (min, +)$ over the domain of the positive real numbers and positive infinity, while parts explosion to $(\oplus, \otimes) = (+, \times)$ over the domain of the natural numbers and positive infinity. Note that both termination and correctness of the programs above depend on the algebraic properties of closed semirings.

As suggested by these examples, we extend the class of stratified Datalog programs with non-recursive aggregation to capture closed semiring queries when the operators $\oplus$ and $\otimes$ are taken from $\{max, min, +, \times\}$. We do this by adding a new kind of rule called *closed semiring rule*. We restrict the extended programs so that predicates in the head of closed semiring rules are non-recursive. Programs in the extended language are assigned meaning by translating them into Datalog programs with recursive aggregation that are executable by Algorithm 2.1. The resulting class of queries is denoted S-DATALOG$^{\mathrm{CSR}}$ (with the corresponding extension of SL-DATALOG$^{\mathrm{AGG}}$ denoted by SL-DATALOG$^{\mathrm{CSR}}$).

Example 2.6: The SL-DATALOG$^{\mathrm{CSR}}$ one rule program

$$\mathsf{total\text{-}quantity(P,S,SUM(PROD(N)))[P;S]} \leftarrow \mathsf{subpart(P,S,N)}.$$

whose semantics is given by the evaluation by Algorithm 2.1 of the program of Example 2.5, computes the parts explosion query. Notation [P;S] indicates which pair (of lists, in general) of variables corresponding to attributes of the relation r defined by the body of the rule is taken to define the endpoints of the edges of the graph given by r. $\square$

There is a fundamental advantage in syntactically recognizing closed semiring computations: instead of evaluating the corresponding stratified Datalog program with recursive aggregation using Algorithm 2.1, we can make use of algorithms for the efficient evaluation of closed semirings. Several database algorithms for the implementation of closed semiring problems have been studied in [ADJ88, CN89, IR88, HAC90].

Once more we have a situation in which not all aggregate functions, in this case not all closed semiring operators, have the same impact on the data complexity of stratified Datalog extended with closed semirings.

If we restrict the $\oplus$ operator to be max or min and the $\otimes$ operator to be one of $\{max, min, +\}$, and use SL-DATALOG$^{(\{max,min\},\{max,min,+\})}$ to denote the resulting set of queries, we have

Lemma 2.10: SL-DATALOG$^{(\{max,min\},\{max,min,+\})}$ $\subseteq$ QNLOGSPACE

Proof: To see how we can answer whether a tuple belongs to an IDB predicate defined by a SL-DATALOG$^{(\{max,min\},\{max,min,+\})}$ program with a non-deterministic Turing Machine using logarithmic space, we first translate the program into an equivalent program whose recursive predicates are defined by transitive closure rules.

Once this is done, the test can be performed in logarithmic space as long as neither transitive closure nor adding (nor finding the maximum or minimum value) along paths is involved. When closures or computations along paths are required, we employ an non-deterministic Turing Machine that uses logarithmic space to store two nodes of the underlying graph and the summarized (with max, min, or $+$) value along a path of the graph. The machine proceeds by non-deterministically "guessing" each step along the path together with the summarized value. Then, it checks that the step corresponds to an edge in the graph and that the summarized value is the result of summarizing the value calculated at the previous "guess" with the value in the corresponding edge in the graph. Encoding the summarized values along paths in the database with a number of bits logarithmic in the size of the database description is possible in the case of sum, because of an earlier requirement that guarantees that the values of the attributes added along paths have a logarithmic representation in the size of the database description.

Finally, to compute the $\oplus$ operators max or min is enough to use order and negation of existential quantification (which can be done in NLOGSPACE due to the closure under complementation of nondeterministic space [Imm88b]); a value computed along a path is the maximum (resp. minimum) value for all paths if there is no path with a larger (resp. smaller) value. □

However, when we consider the $\oplus$ operator to be $+$ or $\times$, or the $\otimes$ operator to be $\times$, we get an increase in data complexity. We do not expect the parts explosion query to be in NLOGSPACE. The non-deterministic logarithmic space algorithm given above cannot encode the "guessed" value for the product of values along paths in logarithmic space, even when the values along the path are constants (e.g., the product of a path of length n labelled by 2 is 2^n, which requires n bits to be represented). In general, we have

Lemma 2.11: SL-DATALOG$^{\mathrm{CSR}} \subseteq$ QNC^2

Proof: The containment of this class of queries in NC^2 is due to the fact that closed semiring problems reduce to matrix powering, which is in NC^2 (see [Coo85]). Matrix powering consists of finding the n^{th} power of a $n \times n$ matrix of n-bit integers. Matrix powering is NC^1-complete for DET, the algebraic complexity class of those problems that are NC^1-reducible to the problem of finding the integer determinant of a $n \times n$ matrix of n-bit integers. □

3 Aggregates in GraphLog

Many databases can be naturally viewed as graphs. Object-oriented databases are especially suited to this representation, and so are Hypertext databases [CM89]. A relational database can be represented by a directed multigraph having an edge labeled $r(c_1, \ldots, c_k)$ from a node labeled $(a_1, \ldots, a_i)$ to a node labeled $(b_1, \ldots, b_j)$ corresponding to each tuple $(a_1, \ldots, a_i, b_1, \ldots, b_j, c_1, \ldots, c_k)$ of each relation r in the database. We intend to profit from this graph based representation of a database to express queries by describing the graph patterns to be retrieved.

Definition 3.1: A *query graph* is a directed multigraph with no isolated nodes having the following properties. The nodes are labeled by sequences of variables and constants. Each edge is labelled by a literal (either an atom or a negated atom) or by a *closure literal*, which is simply a literal s followed by the positive closure operator, denoted s^+. 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.

A *graphical query* is a finite set of query graphs. □

The semantics of a graphical query are given by a translation (defined in [CM90b]) to STC-DATALOG. Each query graph G in a graphical query corresponds to a rule r with the label of the distinguished edge in the head, and as many literals in the body as there are non-distinguished edges in G. An edge of the query graph labeled with a closure literal s^+ introduces an IDB predicate defined by additional rules expressing the transitive closure of the predicate in s. The body of r contains the IDB predicates introduced by the closure literals and the remaining edge labels of G.

We allow as expressions of the GraphLog query language only those graphical queries whose distinguished edges define non-recursive predicates. Note that, although we disallow explicit recursion, recursion is nevertheless implicit in the use of closure literals. We denote by GRAPHLOG the set of queries that are expressible in the language.

The language can be made considerably more concise by generalizing literals and closure literals to arbitrary regular expressions. Each operator introduced is definable in terms of the basic language and is added only for convenience. In addition to the usual operators for positive and Kleene closure, alternation, and concatenation, two new ones are defined: inversion reverses the direction of the edge labeled by the regular expression, and negation negates the predicate defined by its argument.

3.1 Non-recursive Aggregation in GraphLog

In this subsection we define $\text{GRAPHLOG}^{\text{AGG}}$, an extension of GraphLog that captures the Datalog aggregate rules defined in Section 2.

Let us call the nodes that are the end-points of the distinguished edge *distinguished nodes*. The presence of aggregate terms is allowed in the label of the distinguished edge or a distinguished node of GraphLog's query graphs. A query graph with the distinguished edge or a distinguished node labeled by an aggregate atom is translated to the corresponding aggregate rule. The queries expressed by GraphLog extended with aggregates are denoted by $\text{GRAPHLOG}^{\text{AGG}}$.

Example 3.1 shows how a recursive query involving negation and non-recursive aggregation is expressed in $\text{GRAPHLOG}^{\text{AGG}}$.

Example 3.1: The graphical query in Figure 1 defines a relation disk-util-excl-disk1(D,T) giving the total space T in disk (excluding disk1) used by directory D and all its (recursively) contained files and sub-directories in a hierarchical file system. The predicate disk-util-excl-disk1 labels the distinguished edge, which is graphically represented by a bold edge. Relation contains(D,F) says that directory D (directly) contains file (or directory, which is a particular type of file) F. The closure literal contains^{+} represents the transitive closure of relation contains and appears labeling a dashed edge, a graphical notation that suggests that the edge in the query graph matches paths in the database graph. The crossed over edge represents the literal

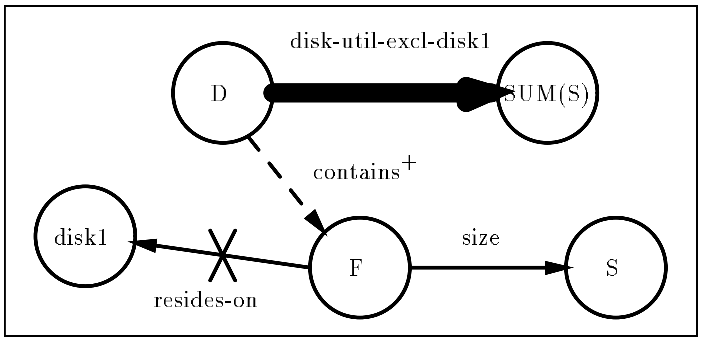

Figure 1: Total disk utilization (excluding disk1) for each directory.

¬resides-on(F,disk1) (note that crossing over the edge is graphical notation replacing negation in the label), where relation resides-on(F,D) gives the disk D on which the file F resides, and disk1 is a constant. Finally, the size S of file F is given by size(F,S).

The GRAPHLOG$^{\mathrm{AGG}}$ query in Figure 1 is given semantics by a translation to S-DATALOG$^{\mathrm{AGG}}$, that yields

tc-contains(D,F) ← contains(D,F).
tc-contains(D,F) ← tc-contains(D,E), contains(E,F).
disk-util-excl-disk1(D,SUM(S)) ← tc-contains(D,F), size(F,S),
¬resides-on(F,disk1).

□

We can easily extend the construction given in [Con89, CM90b] (that shows that STC-DATALOG, SL-DATALOG, and GRAPHLOG all have precisely the same expressive power) to consider aggregates, and prove that

Lemma 3.1: GRAPHLOG$^{\mathrm{AGG}}$ = STC-DATALOG$^{\mathrm{AGG}}$ = SL-DATALOG$^{\mathrm{AGG}}$

3.2 Path Summarization

As we showed in Section 2.3, there are several examples of applications that require not only aggregation of the values appearing in edge labels, but also the capability to summarize information along paths. The archetypal example is the shortest path query. The motivation for introducing path summarization in a recursive query language is similar to that in the PROBE project [DS86, RHDM86], the G^+ query language [Cru87, CMW88], and to relational algebra extensions [Agr87, CN89].

There are two important points that must be taken into consideration when introducing path summarization in the language: what is considered a path along which to summarize, and whether summarization is combined with aggregation. Paths may have arbitrary lengths in cyclic graphs. In that situation, the sum of the values along a path is undefined. Suppose we restrict the summarization to simple paths. If we do not combine summarization with an aggregate operator, then the number of values from all the paths can be exponential in the size of the database. Even when we combine path summarization along simple paths with aggregation, the problem is still intractable. Finding the length of the longest simple path in a directed graph labeled with positive integers is an NP-complete problem [GJ79], although when the graph is acyclic it can be done in polynomial time[3]. Finding the length of the longest simple path is expressible in both the G^+ query language and the α-extension to relational algebra of Agrawal [Agr87]. The latter can also find all the sums of values along paths, and as we said before there may be an exponential number of those values.

In light of the above discussion we restrict the use of path summarization in GraphLog to those cases that admit efficient implementations. As we seen in the previous section, the complexity of path summarization combined with aggregation when the operators constitute a closed semiring is within NC. Consequently, we define the semantics of path summarization in GraphLog by a translation to SL-DATALOG$^{\mathrm{CSR}}$.

A syntactic construct that expresses path summarization is incorporated into graphical queries. We assume that we are given the "summarizing functions" MAX, MIN, SUM and PROD.

Definition 3.2: A *collecting list* is a bracketed list of distinct variables and underscores (_) of the form $<X_1, \ldots, X_n>$, where each variable in the list is

[3] A variation of this problem, in which the graph is expected to be acyclic, is to find the critical path in task scheduling problems. Later, in Example 3.2, we show how critical path is expressible in GraphLog by a combination of SUM path summarization and MAX aggregation.

called a *collecting variable*.

A *summarized term* is a term of the form $f(g(X))$, were X is a collecting variable, f is an aggregate function and g is a summarizing function.

A *summarized edge* is an edge in a query graph whose label is a regular expression E followed by a collecting list. The number of positions in the literal (that is, the arity of the predicate minus the number of terms labeling the endpoints of the edge) defined by the regular expression E has to be equal to the length of the collecting list. □

We extend the definition of query graph to contemplate the presence of both summarized edges and summarized terms, in addition to aggregate terms. All occurrences of collecting variables outside the unique collecting list in which they can appear are restricted to be within summarized terms in the label of the distinguished edge.

The semantics of the extended graphical queries whose query graphs were described above is given by a translation to a stratified Datalog program with non-recursive aggregation containing one closed semiring rule per summarized edge. Let denote by GRAPHLOG$^{\mathrm{CSR}}$ the resulting set of queries. We have the following straightforward expressive power result.

Lemma 3.2: GRAPHLOG$^{\mathrm{CSR}}$ = SL-DATALOG$^{\mathrm{CSR}}$

The following examples illustrate the use of path summarization combined with aggregation in GraphLog.

Example 3.2: Figure 2 shows four graphical queries expressing well-known combinations of path summarization and aggregation. The first one succinctly expresses the shortest path query described in Example 2.4.

Note that if the *underscore* (_) in a closure literal like $\mathsf{distance(\_)}^+$ is substituted by a variable (resp. a constant), then the edge labeled by the closure literal in the query graph would match paths in the database graph such that all the distance values along the path are the same unspecified (resp. constant) value, instead of matching paths with arbitrary distances along them. □

Example 3.3: Figure 3 shows a graphical query with two query graphs that finds the total weight of the parts in a parts hierarchy, a variation of the *bill*

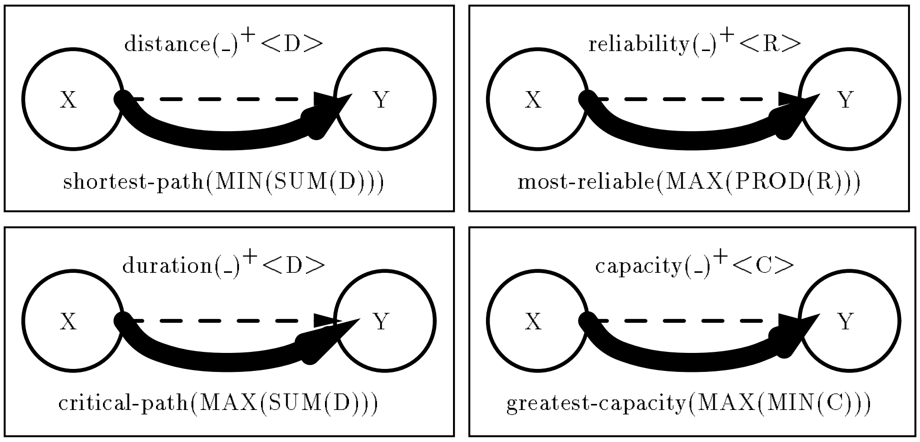

Figure 2: Combining path summarization and aggregation.

of materials query. There is a relation subpart(P1,P2,Q) that has the quantity of part P2 required to assemble part P1, and a relation weight(P,W) giving the weight W of all base (i.e., elementary) parts P.

The predicate total-quantity computes the parts explosion query. Note that, as a result of using the Kleene closure operator instead of positive closure, total-quantity(P,P,1) is defined for all parts P. The relation total-weight(P1,T) is computed by adding the weight that each base subpart of P1 contributes. The weight contribution of a base subpart P2 is the number of subparts required to assemble part P1 times the weight of base subpart P2. □

A frequently used form of path summarization is counting the lengths of paths. This can be accomplished by introducing an extra position with a constant 1 in the literals labeling the edges along the path, and then collecting the values 1 along the path in order to sum them inside a summarized term. For convenience, we introduce a syntactic construct to denote the valuation of a variable to the length of a path. If E is a regular expression, then an edge in a query graph with label $E[X]$ has X valuated to the length of the path described by the edge. An example using this notation is given below.

Example 3.4: Figure 4 shows an example from [KP87]. The relation shorter-than-any(X1,Y1,X2,Y2) represents a path in the graph given by the relation edge shorter than or equal to any path from X2 to Y2. In [KP87], it was

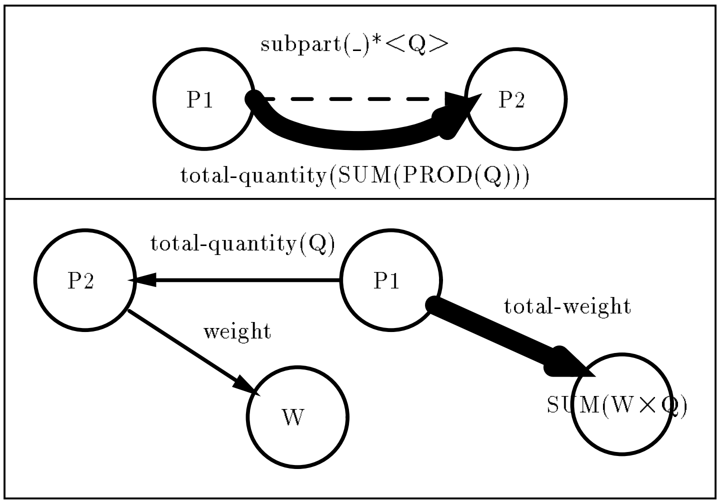

Figure 3: Total weight of the parts.

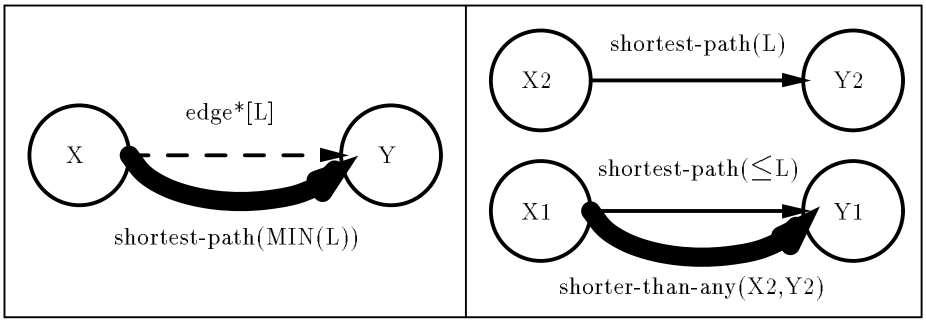

Figure 4: Paths from X1 to Y1 shorter than any path from X2 to Y2.

expressed in Datalog with negation and inflationary semantics (DATALOG$^{\neg}$). The label shortest-path(≤L) is a shorthand for the conjunction of the two literals shortest-path(X1,Y1,L1), and L1≤L. □

So far, the status of variables valuated to the length of paths is comparable to that of a summarizing function applied to a collecting variable. However, variables valuated to the length of paths do not need to be restricted to only appear within an aggregate term in the label of the distinguished edge. We can provide semantics for the valuation of variables to the length of paths independently of whether these variables are aggregated in a summarized term. The only special consideration is that we are careful to assign a $+\infty$

length to paths whenever it is possible to extend the path with a cycle.

The next example presents a familiar query that can be expressed in this way.

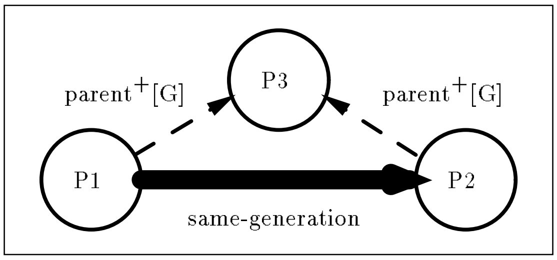

Figure 5: The "same generation" query.

Example 3.5: The well known *same generation* query is shown in Figure 5. Using the same variable G restricts the length of the paths to a common ancestor to be equal. □

Let GRAPHLOG$^{\text{LENGTH}}$ denote the set of GraphLog queries with aggregation (but without summarized edges and summarized terms) that make use of the above extension to find the length of paths. We can show that this addition does not change the data complexity of the extended language (by describing a non-deterministic Turing machine requiring logarithmic space that finds the length of paths by proceeding in a similar way as the machine described in the proof of Lemma 2.10).

Lemma 3.3: GRAPHLOG$^{\text{LENGTH}}$ $\subseteq$ QNLOGSPACE

Our last example combines aggregation and finding path lengths.

Example 3.6: Figure 6 shows the graphical query that finds the average salary of all the employees who are in a lower hierarchical level than employee E1. The relation avg-salary-below(E,C) defined in the second query graph gives the desired information. The label levels-below $\geq$ is a regular expression formed by the concatenation of the predicates that correspond to the relations levels-below(E2,L') and L'$\geq$L.

Notice that the levels are not known beforehand but are determined by the first query graph: the levels are assigned to the predicate levels-below

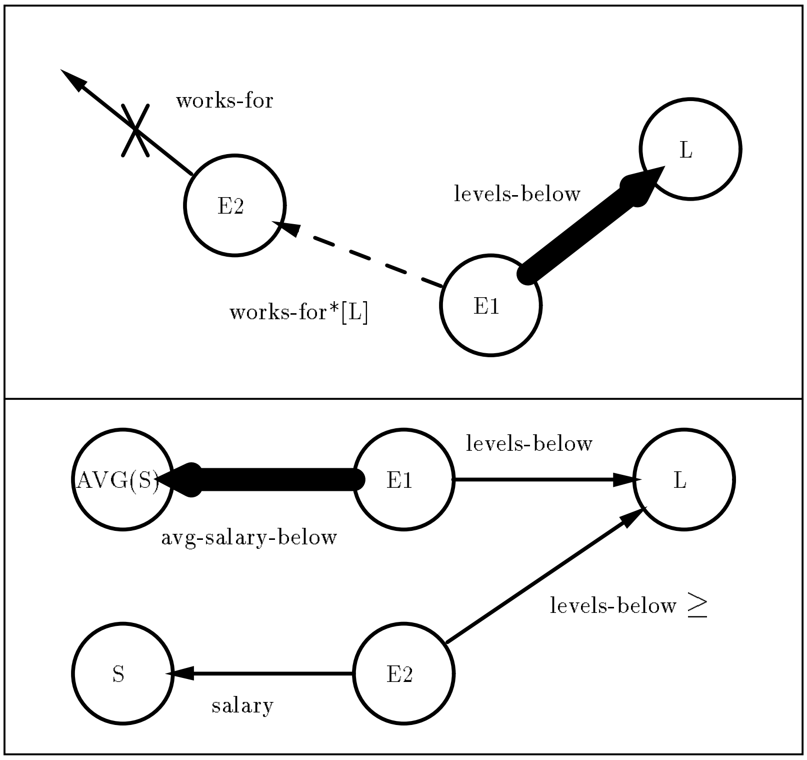

Figure 6: Average salary of the employees below E1.

by counting the edges in the path to the top of the hierarchy. The dangling negated edge labeled works-for is a graphical shorthand to indicate that there exists no employee for whom E2 works for. □

4 Conclusions

We have presented facilities for computing aggregate functions over sets of tuples and along paths in a database graph. These facilities are essential in any practical query language and until recently have received relatively little attention in database theory.

Figure 7 summarizes the complexity results in the paper. Height in the vertical direction indicates non-decreasing expressive power. Except for the containment of QAC^0 in QNC^1, which is known to be proper, all the others

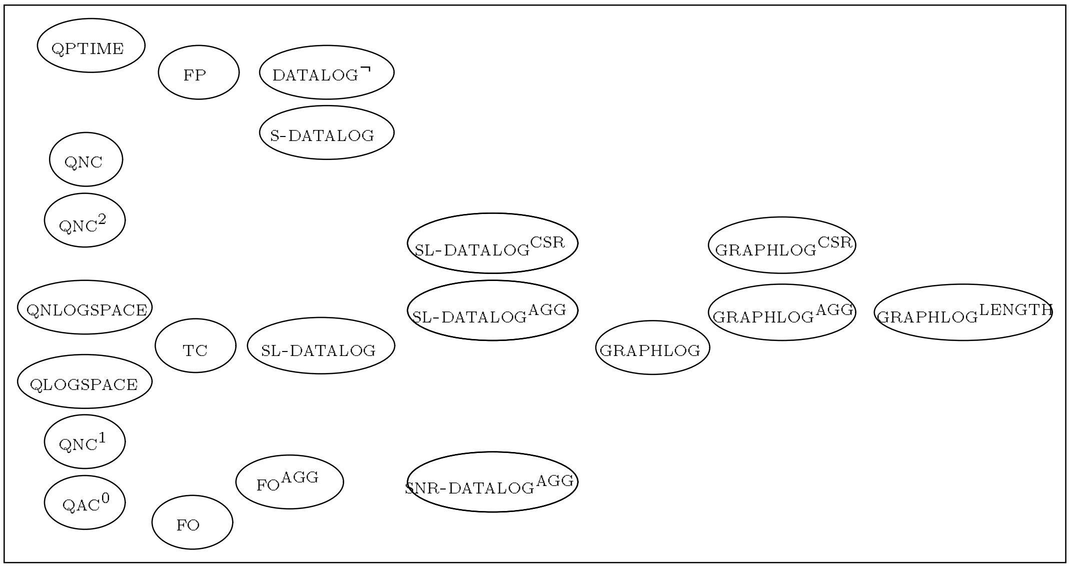

Figure 7: Relative expressive power of languages with aggregates.

are firmly believed to be proper but not proven.

It is interesting to point out that adding aggregate operators increases the complexity of the query language. This happens both at the first-order level, in going from FO$^<$ to FO$^{\text{AGG}}$, and at a higher level in going from GRAPHLOG$^<$ to GRAPHLOG$^{\text{CSR}}$. Another instance of this phenomena is the corporate takeover query: the problem is logspace-complete for PTIME and it can be expressed by a linear Datalog program with recursive aggregation, yet linear Datalog without aggregation is within NLOGSPACE.

There is still work to be done in completing the expressive power picture given above. How are the following classes related (aside from the immediate containments): S-DATALOG$^{\text{AGG}}$, S-DATALOG$^{\text{CSR}}$, ML-DATALOG$^{\text{AGG}}$, and M-DATALOG$^{\text{AGG}}$? There is significant potential for practical applications (and well motivated theoretical problems) in exploring the question of whether there exist any other interesting classes of aggregate queries, particularly within QNC.

Acknowledgments

We are indebted to Isabel Cruz for fruitful discussions and Steve Cook for his help with the finer points of parallel complexity theory. We would like to thank Inderpal Mumick for his comments, Vassos Hadzilacos for bringing [Coo80] to our attention, Dimitra Vista for her comments on the final version, and the referees for providing detailed recommendations.

References

[AB88] Serge Abiteboul and Catriel Beeri. On the power of languages for the manipulation of complex objects. Technical Report 846, INRIA, May 1988.

[ADJ88] R. Agrawal, S. Dar, and H.V. Jagadish. On transitive closure problems involving path computations. Technical report, AT&T Bell Laboratories, 1988.

[Agr87] R. Agrawal. Alpha: an extension of relational algebra to express a class of recursive queries. In *Proc. IEEE 3rd. International Conference on Data Engineering*, pages 580–590, 1987.

[BRSS92] Catriel Beeri, Raghu Ramakrishnan, Divesh Srivastava, and S. Sudarshan. The valid model semantics for logic programs. In *Proceedings of the Eleventh ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, pages 91–104, 1992.

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

[CM90a] Mariano Consens and Alberto Mendelzon. The G^+/GraphLog visual query system. In *Proceedings of the ACM-SIGMOD 1990 Annual Conference on Management of Data*, page 388, 1990. Video presentation summary.

[CM90b] Mariano Consens and Alberto Mendelzon. GraphLog: a visual formalism for real life recursion. In *Proceedings of the Ninth ACM SIGACT-SIGMOD Symposium on Principles of Database Systems*, pages 404–416, 1990.

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

[CN89] Isabel Cruz and Theodore Norvell. Aggregative closure: An extension of transitive closure. In *Proc. IEEE 5th International Conference on Data Engineering*, pages 384–391, 1989.

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

[Coo80] Eric C. Cooper. On the expressive power of query languages for relational databases. Technical Report TR-14-80, Aiken Computation Lab., Harvard University, 1980.

[Coo85] Stephen A. Cook. A taxonomy of problems with fast parallel algorithms. *Information and Control*, 64(1-3):2–22, 1985.

[Cru87] Isabel Cruz. G^+: A graphical system for recursive queries. Master's thesis, Dept. of Computer Science, University of Toronto, 1987.

[DS86] U. Dayal and J.M. Smith. Probe: A knowledge-oriented database management system. In Michel L. Brodie and John Mylopoulos, editors, *On knowledge base management systems*, pages 227–257. Springer-Verlag, 1986.

[GGZ91] Sumit Ganguly, Sergio Greco, and Carlo Zaniolo. Minimum and maximum predicates in logic programming. In *Proceedings of the Tenth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, pages 154–163, 1991.

[GJ79] M.R. Garey and D.S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness.* W.H. Freeman and Co., New York, 1979.

[GZG92] Sergio Greco, Carlo Zaniolo, and Sumit Ganguly. Greedy by choice. In *Proceedings of the Eleventh ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, pages 105–113, 1992.

[HAC90] Maurice Houtsma, Peter Apers, and Stefano Ceri. Complex transitive closure queries on a fragmented graph. In *Proceedings of the Third International Conference on Database Theory*, LNCS 470, pages 470–484. Springer-Verlag, 1990.

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

[Imm88a] Neil Immerman. Expressibility and parallel complexity. Technical report, Department of Computer Science, Yale University, 1988.

[Imm88b] Neil Immerman. Nondeterministic space is closed under complementation. *SIAM Journal on Computing*, 17(5):935–938, 1988.

[IR88] Y.E. Ioannidis and R. Ramakrishnan. Efficient transitive closure algorithms. Technical Report 765, Computer Sciences, Univ. of Wisconsin-Madison, 1988.

[Kan88] 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., 1988.

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

[KP87] P. G. Kolaitis and C.H. Papadimitriou. Why not negation by fixpoint? In *Proceedings of the Sixth ACM SIGACT-SIGMOD

Symposium on Principles of Database Systems, pages 231–239, 1987.

[KS91] David Kemp and Peter Stuckey. Semantics of logic programs with aggregates. In *Proceedings of the International Logic Programming Symposium*, pages 387–401, 1991.

[Lef91] Alexandre Lefebvre. Recursive aggregates in the eks-v1 system. Technical Report TR-KB-34, European Computer-Industry Research Center, 1991.

[MPR90] Inderpal Mumick, Hamid Pirahesh, and Raghu Ramakrishnan. The magic of duplicates and aggregates. In *Proceedings of the Sixteenth International Conference on Very Large Data Bases*, pages 264–277, 1990.

[MS90] Michael Mannino and Leonard Shapiro. Extensions to query languages for graph traversal problems. *IEEE Transactions on Knowledge and Data Engineering*, 2(3):353–363, September 1990.

[OOM87] G. Ozsoyoglu, Z.M. Ozsoyoglu, and V. Matos. Extending relational algebra and relational calculus with set-valued attributes and aggregate functions. *ACM Transactions on Data Base Systems*, 12(4):566–592, 1987.

[RHDM86] A. Rosenthal, S. Heiler, U. Dayal, and F. Manola. Traversal recursion: A practical approach to supporting recursive applications. In *Proceedings of the ACM-SIGMOD 1986 Annual Conference on Management of Data*, pages 166–176, 1986.

[RS92] Kenneth Ross and Yehoshua Sagiv. Monotonic aggregation in deductive databases. In *Proceedings of the Eleventh ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, pages 114–126, 1992.

[She90] Yeh-Heng Sheng. IDLOG: Extending the expressive power of deductive database languages. In *Proceedings of the ACM-SIGMOD 1990 Annual Conference on Management of Data*, pages 54–63, 1990.

[SR91] S. Sudarshan and Raghu Ramakrishnan. Aggregation and relevance in deductive databases. In *Proceedings of the Seventeenth International Conference on Very Large Data Bases*, pages 501–511, 1991.

[TZ86] S. Tsur and C. Zaniolo. LDL: a logic-based data-language. In *Proceedings of the Twelve International Conference on Very Large Data Bases*, pages 33–41, 1986.

[Ull88] J.D. Ullman. *Principles of Database and Knowledge-Base Systems*, volume 1. Computer Science Press, Potomac, Md., 1988.

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

[VG92] Allen Van Gelder. The well-founded semantics of aggregation. In *Proceedings of the Eleventh ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, pages 127–138, 1992.