How expressive is stratified aggregation?

Inderpal Singh Mumick
AT&T Bell Laboratories, Murray Hill, NJ 07974, USA
E-mail: mumick@research.att.com

Oded Shmueli
Department of Computer Science, Technion – Israel Institute of Technology, Haifa 32000, Israel
E-mail: oshmu@cs.technion.ac.il

We present results on the expressive power of various deductive database languages extended with stratified aggregation. We show that (1) Datalog extended with stratified aggregation cannot express a query to count the number of paths between every pair of nodes in an acyclic graph, (2) Datalog extended with stratified aggregation and arithmetic on integers (the + operator) can express all *computable* queries on ordered domains, and (3) Datalog extended with stratified aggregation and generic function symbols can express all *computable* queries (on ordered or unordered domains). Note that without stratified aggregation, the above extensions of Datalog cannot express all computable queries. We show that replacing stratified aggregation by stratified negation preserves expressiveness. We identify subclasses of the above languages that are complete (can express all, and only the, computable queries).

1. Introduction

Datalog [20] is a query language for deductive databases. Datalog is based on Horn clauses. The relations stored in deductive database are called the *edb* (extensional database) relations, and the relations defined by Datalog queries (essentially views) are called the *idb* (intensional database) relations.

1.1. MOTIVATION

Extensions of Datalog to handle aggregation are a popular research topic [13, 4, 5, 15, 9, 21, 17]. Several authors have argued that stratified aggregation is not expressive enough, and non-stratified aggregation is needed. Well-founded semantics, stable semantics, and valid semantics for aggregation have been proposed to define the intuitive meaning of programs with non-stratified aggregation. At this point, one must ask the question: How expressive is simple stratified aggregation? This paper presents a theoretical study of the expressive power of Datalog (denoted D) extended by one or more of stratified aggregation, the arithmetic +

operator, function symbols, and stratified negation. While we do not argue the need for non-stratified aggregation for ease of programming and efficiency of implementation, this paper shows that stratified aggregation is sufficient, in a theoretical sense, to express all *computable queries* [3].

Apt and Blair [2] have shown that every query expressible in Datalog extended with function symbols and stratified negation, such that the characteristic function of every negated predicate is recursive, is Turing computable. One may wonder whether the above language expresses all computable queries. Another interesting problem is to define appropriate restrictions on a language with stratified aggregation so that all queries expressed in the language are computable. Lastly, would these restrictions result in a complete language.

1.2. NOTATION

Our version of Datalog includes constants and the built-in predicates $<, \leq, >, \geq, \neq$. One of the predicates in a Datalog program is denoted as the *query* predicate. The answer to a Datalog program is the relation for the query predicate that can be derived from a given instance of the extensional database. Datalog rules may be adorned using the b (bound) and f (free) adornments [19] to indicate that certain arguments of a predicate are guaranteed to be bound in any subgoal.

Datalog will be extended with stratified aggregation [12] by permitting a second order "Groupby" predicate of the form:

$$(G):\quad \texttt{GROUPBY}(r(\bar{t}),\ [Y_1, Y_2, \ldots, Y_m], [Z_1 = A_1(E_1), \ldots, Z_n = A_n(E_n)]),$$

to appear as a subgoal in a Datalog rule. The groupby predicate takes as arguments a predicate r with its attribute list $\bar{t}$, a grouping list $[Y_1, Y_2, \ldots, Y_m]$, and an aggregation list AL of aggregation specifications. The predicate r is called the grouping predicate. The arguments, $\bar{t}$ of r can be general terms: constants, variables or complex terms. However, all variables in $\bar{t}$ are local to the GROUPBY construct, unless they also appear in the grouping list $[Y_1, Y_2, \ldots, Y_m]$. The grouping list consists of zero or more distinct variables that must appear in $\bar{t}$. The aggregation list consists of a list of terms of the form $Z_i = A_i(E_i)$. Each Z_i is a new variable, E_i is an expression that uses variables of $\bar{t}$, and A_i is an aggregate operator that maps a set of values onto a single value (e.g. SUM, COUNT). Within the rule body, a GROUPBY subgoal represents a relation over variables $Y_1, \ldots, Y_m$ in the grouping list and variables $Z_1, \ldots, Z_m$ in the aggregation list.

Example 1.1. Consider an edb table $\texttt{contains}(P, S, C)$, meaning that part P contains C copies of a subpart S.

$$(E1):\quad \texttt{num\_tuples}(N) \texttt{ :- } \texttt{GROUPBY}(\texttt{contains}(P, S, C), [\,], N = \texttt{COUNT}(P)).$$

$(E2)$: needs(P,N) :– GROUPBY(contains(P,S,C), $[P]$, $N =$ SUM(C)).

Rule $E1$ computes the total number of tuples in the contains relation. The empty grouping list causes the entire relation to form one group. The COUNT aggregation function then counts the number of tuples in this one group.

Rule $E2$ computes the total number of subparts for each part. The tuples are grouped based upon their P value, and the aggregation function SUM then adds up the count fields for all tuples in each group. ■

Let D^a be Datalog extended with stratified aggregation as above, with SUM, AVG, MIN, MAX, and COUNT as the only aggregate functions. Stratification means that if a derived relation $r1$ is defined by applying aggregation on a derived relation $r2$, then relation $r2$'s definition does not depend, syntactically, on relation $r1$. Let $D^{a(\mathtt{MAX})}$ be a restriction of D^a where MAX is the only aggregation function, and let $D^{a(\mathtt{COUNT,MAX})}$ be a restriction of D^a where COUNT and MAX are the only aggregation functions.

Other Datalog extensions: Let $D^{\neg}$ be Datalog extended with safe stratified negation. Let D^f be Datalog extended with two function symbols, a unary function symbol f and the list constructor [ ], and a special constant, 0. Let D^+ be Datalog extended with the evaluable function + and two special constants, 0 and 1. Let D^{a+} and $D^{\neg+}$ be versions of the languages D^a and $D^{\neg}$, respectively, extended with the evaluable function + and two special constants, 0 and 1. Let D^{af} and $D^{\neg f}$ be versions of the languages D^a and $D^{\neg}$, respectively, extended with the two function symbols of D^f and one special constant, 0.

Let $D^{v(\neg)}$ be a restriction of $D^{\neg}$ such that every negated predicate is *verifiable* (to be defined in Section 4). Let $D^{l(a)}$ be a restriction of D^a such that every aggregated predicate is *finitely computable* (to be defined in Section 4).

The universal domain U, is the domain from which the attribute values of the input and output relations are drawn. The domain of a query is the same as the universal domain under consideration. The computation domain contains at least the universal domain, and may include an auxiliary domain, such as the domain of integers or the domain of functional terms.

1.3. SUMMARY OF RESULTS

We obtain the following results:

1. The *summarized explosion query* asks to compute the number of instances of a part needed to construct a bigger part. The summarized explosion query cannot be expressed in the language D^a.
2. The languages $D^{a(\mathtt{MAX})+}$ and $D^{\neg+}$ can express all computable queries on the integer domain. The languages $D^{l(a(\mathtt{MAX}))+}$ and $D^{v(\neg)+}$ can express all, and only the, computable queries on the integer domain.

3. The languages $D^{a(\mathtt{COUNT},\mathtt{MAX})+}$ and $D^{\neg+}$ can express all computable queries on ordered domains, provided that the computation domain includes the non-negative integers. The languages $D^{l(a(\mathtt{COUNT},\mathtt{MAX}))+}$ and $D^{v(\neg)+}$ can express all, and only the, computable queries on ordered domains, provided that the computation domain includes the non-negative integers.
4. The languages $D^{a(\mathtt{MAX})f}$ and $D^{\neg f}$ can express all computable queries (on ordered or unordered domains), provided that the computation domain includes functional terms. The languages $D^{l(a(\mathtt{MAX}))f}$ and $D^{v(\neg)f}$ can express all, and only the, computable queries (on ordered or unordered domains), provided that the computation domain includes functional terms. Here, MAX operates on terms of the form $f^i(0)$ rather than on integers.

The result on non-expressivity of summarized explosion is very interesting, and in fact unintuitive [14]. The result was conjectured in [13], but not proven. We provide declarative query languages that are complete. Other complete languages [3, 8, 1] have included procedural looping constructs (**while**).

1.4. APPROACH

A well known result is that Datalog with function symbols can simulate a Turing machine [18], or a partial recursive function [11]. This does not mean that the language D^f can express every computable query on relations, since a Turing machine requires the input relation to be coded on tape, and a Datalog query must be formulated on a relation (that is not coded on a tape). Similarly, a partial recursive function requires the input relation to be coded as an integer (or a tuple of integers), and a D^f query is on relations that are not coded as integers. For example, we cannot express the query to count the number of tuples in a relation using the language D^f. In fact, the language D^f cannot express any non-monotonic query. What this means is that the language D^f lacks the ability to code relations on a tape, or as integers. If stratified aggregation can help us encode relations into integers and to decode integers into relations, we could then use the remaining language to derive expressiveness results. This is indeed the case.

We prove that a relation can be coded into a single integer using stratified aggregation and the arithmetic + function on integers, or stratified aggregation (using MAX on functional terms) and the successor function symbol on uninterpreted domains.

It is known that Datalog with function symbols and stratified negation can express queries that are not computable [2]. However if we restrict negation to be over predicates whose characteristic function is recursive, then every expressible query is computable. The above restriction permits a stratum to be evaluated without waiting for the computation of the lower stratum to be completed. When stratified aggregation is used, the requirement that the characteristic function of every aggregated predicate be recursive is not sufficient to guarantee that expressible queries will be computable. We must impose restrictions that let us compute the aggregated function in a finite amount of time.

The paper is organized as follows. In Section 2 we provide background concerning computable queries. Section 3 shows that Datalog extended with stratified aggregation cannot express certain path aggregation queries. Section 4 defines *finitely computable* queries, and proves that if we restrict stratified aggregation to be over finitely computable predicates, every expressible query is computable. Section 5 characterizes the expressiveness of D^{a+} and $D^{\neg+}$ on the integer domain. We show that the above languages are expressive, and that $D^{l(a)+}$ and $D^{v(\neg)+}$ are complete. Section 6 extends this treatment to non-integer, i.e. "uninterpreted", ordered domains. We consider unordered uninterpreted domains in Section 7, and show that the languages D^{af} and $D^{\neg f}$ are expressive on unordered uninterpreted domains, while $D^{l(a)f}$ and $D^{v(\neg)f}$ are complete. The completeness results extend to typed queries on multiple domains (Section 8). Related work is discussed in Section 9, and conclusions are presented in Section 10.

2. Background

We use $\Subset$ to denote a finite subset. Thus, $A \Subset B$ means that A is a finite set, and that A is a subset of B. All rules are required to be safe, meaning that all variables used in the head of a rule are bound through (1) an appearance in a positive subgoal, or (2) appearance in the grouping or aggregation list of a groupby subgoal, or (3) appearance in a bound argument of the adorned rule head, or (4) being equated to an otherwise bound variable. Negation and aggregation are used in a stratified manner, meaning that there are no cycles involving negation or aggregation in the dependency graph [20] of a program. Further, all negation must be safe, meaning that all variables used inside a negated subgoal are bound in the rule body prior to being used.

The programs are given the traditional bottom-up fixed point semantics. We present the bound-free (bf) adorned versions of the programs. If needed, we assume that a program will undergo a transformation that will introduce extra body literals in each rule to ensure that all variables that are indicated bound (b) in the adorned rule head now appear in the additional body literals. The transformation we use is the magic sets transformation [20]. Special care is taken when applying the magic-sets transformation to aggregation and negation rules to ensure that stratification is retained.

The characteristic function of an n-ary relation $R(X_1, \ldots, X_n)$ is an n-ary function $f(X_1, \ldots, X_n)$ such that $f(x_1, \ldots, x_n) = 1$ iff the tuple $(x_1, \ldots, x_n) \in R$, and $f(x_1, \ldots, x_n) = 0$ iff the tuple $(x_1, \ldots, x_n) \notin R$. We use the standard definitions of recursive and partial recursive functions from recursive function theory [7].

2.1. COMPUTABLE QUERIES

Computable queries were defined in a seminal paper by Chandra and Harel [3]. We present an alternative and equivalent formulation of their definitions.

Let U be a countably infinite universal domain. U is the universe out of which the database tuples' attribute values are chosen.

In addition, certain languages may use additional domains while computing, for example the integer domain or a domain containing complex terms involving function symbols. The domain used during computation will be denoted as the *computation domain*. A computation domain will always include the input and the output domains.

A database $B = (R_1, \ldots, R_m)$ of type $\bar{a} = (a_1, \ldots, a_m)$ is a collection of relations $R_1, \ldots, R_m$ of arities $(a_1, \ldots, a_m)$, where each relation R_i is a finite subset of U^{a_i}.

Definition 2.1. Encoding: *A* $1-1$ *onto mapping e from set A (the coded set) to set B (the codes) is called an encoding. The sets A and B may be either finite or countably infinite. An encoding is an invertible function, with* e^{-1} *being the inverse of encoding e.*

Let I be the set of non-negative integers. Given an encoding e from U to I, we can map the database $B = (R_1, \ldots, R_m), R_i \subseteq U^{a_i}$ to a database

$$e(B) = (e(R_1), \ldots, e(R_m)), e(R_i) \subseteq I^{a_i}$$

on the domain of integers, by replacing each element k with the integer $e(k)$ in every tuple of every relation R_i. Given a database B_I on the domain of integers (such as the database $e(B)$) we can define a function g that maps the database B_I onto a single integer $g(B_I)$. The function g uses a fixed order for relations, ordering by lexicographic order of relation names, and a fixed order for tuples within a relation, in increasing order of the integer encoding of tuples. The database $e(B)$ can thus be mapped to a single integer $g(e(B))$. The coding of domain U is implicit in the encoding e, and is not represented in the integer $g(e(B))$. The integer $g(e(B))$ is called the *code* for database B. Furthermore, g codes the database in a fixed format, say using prime numbers exponentiations, so that the integer $g(e(B))$ can always be effectively decoded to get the database $e(B)$ (the encoding of the database B under e).

Let the function g be fixed from now on.

Definition 2.2. Query: *A query is a partial function from databases to relations.*

Definition 2.3. WC-computable[2] queries [3]: *Let U, the universal domain, be a given*

[2] Without constants computable.

countably infinite set. Let Q be a partial function from databases $B = (R_1, \dots, R_m)$ of type $\bar{a} = (a_1, \dots, a_m)$, with each $R_i \subseteq U^{a_i}$, to output relations $R_o \subseteq U^{a_o}$.

The query Q is a wc-computable query if there exists a partial recursive function f from integers to integers such that, for all databases B of type $\bar{a}$, and for all encodings e from U to I, the function f maps $g(e(B))$ onto the integer $g(e(R_o))$, where $R_o = Q(B)$.

If Q is undefined on B, then $f(g(e(B)))$ must be undefined for all encodings e.

We Note that:

- Q is a function. So, the result relation R_o obtained by decoding the output $g(e(R_o))$ of the partial recursive function f using the inverse encoding function e^{-1} must be the same for all encodings e.
- In Definition 2.3, the partial recursive function f on integers has no direct access to information concerning the encoding of any particular constant in U when operating upon the code of the database.

The following theorem says that the queries computable according to Definition 2.3 satisfy the isomorphism condition of [3]. An *isomorphism* is a 1-1 mapping extended to tuples and relations.

Theorem 2.1. *Given a universal domain U, two databases B_1 and B_2 of type $\bar{a}$, an isomorphism h from B_1 to B_2 such that $B_2 = h(B_1)$, and a wc-computable query Q from databases of type $\bar{a}$ to relations of type a_o; let $Q(B_1) = R_1$ and let $Q(B_2) = R_2$. Then*

$$R_2 = h(R_1).$$

Proof. Consider two isomorphic databases B_1 and B_2, both over a universal domain U. Consider any set $D_1 \subseteq U$ that contains all constants in database B_1. Let the cardinality of set D_1 be n. Let $D_2 \subseteq U$ be the set of cardinality n that contains all constants in database B_2. Consider a function h from D_1 to D_2 defining an isomorphism between B_1 and B_2. Consider an arbitrary order $a_1 < \dots < a_n$ on the elements D_1 and the implied order $h(a_1) < \dots < h(a_n)$ on the elements of D_2. Let $e_1 : U \to I$ be an encoding of D_1 that maps a_i to i. Let $e_2 : U \to I$ be an encoding of D_2 that maps $h(a_i)$ to i. The isomorphism function h is therefore a sub-function of $(e_2^{-1} \circ e_1)$, i.e. $h(a) = (e_2^{-1} \circ e_1)(a)$ for all elements $a \in D_1$. Let f be the partial recursive function associated with query Q. Let g be the function that maps databases to single integers based on their encoding.

- By definition of wc-computable, $f(g(e_1(B_1))) = g(e_1(R_1))$ where $Q(B_1) = R_1$.
- By definition of wc-computable, $f(g(e_2(B_2))) = g(e_2(R_2))$ where $Q(B_2) = R_2$.
- $B_2 = h(B_1) = (e_2^{-1} \circ e_1)(B_1)$ and so $e_2(B_2)) = e_2((e_2^{-1} \circ e_1)(B_1)) = e_1(B_1)$.
- Since f is a partial recursive function defining query Q, $f(e_2(B_2)) = e_2(R_2)$, and $f(e_1(B_1)) = e_1(R_1)$. But $e_2(B_2) = e_1(B_1)$, so $e_2(R_2) = e_1(R_1)$.

- And, thus $R_2 = e_2^{-1}(e_2(R_2)) = e_2^{-1}(e_1(R_1)) = (e_2^{-1} \circ e_1)(R_1) = h(R_1)$, i.e. the isomorphism is ensured. ■

We provide an extension of WC-computable queries to allow queries with constants. Our definition is an alternative formulation of a similar extension in [3].

Definition 2.4. Constant query: *Let R_k be a given finite subset of the universal domain U. A query Q is a constant query if for all databases B of any type, $Q(B) = R_k$. (Note that R_k is an arity 1 relation.)*

Definition 2.5. Computable queries: *Let U be a given countably infinite universal domain set. Let Q be a partial function from databases $B = (R_1, \ldots, R_m)$ of type $\bar{a} = (a_1, \ldots, a_m)$, with each $R_i \Subset U^{a_i}$, to the output relation $R_o \Subset U^{a_o}$.*

The query Q is a computable query if there exists a partial recursive function f from I^{n+1} to I for some n, and there exist a finite set of constant queries,

$$Q_1, \ldots, Q_n,$$

such that, for all databases B of type $\bar{a}$, and for all encodings e from U to I, the function f maps $(g(e(B)), g(e(Q_1)), \ldots, g(e(Q_n)))$ onto the integer $g(e(R_o))$. (Strictly speaking we should write $Q_i(B)$ instead of Q_i.)

If Q is undefined on B, then $f(g(e(B)), g(e(Q_1)), \ldots, g(e(Q_n)))$ must be undefined for all encodings e.

In Definition 2.5, as in Definition 2.3, the partial recursive function f has no information about the encoding. However, any constant the function f "cares about" must be encoded using the same encoding e as the rest of the database, and be made available to the function f in a separate argument. Consequently, the queries defined in Definition 2.5 can compute constant queries, as well as other queries that may use constants.

Definition 2.6. Expressiveness: *A language L is expressive if every computable query can be expressed in L.*

We first notice that the language D^f is not expressive:

Lemma 2.1. *There exist computable queries that cannot be expressed in Datalog extended with function symbols.*

An example of such a computable, but non-D^f-expressible, query is $Q_{\mathtt{size}}(R, S)$ that computes the relation R if the number of tuples in relation R is greater than the number of tuples in relation S, else it computes the empty relation. The query $Q_{\mathtt{size}}(R, S)$ is non-monotonic since adding tuples to S may make previously computed tuples in the result invalid. The language D^f is monotonic, hence it cannot express the query $Q_{\mathtt{size}}(R, S)$.

Definition 2.7. Boundedness: *A language L is bounded if every query that can be expressed in language L is computable.*

Definition 2.8. Completeness: *A language L is complete if every computable query can be expressed in L, and if every query that can be expressed in language L is computable.*

3. Stratified aggregation cannot express summarized explosion

The language D^a with stratified aggregation can express some queries not expressible in the language D^f. For instance, the size comparison query $Q_{\mathtt{size}}(D, R, S)$ is expressed by the following program Sz:

$$(Sz1):\quad Q_{\mathtt{size}}(U) :- r(U)\ \&\ \mathtt{GROUPBY}(r(X_1), [\,], Y = \mathtt{COUNT}(X_1))\ \&$$
$$\mathtt{GROUPBY}(s(X_2), [\,], Z = \mathtt{COUNT}(X_2))\ \&\ Y > Z.$$

The groupby subgoal $\mathtt{GROUPBY}(r(X_1), [\,], Y = \mathtt{COUNT}(X_1))$ is interpreted as follows (Section 1.2): Take the relation $r(X)$, and let $Y =$ the count of the number of values of X. The groupby subgoal then represents a relation on attribute Y.

We prove that the following bill-of-materials query cannot be expressed in the language D^a.

Definition 3.1. Summarized explosion query *SE*: *Consider a relation* $\mathtt{subpart}(P, S, C)$, *listing the direct subparts S for each part P in a car. The attribute C is a count of the number of copies of subpart S used in part P. The aim is to derive a relation* $\mathtt{need}(P, S, C)$ *that gives a count for the number of copies of component S needed to construct one copy of part P. Assume that P, S, and C are integers.*

Theorem 3.1. *The summarized explosion query SE cannot be expressed in the language D^a.*

Proof. The idea underlying the proof is a non-trivial counting argument in which we derive a bound on the largest integer that can be computed by a program with k strata and maximum predicate arity A; thereby showing that no program P can compute the query SE on a part-subpart graph G.

Let, if possible, $P \in D^a$ be an aggregate stratified program for query SE. Let the maximum arity of a predicate in program P be A (clearly, $A \geq 3$) and let program P have k strata, where each stratum is defined through an aggregation over lower strata. We derive a bound on the largest integer that can be computed by a program with k strata and maximum arity A. Assume that a database $\mathcal{D}$ is for a part-subpart labeled directed graph G with M nodes, where the edge (u, v) expresses the fact that part v is a subpart of part u, and the label indicates the number of vs parts contained in u. Assume that the maximum integer in any `subpart` tuple is C. We assume $C \geq 2$ and $M \geq 2$.

Let S_n be the maximum count value in any predicate at stratum n. Then the maximum number of tuples in any one relation in stratum n is $(M + S_n)^A$. Since a count value at stratum $n+1$ is obtained by aggregation on a single relation of stratum n, the maximum count value in a relation of stratum $n+1$, obtained by using SUM , is $S_{n+1} = S_n(M + S_n)^A$.

With $S_1 = C$ (assume C is greater than the largest integer representing a node), $M \geq 2$, $C \geq 2$, and $A \geq 3$, one can verify that $S_n \leq T_n$, where $T_1 = S_1 = C$ and $T_{n+1} = (MT_n)^{(A+1)} \leq (CM)^{(A+1)^n}$. Thus, the maximum count computable by P is $S_k \leq (CM)^{(A+1)^k}$.

Consider a part-subpart graph G that looks like a chain, $\{(1, 2, C), (2, 3, C), \ldots, (M-1, M, C)\}$. The answer to query SE on graph G includes the tuple $(1, M, C^{M-1})$.

For sufficiently large C and M, $C^M > S_k$. Hence, program P cannot compute the query SE on graph G. ■

The above technique for showing a non-expressivity result applies to many closed semi-ring problems. It is a property of closed semi-ring problems that the number of aggregations needed is a function of the database. In an aggregate stratified program, the number of possible aggregations is fixed by the program. Thus, closed semi-ring problems are not likely to be expressible by an aggregate stratified program. In the case of query SE, each aggregation leads to an increase in the count, and it was impossible to simulate an arbitrary number of increases by a fixed number of strata.

It is straightforward to show that every D^a query can be computed in polynomial time. However, there exist polynomial time queries that cannot be expressed in D^a. For example, the summarized explosion query can be computed in polynomial time, but cannot be expressed in D^a.

Corollary 3.1. *Every query expressed in Datalog extended with stratified aggregation can be computed in polynomial time, but there exist polynomial time queries that cannot be expressed in Datalog extended with stratified aggregation.*

4. Boundedness

It was shown by Apt and Blair [2] that the languages $D^{\neg +}$ and $D^{\neg f}$ are not bounded. One can encode a partial recursive function into a predicate; negation can then be used to check whether the predicate is defined on a given input – a problem known to be undecidable. The non-bounded results applies for queries over an atomic universal domain, as well as for queries over a universal domain that contains function symbols. Apt and Blair defined a bounded class of programs, which we formulate below.

In the definitions that follow, we assume that the universal domain is countably infinite, and may include function symbols. Since the database relations, the intermediate predicates, and the query predicate may have functional terms in their extension, we need to extend the coding function g introduced in Section 2.1 to code functional terms. We assume a finite number of function symbols with a total order defined on them (similar to the total order on relation names). This enables defining g so that it can code terms involving function symbols.

Definition 4.1. Verifiability: *A predicate p in a program P, having n constants*[3] *$c_1, \dots, c_n$, is said to be verifiable if the characteristic function of p is* recursive. *That is, there exists a Turing machine M such that when M is presented on its input tape with*

$$g(e(B)), g(e(Q_1)), \dots, g(e(Q_n)), g(e(t)),$$

where Q_i is a constant query returning the constant c_i, and t is a tuple, M outputs 0 if t is not in the extension of predicate p given database B; otherwise M outputs 1.

Given a verifiable predicate p, it is possible to compute in finite time, whether a given tuple is in the relation for p or not. It may not be possible to compute the extension of p, which may be infinite.

Theorem 4.1. *[2] Let $D^{v(\neg)f}$ be Datalog extended with function symbols and stratified negation, such that every negated predicate is verifiable. The language $D^{v(\neg)f}$ is bounded.*

Proof. Given a program P in $D^{v(\neg)f}$ with a query predicate computing the output relation R_0, define a constant relation Q_i of arity one for each constant i used in program P. The relation Q_i has only one tuple, the constant i. Then, rewrite program P into program P' without constants, substituting relation Q_i for constant i.

Without loss of generality let e be an encoding. There exists a Turing machine M_1 such that whenever M_1 is presented with $(g(e(B)), g(e(Q_1)), \dots, g(e(Q_n)))$, on its input tape, M_1 computes $g(e(R_o))$ on its output tape. Operationally, M_1 mimics a bottom-up computation of P' and derives the result relation on a tape. M_1 makes calls to auxiliary Turing machines to obtain a yes or no answer concerning any negated ground literals. Such auxiliary machines exist because of verifiability.

[3] A *constant* is an element of the computation domain, so constants are actually built-in, or pre-interpreted, in the language.

Let M_2 be a Turing machine implementing g^{-1} and M_3 a Turing machine implementing g. Using M_1,M_2, and M_3 as "subroutines", we can construct a Turing machine M that maps integers to integers (strictly speaking, representations of integers to representations of integers on a Turing machine tape). So, there exists a partial recursive function f, from I^n to I, which accomplishes the same mapping as M. The existence of f implies that the query is computable, because f when applied to $(g(e(B)), g(e(Q_1)), \ldots, g(e(Q_n)))$, computes $g(e(R_o))$. ■

The boundedness result above does not extend to stratified aggregation. For example, consider the program P where q is a verifiable predicate, and predicate p is defined as:

(r): $p(X, Y)$:– $\mathtt{GROUPBY}(q(X, Z), [X], [Y = \mathtt{MAX}(Z)])$.

The grouping involves an implicit negation on an existential query on q: a value y is the maximum Z value if there does not exist a value Z_1 that is larger. Knowing that the characteristic function of predicate q is recursive is not sufficient to compute p.

To compute with aggregations, it is important to be able to enumerate the relation of the predicate being aggregated upon. Again, we assume that g can code terms involving function symbols.

Definition 4.2. Finitely computable queries: *Let U_f be a given countably infinite universal domain that may include function symbols. Let Q be a* total *function from databases $B = (R_1, \ldots, R_m)$ of type $\bar{a} = (a_1, \ldots, a_m)$, with each $R_i \subseteq U_f^{a_i}$, to the output relation $Q(B) = R_o \subseteq U_f^{a_o}$.*

The query Q is a finitely computable query if there exists a recursive *function f from I^{n+1} to I for some n, and there exists a finite set of constant queries, $Q_1(B), \ldots, Q_n(B)$, such that, for all databases B of type $\bar{a}$, and for all encodings e from U_f to I, the function f maps $(g(e(B)), g(e(Q_1(B))), \ldots, g(e(Q_n(B))))$ onto the integer $g(e(R_o))$.*

Note that a finitely computable query is such that the mapping from the input database to the output is defined by a *total* function. Thus, given any database, the output must be defined, and finite. Further, the output can be computed by applying the corresponding total recursive function to an encoding of the database. The finitely computable queries can be related to verifiable predicates as follows:

Definition 4.3. Finitely computable predicates: *Let P be a program and q an idb predicate in P. Predicate q is finitely computable if the program with q as its query predicate computes a query which is finitely computable.*

Lemma 4.1. *If q is the query predicate of a finitely computable query, then q is a verifiable predicate. If p is a verifiable predicate, then the query corresponding to p may not be a finitely computable query.*

Proof. Given a finitely computable query q, we will construct a Turing machine M that evaluates the characteristic function of q and always halts, thereby proving that q is a verifiable predicate. M operates as follows. Given $g(e(B)), g(e(Q_1(B))), \ldots, g(e(Q_n(B))), g(e(t))$ on its input tape, it calls on an auxiliary machine $M1$ to compute the finite query corresponding to q (with $g(e(B)), g(e(Q_1(B))), \ldots, g(e(Q_n(B)))$ on its tape). M then checks whether $g(e(t))$ is in the result computed by $M1$.

Consider a predicate p that is true of all integers. The characteristic function of p is certainly recursive. However, since the output is infinite, the query corresponding to p is not finitely computable. ■

We can now state the following boundedness result:

Theorem 4.2. *Let $D^{l(a)f}$ be Datalog extended with function symbols and stratified aggregation, such that every aggregated predicate is finitely computable. The language $D^{l(a)f}$ is bounded.*

Proof. The proof is essentially the same as in Theorem 4.1. The only difference is that prior to doing any aggregation, an auxiliary Turing Machine for a finitely computable predicate q is called, it computes the whole answer to the query q, and then evaluates the aggregation. ■

5. Expressiveness on the integer domain

In this section we restrict the universal domain U to be the set of non-negative integers I. The universal domain is thus totally ordered. The computation domain will be equal to the universal domain.

The results derived in this section can extend to arbitrary ordered domains, and we do so in the next section. First we define computable queries on ordered domains:

Definition 5.1. Order-preserving encoding: *An encoding e from an ordered set $A = \{a_1, a_2, \ldots\}$ to an ordered set $B = \{b_1, b_2, \ldots\}$ is said to be order preserving if the following holds:*

$$(a_i < a_j) \Rightarrow (e(a_i) < e(a_j)).$$

We use $<$ to denote the ordering relation in both the sets A and B.

Definition 5.2. Computable queries on ordered domains: *Let U be a given* ordered *countably infinite universal domain set. Let Q be a partial function from databases $B = (R_1, \ldots, R_m)$ of type $\bar{a} = (a_1, \ldots, a_m)$, with each $R_i \subseteq U^{a_i}$ to output relations $R_o \subseteq U^{a_o}$.*

The query Q is a computable query on an ordered domain if there exists a partial recursive function f from I^{n+1} to I, and there exist a finite set of constant queries, $Q_1, \ldots, Q_n$, such that, for all databases B of type $\bar{a}$, and for all order-preserving encodings e from U to I, the function f maps the tuple $(g(e(B)), g(e(Q_1)), \ldots g(e(Q_m)))$ onto the integer $g(e(R_o))$.

If Q is undefined on B, then $f(g(e(B)), g(e(Q_1)), \ldots, g(e(Q_n)))$ is undefined for all encodings e.

Definition 5.3. Computable queries on the integers: *Let $U = I^+$, that is, the universal domain is the set of non-negative integers, with a total ordering on U being the usual total ordering on integers. A query Q that is computable on the ordered domain $U = I^+$ is said to be a computable query on the integers.*

We show that the language $D^{a(\mathtt{MAX})+}$ can express all computable queries on integers.

Theorem 5.1. *For the universal domain of the non-negative integers, the language $D^{a(\mathtt{MAX})+}$ is expressive.*

We illustrate the idea behind the proof by outlining the construction of a program in the language $D^{a(\mathtt{MAX})+}$ to evaluate the summarized explosion query *SE* given in Section 3. Since we are dealing with the integer domain, we assume that the relation $\mathtt{subpart}(P, S, C) \subseteq I^3$. Our strategy is as follows:

1. Since we are dealing with integers, we chose the identity function as the encoding function e. Each tuple (X, Y, C) of the subpart relation is then coded as a single integer. The coding is straightforward, and uses the $+$ built-in function. Aggregation is not needed.
2. *Encode the entire relation of integers built in Step 1 into a single integer*. This step is the crucial step in the construction, and is the only one where aggregation is needed. We use the $+$ function to build a set of codes (one for each possible ordering of some or all of the tuples in the relation), and then use the MAX aggregation function in a stratified manner, and over a finitely computable relation, to select *the* codes that encode all the tuples of the subpart relation. *It is essential to discard codes encoding only some of the tuples of the relation at this stage. Keeping all codes leads to an incorrect answer.* Optionally, we can again use the MAX aggregation function in a stratified manner, and over a finitely computable relation, to select *one* of the codes that encode all the tuples of the subpart relation. A $D^{a(\mathtt{MAX})+}$ program to do the encoding and selection of the codes is given Section 5.1.2. So, this stage computes the function g that maps a whole database into a single integer. Note that a similar technique can be used to encode any relation into a single integer using the MAX aggregation function.

3. The query SE is computable, so a partial recursive function g_{SE} that maps the given subpart relation (represented by an integer code) into the answer need relation (again represented by an integer code) exists. We simulate the function g_{SE} using a safe program in the language D^+, without aggregation. *Note that partial recursive functions on integers can be computed in D^+ without aggregation.* The D^+ program scheme to evaluate a given partial recursive function is discussed in Section 5.1.3.
4. Decode the integer code representing the need relation to obtain the answer relation need(P, S, C). The decoding is straightforward, and uses the + built-in function. Aggregation is not needed. This implements g^{-1}.

Since stratified MAX aggregation can be expressed using stratified negation, we also have:

Corollary 5.1. *For the universal domain of non-negative integers, the language $D^{\neg+}$ is expressive.*

Proof. Stratified MAX aggregation can be expressed using stratified negation. The subgoal

(g): GROUPBY(s$(\bar{X}, Y, \bar{Z})$, $[\bar{X}]$, $W =$ MAX(Y))

where $\bar{X}$ and $\bar{Z}$ are vectors of variables, can be replaced by the subgoal $s(\bar{X}, W, U), \neg\text{gs}(\bar{X}, W)$ where U is a variable that does not appear elsewhere in the rule, and the predicate gs is defined as:

gs$(\bar{X}, Y1)$:– s$(\bar{X}, Y1, \bar{Z}1)$ & s$(\bar{X}, Y2, \bar{Z}2)$ & $Y2 > Y1$. ■

Example 5.1. A relation cc_subpart of *candidate codes* is defined in Section 5.1.2 using the MAX aggregation function as follows:

$(P9)$: cc_subpart$^f(C)$:– rc_subpart$^{ff}(M, C)$ &
GROUPBY(rc_subpart$^{ff}(N, F)$, [ ], $M =$ MAX(N)).

We could use stratified negation here instead of MAX, by first defining an auxiliary predicate grc_subpart$(N1)$:

$(P9a)$: grc_subpart$(N1)$:– rc_subpart$(N1, F1)$ &
rc_subpart$(N2, F2)$ & $N2 > N1$.

And then rewriting rule $P9$ as:

$(P9')$: cc_subpart$^f(C)$:– rc_subpart$^{ff}(M, C)$ & rc_subpart$^{ff}(M, C')$ &
$\neg$grc_subpart(M). ■

5.1. $D^{a(\mathtt{MAX})+}$ PROGRAMS SHOWING EXPRESSIVENESS

We give details of the $D^{a(\mathtt{MAX})+}$ program needed to encode relations into unique integers, and the D^+ program needed to evaluate partial recursive functions of integers. The given program is adorned using *bf* adornments to indicate that some of the variables in the head must be bound during a top-down evaluation. A rule in such a program is safe if all the head variables adorned free appear in a positive subgoal or in the output variables of a groupby subgoal in the rule. A magic-sets transformation [12] on this program can be done to add subgoals that restrict the variables adorned bound in the head. We will show how to deal with the aggregation subgoals during a magic-sets transformation to ensure that stratification is retained.

Assume that the built-in function $X + Y = Z$ binds argument Z when given bindings for X and Y. The following adorned arithmetic predicates can then be defined using +: $\mathtt{minus}^{bbf}(X, Y, Z)$ iff $Z = X - Y$ and $Y \leq X$; $\mathtt{times}^{bbf}(X, Y, Z)$ iff $X * Y = Z$; $\mathtt{div}^{bbff}(X, Y, D, R)$ iff $X = Y * D + R$ and $R < Y$; $\mathtt{exp}^{bbf}(X, Y, Z)$ iff $X^Y = Z$; $\mathtt{log}^{bbf}(X, Y, Z)$ iff $Y^Z = X, Y > 1$; $\mathtt{prime}^{b}(X)$ iff X is a prime number; $\mathtt{nextprime}^{bf}(P, Q)$ iff Q is the next prime number after P; and $\mathtt{nthPrime}^{bf}(N, X)$ iff X is the N^{th} prime number. Safe rules defining each of the above adorned predicates, starting from the built-in function +, are given in Appendix A.

5.1.1. *Coding tuples*

Code a $\mathtt{subpart}(P, S, C)$ tuple by the integer $2^P 3^S 5^C$. Note that P, S, and C are integers. We first define $\mathtt{prime3}(X, Y, Z)$ to be the relation with a single tuple containing the first three prime numbers:

$$(P1):\quad \mathtt{prime3}^{fff}(P1, P2, P3) :- \mathtt{nextprime}^{bf}(1, P1)\ \&\ \mathtt{nextprime}^{bf}(P1, P2)\ \&\ \mathtt{nextprime}^{bf}(P2, P3).$$

Tuple-coded (tc) subpart, $\mathtt{tc\_subpart}(D)$, is a relation containing, for each tuple in the subpart relation, its code D.

$$\begin{aligned}(P2):\quad \mathtt{tc\_subpart}^{f}(D) :-\ & \mathtt{subpart}^{fff}(P, S, C)\ \&\\ & \mathtt{prime3}^{fff}(P1, P2, P3)\ \&\\ & \mathtt{exp}^{bbf}(P1, P, D1)\ \&\\ & \mathtt{exp}^{bbf}(P2, S, D2)\ \&\ \mathtt{exp}^{bbf}(P3, C, D3)\ \&\\ & \mathtt{times}^{bbf}(D1, D2, D4)\ \&\ \mathtt{times}^{bbf}(D4, D3, D).\end{aligned}$$

Observe that based on the data in relation $\mathtt{subpart}^{fff}$, we can write magic rules that will evaluate all possible bound arguments of possible calls to *prime*, *exp* and *times* at this stage.

5.1.2. Coding a relation

Let $\texttt{tc\_subpart}(D)$ be a relation of integer codes for each tuple in relation $\texttt{subpart}(P, S, C)$, and let it have tuples $\{d_1, \ldots, d_n\}$. We will code the relation by the integer $(p_1^{d_1} \times p_2^{d_2} \times \ldots \times p_n^{d_n})$, where p_i is the i^{th} prime number. The predicate $\texttt{rc\_subpart}(N, F)$ ($\texttt{rc}$ = relation_coded) holds such codes F for N tuples in $\texttt{subpart}$, one code for each of the $N!$ possible orderings $< d_1, \ldots, d_N >$ of any N tuples in $\texttt{tc\_subpart}(D)$.

We first define the predicate $\texttt{notcoded}^{bbb}(M, X, N)$ to be true when M is not an element of the N element set $\{d_1, \ldots, d_N\}$, where the integer X is seen as a code of the form $\{p_1^{d_1} \times p_2^{d_2} \times \ldots \times p_N^{d_N}\}$.

$(P3)$: $\texttt{notcoded}^{bbb}(M, 1, 0)$.

$(P4)$: $\texttt{notcoded}^{bbb}(M, X, N)$:– $\texttt{nthPrime}^{bf}(N, P)$ &
$\texttt{factor}^{bbf}(P, X, D)$ & $M \neq D$ &
$\texttt{exp}^{bbf}(P, D, Y)$ &
$\texttt{div}^{bbff}(X, Y, X', 0)$ & $\texttt{minus}^{bbf}(N, 1, N')$ &
$\texttt{notcoded}^{bbb}(M, X', N')$.

The goal $\texttt{nthPrime}^{bf}(N, P)$ is true if P is the N^{th} prime number. We now define the predicate $\texttt{factor}^{bbf}(P, X, D)$, which is true if the prime factorization of X includes P^D.

$(P4)$: $\texttt{factor}^{bbf}(P, 1, 0)$.

$(P5)$: $\texttt{factor}^{bbf}(P, X, 0)$:– $\texttt{div}^{bbff}(X, P, X', R)$ & $R \neq 0$.

$(P6)$: $\texttt{factor}^{bbf}(P, X, D)$:– $X \neq 0$ & $\texttt{div}^{bbff}(X, P, X', 0)$ &
$\texttt{factor}^{bbf}(P, X', D')$ & $D = D' + 1$.

The relation $\texttt{rc\_subpart}(N, F)$, where F is a code for some permutation of N tuples in the $\texttt{tc\_subpart}$ relation, is built as follows:

$(P7)$: $\texttt{rc\_subpart}^{ff}(0, 1)$.

$(P8)$: $\texttt{rc\_subpart}^{ff}(N, F)$:– $\texttt{rc\_subpart}^{ff}(N', F')$ &
$\texttt{tc\_subpart}^{ff}(D)$ &
$\texttt{notcoded}^{bbb}(D, F', N')$ &
$N = N' + 1$ & $\texttt{nthPrime}^{bf}(N, P)$ &
$\texttt{exp}^{bbf}(P, D, C)$ & $\texttt{times}^{bbf}(F', C, F)$.

Rule $P7$ states that the code for zero tuples of a relation is 1. Rule $P8$ constructs the code for N tuples by extending the code F' for $N - 1$ tuples by selecting another tuple that has not been coded in F'.

If relation tc_subpart has n tuples, there will be tuples of the form rc_subpart(n, F), but no tuples of the form rc_subpart($n+1, G$). Any F value in a tuple of rc_subpart(n, F) can be used as a code for the subpart relation. We call such F values *candidate codes*, defined in the relation cc_subpart.

($P9$): cc_subpart$^f(C)$:– rc_subpart$^{ff}(M, C)$ &
GROUPBY(rc_subpart$^{ff}(N, F)$, [], M = MAX(N)).

Rule $P9$ is the crucial (and only) step where stratified MAX aggregation is needed for correctness. If we do not discard the codes encoding $n-1$ or fewer tuples, we would do the subsequent computation on all subgraphs of the given graph. Consequently, the answer need relation will be the union of the need relations from each subgraph, which is not equal to the need relation for the entire graph. Also note that the rc_subpart relation is finitely computable, so the aggregation is over a finitely computable relation.

We observe that the cc_subpart$^f(C)$ relation is a subset of the second argument of rc_subpart$^{ff}(M, C)$. To ensure stratification of the magic-sets transformed program, we must simply substitute all occurrences of cc_subpart$^f(C)$ in any magic rule by rc_subpart$^{ff}(M, C)$.

Subsequent computation can be done using the set of all codes encoding all the n tuples. However, since we have the MAX operation, we can optimize further computation by selecting exactly one of the codes encoding all n tuples. The selected best code, bc_subpart(C) may be obtained by:

($P10$): bc_subpart$^f(C)$:– GROUPBY(cc_subpart$^f(F)$, [], C = MAX(F)).

We again observe that the bc_subpart$^f(C)$ relation is a subset of the cc_subpart$^f(C)$ relation. To ensure stratification of the magic-sets transformed program, we must simply substitute all occurrences of bc_subpart$^f(C)$ in any magic rule by cc_subpart$^f(C)$, and hence by rc_subpart$^{ff}(M, C)$.

5.1.3. Evaluating partial recursive functions using safe D^{a+}

Given an arbitrary partial recursive function $g(X_1, \ldots, X_n) = Y$, Lloyd ([11], Page 53) constructs a program for the predicate $p_g^{b \ldots bf}(X_1, \ldots, X_n, Y)$ by induction on the number of applications of composition, primitive recursion, and minimalizations needed to define the partial recursive function $g(X_1, \ldots, X_n)$. Aggregation or negation is not needed in the construction.

Given the predicate $p_{g_{SE}}^{bf}(C, D)$ representing the partial recursive function g_{SE} that evaluates query SE, the relation rc_need(D) is given by the rule:

($P11$): rc_need$^f(D)$:– bc_subpart$^f(C)$ & $p_{g_{SE}}^{bf}(C, D)$.

rc_need contains the integer code D for the entire need relation.

5.1.4. Decoding the answer to query SE using $D^{a(\texttt{MAX})+)}$

Having obtained the integer code D for the entire need relation, we must now decode D to derive the tuples in the need relation. D is of the form $\{p_1^{d_1}p_2^{d_2},\ldots,p_N^{d_N}\}$, where p_i is the i^{th} prime number, d_i is the tuple code for some tuple in the need relation, and N is the number of tuples in the need relation. We define a $\texttt{tci\_need}^{ff}(F,M)$ relation that contains the tuple (d_i,i), $1 \leq i \leq N$. The $\texttt{tc\_need}^{f}(F)$ relation of tuple codes of the need relation is then obtained by a projection of the $\texttt{tci\_need}^{ff}(F,M)$ relation.

$(P12)$ $\texttt{tci\_need}^{ff}(F,1)$:– $\texttt{rc\_need}^{f}(D)$ & $\texttt{nthPrime}^{bf}(1,P)$ & $\texttt{factor}^{bbf}(P,D,F)$.

$$\begin{aligned}(P13):\quad \texttt{tci\_need}^{ff}(F,N) :-\ & \texttt{tci\_need}^{ff}(F',N')\ \&\\ & N = N'+1\ \&\\ & \texttt{rc\_need}^{f}(D)\ \&\\ & \texttt{nthPrime}^{bf}(N,P)\ \&\\ & \texttt{factor}^{bbf}(P,D,F).\end{aligned}$$

$(P14)$: $\texttt{tc\_need}^{f}(F)$:– $\texttt{tci\_need}^{ff}(F,N)$.

The first two subgoals in rule $P13$ are required only for safety. We assume that the partial recursive function g_{SE} was written such that each tuple need(P,S,C) was coded as $2^P3^S5^C$. Consequently, the need relation is derived as follows:

$$\begin{aligned}(P15):\quad \texttt{need}^{fff}(P,S,C) :-\ & \texttt{tc\_need}^{f}(F)\ \&\\ & \texttt{factor}^{bbf}(2,F,P)\ \&\\ & \texttt{factor}^{bbf}(3,F,S)\ \&\\ & \texttt{factor}^{bbf}(5,F,C).\end{aligned}$$

5.2. COMPLETENESS OF $D^{l(a(\texttt{MAX}))}$

The only aggregation operation needed to obtain expressiveness is the MAX computation on the relation rc_subpart(N,F), where F is a code for some permutation of N tuples in the subpart relation. Given an arbitrary database, it is easy to see that N will be bounded by the total number of tuples in the database, a finite number. Thus the relation rc_subpart is finitely computable. The same technique can be used to encode any finite edb over integers into a single integer, using only the MAX aggregation function over finitely computable relations. Thereby, by Theorems 4.2 and 5.1 we obtain:

Corollary 5.2. *Let $D^{l(a(\texttt{MAX}))+}$ be Datalog extended with arithmetic + and stratified* MAX *aggregation, such that every aggregated predicate is finitely computable. Then, for the universal domain of the non-negative integers, the language $D^{l(a(\texttt{MAX}))+}$ is complete.*

6. Completeness on ordered uninterpreted domains

We assumed that the subpart(P, S, C) relation in the summarized explosion query was on integers. More realistically, the part names come from the domain S of strings. The domain of strings is also countably infinite, and can be totally ordered by lexicographic order.

Thus the summarized explosion query is typically expressed over two typed domains, with each attribute in the input and output relations typed over one of these domains. In general, a query can be over multiple ordered domains. Definition 5.2 defines computable queries on a single ordered universal domain. We could extend the definition over multiple ordered domains, as we do in Section 8 for queries over multiple ordered or unordered domains. With such an extension, we can show that the summarized explosion query over two ordered domains (the domain of strings and the domain of integers) can be expressed in languages $D^{l(a)+}$ and $D^{v(\neg)+}$.

In this section we limit ourselves to the case of a universal domain consisting of a *single* atomic ordered domain. We further permit the computation domain to contain non-negative integers. We illustrate, with an example, that any ordered domain can be mapped onto integers, so a query over any ordered domain can be mapped into a query over the integer domain. Stratified COUNT over finitely computable relations, or stratified negation over verifiable relations are used to do the mapping. We can then lift our results from Section 5 to derive the expressiveness of $D^{a(\text{COUNT},\text{MAX})+}$ or $D^{\neg+}$ over ordered domains.

Theorem 6.1. *Given an extensional database on an atomic ordered domain, the languages* $D^{v(\neg)+}$ *and* $D^{l(a(\text{COUNT},\text{MAX}))+}$ *are expressive.*

The languages $D^{l(a(\text{COUNT},\text{MAX}))+}$ and $D^{v(\neg)+}$ are also bounded (*cf.* Theorems 4.2 and 4.1), so Theorem 6.1 establishes completeness.

As a proof outline, we show how to map an ordered domain to the integer domain using $D^{l(a(\text{COUNT}))+}$ or $D^{v(\neg)+}$. Consider a subpart(P, S) relation over an ordered domain of strings. Let node(X) be a unary relation of constants used in the subpart(P, S) relation:

$(T1)$: node(P) :– subpart(P, S).

$(T2)$: node(S) :– subpart(P, S).

Using stratified and finitely computable COUNT. For each X value in the edb, collect all values less than it. Since the domain is ordered, we can use the *less than* built-in predicate.

$(T3)$: less(X, Y) :– node(X) & node(Y) & $Y < X$.

Note that the relation less(X, Y) is finitely computable. For each X value, count the number of domain values less than X, say it is N. The value X in the ordered domain is then mapped to the integer N.

$(T4)$: map(X,N) :– node(X) & GROUPBY(less(X, Y), $[X]$, $N =$ COUNT(Y)).

Any edb predicate can now be mapped to an integer equivalent. For example, we can derive int_subpart(IP, IS) as follows:

$(T5)$: int_subpart(IP, IS) :– subpart(P, S) & map(P, IP) & map(S, IS).

Using safe stratified verifiable negation. Define existlessthan(X) to be true if X is not the least element in the node relation, and between(W, X) to be true if there exists a node element between W and X.

$(T6)$: existlessthan(X) :– node(X) & node(Y) & $Y < X$.

$(T7)$: between(W,X) :– node(W) & node(Y) & node(X) & $W < Y$ & $Y < X$.

Both relations existlessthan and between are verifiable (indeed, they are finitely computable), so negation on them is permitted in the language $D^{v(\neg)+}$. Now we map the least element to code 0. Every other element is mapped to a code 1 more than the element just below it in the ordering.

$(T8)$: map$(X,0)$:– node(X) & ¬existlessthan(X).

$(T9)$: map(X,N) :– node(X) & map(W,N') & $W < X$ & $N = N' + 1$ &
¬between(W,X).

7. Completeness on unordered domains

We now consider universal domains that are unordered and atomic (no function symbols). Input and output relations are over this universal domain. We will allow the computation domain to include function symbols. Function symbols can be used to construct terms that can be interpreted as non-negative integers.

We have shown that the languages $D^{l(a(\mathtt{COUNT},\mathtt{MAX}))+}$ and $D^{v(\neg)+}$ can express any computable query on ordered domains. We need a version of MAX which operates on a set of terms of the form $f^i(0) = f(f(\ldots(0)\ldots))$ which represents the integer i, and obtains the "largest" one. For convenience we shall call this new operator MAX as well.

Theorem 7.1. *The languages $D^{l(a(\mathtt{MAX}))f}$ and $D^{v(\neg)f}$ are expressive for all atomic domains.*

Languages $D^{l(a(\texttt{MAX}))f}$ and $D^{v(\neg)f}$ are also bounded (Theorems 4.1 and 4.2), so Theorem 7.1 establishes completeness.

The proof sketch for expressiveness is as follows. We assume that the universal domain does not include functional terms. Thus, the edb and the output relations are limited to atomic terms.

Given a computable query Q, let there be n domain elements in the edb and constant relations. We start by showing that it is possible to obtain the set of all possible encodings from the n domain elements in the edb and constant relations to the set $I^{|n|}$ of the first n non-negative integers. Stratified `MAX` aggregation, the list functor, and a successor function are used to build the encodings. Thus, the integer i is represented by the term $f^i(0)$, where f is the unary function symbol (successor). Then, for each of these encodings, we can code the database and constant relations into a single integer (using stratified `MAX` aggregation over a finitely computable relation to select one of the many codes that would be generated). For each encoding, the partial recursive function for the query can be evaluated on the database, and the result decoded to get the output relation. Since the query is computable, the decoded output of the partial recursive function must be the same for all encodings. Finally, the encoding argument can be projected out to get the output relation. The details follow.

7.1. ENCODING DOMAIN ELEMENTS

1. Construct the `node` relation of all domain elements in the edb and constant relations. The `node` relation is built by projection of edb relations, and by adding facts for each constant used in the constant relations of the query.
2. Construct a list of all nodes, using the binary function symbol as a list constructor. The predicate $\texttt{node\_list}(L, N)$ can be defined to be true for each list L of some permutation of N nodes. This is done as follows:

   $(T1)$: $\texttt{notin}^{bb}(X, [\,])$.

   $(T2)$: $\texttt{notin}^{bb}(X, [Y|L])$:– $X \neq Y$ & $\texttt{notin}^{bb}(X, L)$.

   $(T3)$: $\texttt{node\_list}^{ff}([\,], 0)$.

   $(T4)$: $\texttt{node\_list}^{ff}([X|L], f(N))$:– $\texttt{node\_list}(L, N)$ & $\texttt{node}^{f}(X)$ & $\texttt{notin}^{bb}(X, L)$.

   The `node_list` is finitely computable (and hence verifiable), so aggregation and negation on `node_list` is permitted in languages $D^{l(a(\texttt{MAX}))f}$ and $D^{v(\neg)f}$.
3. Select the *candidate lists* that code all the nodes. We can use safe stratified verifiable negation or stratified and finitely computable aggregation (`MAX`) to determine the cardinality (`card`) of the domain. Candidate lists are

lists of size card.

$(T5)$: $\texttt{card}^f(N)$:– $\texttt{node\_list}^{ff}(L1, N)$ &
$\texttt{GROUPBY}(\texttt{node\_list}^{ff}(L2, M), [], N = \texttt{MAX}(M))$.

$(T6)$: $\texttt{clist}^f(CL)$:– $\texttt{card}^f(N)$ & $\texttt{node\_list}^{ff}(CL, N)$.

Each candidate list CL represents a permutation of the domain elements of interest.

4. For each candidate list clist(CL), construct a map relation map(CL, X, N) from nodes to integers, meaning that X is at the N^{th} position (starting from 0) in list CL and is thus mapped to integer N. We use an auxiliary relation strip_list(L, N, Z, M) that associates each element Z of list L to an integer M starting from integer N for the first element, integer $N + 1$ for the second element, and going on to the integer $N + |L| - 1$ for the last element.

$(T9)$: $\texttt{map}^{bff}(CL, X, M)$:– $\texttt{strip\_list}^{bbff}(CL, 0, X, M)$.

$(T10)$: $\texttt{strip\_list}^{bbff}([X], N, X, N)$.

$(T11)$: $\texttt{strip\_list}^{bbff}([X|[Y|L]], N, X, N)$.

$(T12)$: $\texttt{strip\_list}^{bbff}([X|[Y|L]], N, Z, M)$
:– $\texttt{strip\_list}^{bbff}([Y|L], f(N), Z, M)$.

The map relation now holds the set of all encodings from domain elements of interest to integers less than card (the cardinality).

7.2. CODING THE DATABASE

1. For each candidate list, code the edb into an equivalent edb on integers. Each relation in the new edb on integers will have one extra attribute containing the candidate list with respect to which the coding is done. The function symbol f is used as the successor function to implement $+$. The rules used are similar to those for the integer domain (Section 5.1.1).
2. For each candidate list, code each edb tuple by a single integer. The rules used are similar to those for the integer domain (Section 5.1.1).
3. For each candidate list, code each edb relation by a single integer (represented as a functional term), using the technique of Section 5.1.2. This step requires stratified (and finitely computable) aggregation (or stratified verifiable negation) to first select the codes that encode all tuples of the edb relation, and then to select the largest of such codes.
4. For each candidate list, code the full database by a single integer, using

lexicographic ordering between edb relations. The result will be the relation code_db(CL, CC), meaning that CC is a code for the database under the ordering implied by the candidate list CL of the domain elements.

Note that at this point we may not be able to select one code for the database using `MAX` since there may be several candidate lists that result in the same encoding of the database.

7.3. EVALUATING AND DECODING THE PARTIAL RECURSIVE FUNCTION

1. For each candidate list, evaluate the partial recursive function f_Q that computes query Q on the integer codes for the database, deriving result codes for the answer relation. If $p_f(I_1, \ldots I_n, O)$ is the predicate, and P the program, evaluating the partial recursive function f_Q, our program will augment each idb predicate in program P with an additional attribute for the candidate lists. The rules used are similar to those for the integer domain (Section 5.1.3).
2. For each candidate list, decode the result code to get an integer representation of the result relation, one representation for each candidate list (Section 5.1.4).
3. For each candidate list, use the `map` relation to derive the result relation on the uninterpreted domain, with one additional argument for the candidate list in each relation.
4. Project out the candidate list argument to derive the final result relation.

By definition of computable queries, the result relation for each candidate list must be identical. So, the projection simply removes duplicates.

8. Multiple domains

Given universal domains $U_1, \ldots U_k$, where each U_i may be countably infinite, a relation R of type $a = (a1, \ldots, al)$ is a finite subset of $U_{a1} \times \ldots \times U_{al}$.

Definition 8.1. Typed computable queries: *Let $U_1, \ldots, U_k$, be a given set of finite or countably infinite, ordered or unordered, universal domains. Let Q be a partial function from a database $B = R_1, \ldots, R_m$ of type $\bar{a} = a_1, \ldots, a_m$, where each $R_i \subseteq (U_{a_{i1}} \times \ldots \times U_{a_{il_i}})$ to the output relation $R_o \subseteq (U_{a_{o1}} \times \ldots \times U_{a_{ol_o}})$.*

The query Q is a typed computable query if there exists a partial recursive function f from I^{m+1} to I, and there exists a finite set of constant queries, $Q_1, \ldots, Q_n$ such that, for all databases B of type $\bar{a}$, and for all order-preserving encodings $e = (e_1, \ldots, e_k)$ from $(U_1, \ldots, U_k)$ to $(I, \ldots, I)$, the function f maps $(g(e(B))$, $g(e(Q_1)), \ldots, g(e(Q_m)))$ onto the integer $g(e(R_o))$.

If Q is undefined on B, then $f(g(e(B)))$ must be undefined for all encodings e.

(Note that Definition 8.1 differs from typed queries of [3]. Typed queries of [3] are referred to as CH-Computable queries in this paper.)

Theorem 8.1. *The languages $D^{l(a)f}$ and $D^{v(\neg)f}$ are complete for typed computable queries.*

Proof. Expressiveness: We need to show that the encoding over each universal domain can be done using the languages $D^{l(a)f}$ and $D^{v(\neg)f}$. Start by projecting out elements of each domain used in the edb onto a relation U_i. Then, encode each domain separately onto integers. Decoding of an attribute uses its type information to select the correct decoding function.
Boundedness: The proof is similar to the proofs for Theorems 4.1 and 4.2 in Section 4. ∎

9. Related work

All computability definitions in this paper are alternative formulations of definitions given by Chandra and Harel [3]. Chandra and Harel also define a language QL that they prove to be complete. QL is a procedural language with a *while* looping construct. In contrast, $D^{l(a)f}$ and $D^{v(\neg)f}$ are declarative complete languages.

Apt and Blair [2] show that the language $D^{\neg+}$ is not bounded. We show that the language D^{a+} is also not bounded. As discussed in Section 4, [2] presents restrictions of $D^{\neg+}$ that are bounded.

Abiteboul and Vianu [1] study relational queries by introducing a new Turing Machine model in which the machine is equipped with a *relational store* that can be manipulated using first order formulas. They present two versions of the machine: one in which all interactions with the store are through an interface, and a second that allows to move relations onto the Turing Machine tapes of a collection of machines operating in parallel (one tuple on each machine). They show that while the second version is complete the first one is not (on unordered domains). They also use the first version to classify the complexities of *Fixpoint* and *while* queries (the two are the same iff PTIME = PSPACE). Their work thus focuses on complexity issues; our work concentrates on analyzing the power of aggregation constructs added to Datalog.

Schlipf [16] characterizes the complexity class of relations that can be expressed using logic programs with arbitrary negation, treated under three possible negation semantics. The paper derives results about different classes of programs for which a particular negation semantics gives polynomial time or Π_1^1 computability. For example, over finite edb's, three valued program completion and well founded semantics are shown to express uniformly computable relations. For ordered domains, that translates to PTIME computable queries. The paper also relates expressivity of different types of semantics. We consider only stratified programs, over which all the semantics considered by [16] coincide. We consider the question of whether the language under consideration can express a computable query. We do not consider complexity issues.

Hull and Su [8] consider an algebra and a calculus augmented with the *while* looping construct, and characterize the augmented algebra as complete on untyped sets, while the augmented calculus can express non-computable queries.

10. Conclusions and future work

We have defined declarative query languages with stratified aggregation that are expressive and bounded for computable queries.

We have explored the expressive power of Datalog (a declarative query language) in the presence of stratified aggregation. Stratified aggregation by itself is not powerful enough to express path aggregation queries. However when arithmetic built-in predicates are added, the resulting language is expressive for all computable queries on the non-negative integers and other ordered domains. With function symbols, the language is expressive on all domains (ordered and unordered). `COUNT` and `MAX` are the primitive aggregation functions needed to obtain expressiveness. We also show that the languages can be made complete by restricting the aggregation to be over predicates that are finitely computable.

Safe stratified negation and stratified aggregation have the same expressive power when used in conjunction with arithmetic or function symbols. Thus Datalog extended with safe stratified negation and function symbols is also an expressive query language.

The expressivity of aggregation in conjunction with arithmetic or function symbols means that there is no reason to consider non-stratified aggregation due to lack of expressive power.

Several issues need to be explored further. One is the incorporation of function symbols into the input and output domains. Another interesting question is how should nondeterministic queries [10, 6] be defined and computed. Certain types of non-determinism can be introduced into D^{af} by allowing different encodings to lead to different answers. Each permutation of domain elements thus represents a possible choice.

Acknowledgements

We thank the anonymous referees for comments that contributed greatly to the quality of this paper. We also thank Catriel Beeri, S. Sudarshan, and Allen Van Gelder for comments and suggestions.

Appendix A: Arithmetic predicates

Given the built-in function $X + Y = Z$ that binds argument Z when given bindings for X and Y, we define various adorned arithmetic predicates. Each predicate must be invoked using an adornment that is as bound or more bound than the one indicated.

Each rule of the programs for the adorned predicates has the property that each head variable with an f adornment appears in the rule body. This ensures that if the adorned predicate is invoked after a binding for all b adorned arguments is obtained, the program will compute finite relations.

- $\texttt{minus}^{bbf}(X, Y, Z)$ iff $Z = X - Y$ and $Y \le X$.

  $\texttt{minus}^{bbf}(X, Y, Z)$:– $\texttt{sum}^{bbff}(X, Y, Z, X)$.

  $\texttt{sum}^{bbff}(X, Y, 0, Y)$:– $Y \le X$.
  $\texttt{sum}^{bbff}(X, Y, Z, X')$:– $\texttt{sum}^{bbff}(X, Y, Z', X'')$ &
  $X'' < X$ & $Z = Z' + 1$ & $X' = X'' + 1$.

  The auxiliary predicate $\texttt{sum}^{bbff}(X, Y, Z, W)$ is defined to have the following properties: (1) $Y + Z = W$, (2) $Y \le X$, and (3) $W \le X$.

- $\texttt{times}^{bbf}(X, Y, Z)$ iff $X * Y = Z$.

  $\texttt{times}^{bbf}(X, 0, 0)$.
  $\texttt{times}^{bbf}(X, Y, Z)$:– $\texttt{minus}^{bbf}(Y, 1, Y')$ & $\texttt{times}^{bbf}(X, Y', Z')$ &
  $Z = X + Z'$.

- $\texttt{div}^{bbff}(X, Y, D, R)$ iff $X = Y * D + R, R < Y$.

  $\texttt{div}^{bbff}(X, Y, 0, X)$:– $X < Y$.
  $\texttt{div}^{bbff}(X, Y, D, R)$:– $\texttt{minus}^{bbf}(X, Y, X')$ & $\texttt{div}^{bbff}(X', Y, D', R)$ &
  $D = D' + 1$.

- $\texttt{exp}^{bbf}(X, Y, Z)$ iff $X^Y = Z, LOG_Y X = Z$.

  $\texttt{exp}^{bbf}(X, 0, 1)$.
  $\texttt{exp}^{bbf}(X, Y, Z)$:– $\texttt{minus}^{bbf}(Y, 1, Y')$ & $\texttt{exp}^{bbf}(X, Y', Z')$ &
  $\texttt{times}^{bbf}(Z', X, Z)$.

- $\texttt{log}^{bbf}(X, Y, Z)$ iff $Y^Z = X, Y > 1$.

  $\texttt{log}^{bbf}(X, Y, Z)$:– $\texttt{goodlog}^{bbff}(X, Y, Z, X)$.

  $\texttt{goodlog}^{bbff}(X, Y, 0, 1)$:– $Y > 1$ & $X > 0$.

  $\texttt{goodlog}^{bbff}(X, Y, Z, X')$:– $\texttt{goodlog}^{bbff}(X, Y, Z', X'')$ &
  $X'' < X$ & $Z = Z' + 1$ &
  $\texttt{times}(X'', Y, X')$.

  The auxiliary predicate $\texttt{goodlog}^{bbff}(X, Y, Z, W)$ is defined to have the following properties: (1) $Y^Z = W$, (2) $Y > 1$, (3) $X > 0$, and (4) $W < X * Y$.

- $\texttt{notprime}^{b}(X)$ iff X is not a prime number.

$\texttt{notprime}^{b}(0).$
$\texttt{notprime}^{b}(1).$
$\texttt{notprime}^{b}(X) \text{ :- } \texttt{divides}^{bff}(X, Y, 1) \ \& \ Y < X.$

$\texttt{divides}^{bff}(X, Y, 1)$ is true for the smallest $Y > 1$ that divides X.
$\texttt{divides}^{bff}(X, Y, 0)$ is true for all Y less than the smallest divisor (besides 1) of X.

$\texttt{divides}^{bff}(X, 1, 0).$

$$\begin{aligned}\texttt{divides}^{bff}(X, Y, 1) \text{ :- } & \texttt{divides}^{bff}(X, Y', 0) \ \& \\ & Y = Y' + 1 \ \& \ Y \leq X \ \& \\ & \texttt{div}^{bbff}(X, Y, D, R) \ \& \ R = 0.\end{aligned}$$

$$\begin{aligned}\texttt{divides}^{bff}(X, Y, 0) \text{ :- } & \texttt{divides}^{bff}(X, Y', 0) \ \& \\ & Y = Y' + 1 \ \& \ Y < X \ \& \\ & \texttt{div}^{bbff}(X, Y, D, R) \ \& \ R \neq 0.\end{aligned}$$

- $\texttt{prime}^{b}(X)$ iff X is a prime number.

  $\texttt{prime}^{b}(X) \text{ :- } \texttt{divides}^{bff}(X, X, 1).$

- $\texttt{nextprime}^{bf}(P, Q)$ iff Q is the next prime number after P.

  $\texttt{nextprime}^{bf}(P, Q) \text{ :- } \texttt{goodnextint}^{bff}(P, Q, 1).$

  $\texttt{goodnextint}^{bff}(P, Q, 0) \text{ :- } Q = P + 1 \ \& \ \texttt{notprime}^{b}(Q).$
  $\texttt{goodnextint}^{bff}(P, Q, 1) \text{ :- } Q = P + 1 \ \& \ \texttt{prime}^{b}(Q).$

  $$\begin{aligned}\texttt{goodnextint}^{bff}(P, Q, 0) \text{ :- } & \texttt{goodnextint}^{bff}(P, Q', 0) \ \& \\ & Q = Q' + 1 \ \& \ \texttt{notprime}^{b}(Q).\\ \texttt{goodnextint}^{bff}(P, Q, 1) \text{ :- } & \texttt{goodnextint}^{bff}(P, Q', 0) \ \& \\ & Q = Q' + 1 \ \& \ \texttt{prime}^{b}(Q).\end{aligned}$$

- $\texttt{nthPrime}^{bf}(N, X)$ iff X is the N^{th} prime number.

  $\texttt{nthPrime}^{bf}(1, 2).$

  $$\begin{aligned}\texttt{nthPrime}^{bf}(N, X) \text{ :- } & \texttt{minus}^{bbf}(N, 1, N') \ \& \ \texttt{nthPrime}^{bf}(N', X') \ \& \\ & \texttt{nextprime}^{bf}(X', X).\end{aligned}$$

References

[1] S. Abiteboul and V. Vianu, Generic computation and its complexity, *Proceedings of the Twenty-Third Symposium on Theory of Computing*, New Orleans, LA, 1991, pp. 154–163.

[2] K.R. Apt and H.A. Blair, Arithmetic classification of perfect models of stratified programs, in R.A. Kowalski and K.A. Bowen, editors, *Proceedings of the Fifth International Conference and Symposium on Logic Programming*, Seattle, WA (MIT Press, 1988) pp. 765–779.

[3] A.K. Chandra and D. Harel, Computable queries for relational data bases, J. Comp. and Syst. Sci. 21(2) (1980) 156–178.

[4] M.P. Consens and A.O. Mendelzon, Low complexity aggregation in Graphlog and Datalog, *Proceedings of the International Conference on Database Theory (ICDT)*, 1990.

[5] S. Ganguly, S. Greco, and C. Zaniolo, Minimum and maximum predicates in logic programming, *Proceedings of the Tenth Symposium on Principles of Database Systems (PODS)*, Denver, CO, 1991, pp. 154–163.

[6] F. Giannotti, D. Pedreschi, D. Sacca, and C. Zaniolo, Non-determinism in deductive databases, in C. Delobel, M. Kifer, and Y. Masunaga, editors, *Proceedings of the Second International Conference on Deductive and Object-Oriented Databases (DOOD '91)*, LNCS 566, Munich, Germany, 1991, pp. 129–146.

[7] J.E. Hopcroft and J.D. Ullman, *Introduction to Automata Theory, Languages, and Computation* (Addison-Wesley, 1979).

[8] R. Hull and J. Su, Untyped sets, invention, and computable queries, *Proceedings of the Eighth Symposium on Principles of Database Systems (PODS)*, Philadelphia, PA, 1989, pp. 347–359.

[9] D.B. Kemp and P.J. Stuckey, Semantics of logic programs with aggregates, *Proceedings of the 1991 Symposium on Logic Programming*, 1991, pp. 387–404.

[10] R. Krishnamurthy and S.A. Naqvi, Non-deterministic choice in datalog, *Proceedings of the 3rd International Conference on Data and Knowledge Bases*, 1988, pp. 416–424.

[11] J.W. Lloyd, *Foundations of Logic Programming* (Springer-Verlag, 1984).

[12] I.S. Mumick, Query optimization in deductive and relational databases, PhD Thesis, Stanford University, Stanford, CA 94305, USA (1991). Technical Report No. STAN-CS-91-1400. Also available from University Microfilms International, 300 N. Zeeb Road, Ann Arbor, MI 48106. (313)761-4700.

[13] I.S. Mumick, H. Pirahesh, and R. Ramakrishnan, The magic of duplicates and aggregates, *Proceedings of the Sixteenth International Conference on Very Large Databases (VLDB)*, Brisbane, Australia, 1990, pp. 264–277.

[14] I.S. Mumick, J.D. Ullman, and M. Vardi, Personal communications on summarized explosion query (1989).

[15] K.A. Ross and Y. Sagiv, Monotonic aggregation in deductive databases, *Proceedings of the Eleventh Symposium on Principles of Database Systems (PODS)*, San Diego, CA (1992).

[16] J.S. Schlipf, The expressive powers of the logic programming semantics, *Proceedings of the Ninth Symposium on Principles of Database Systems (PODS)*, Nashville, TN, 1990, pp. 196–204.

[17] S. Sudarshan, D. Srivastava, and R. Ramakrishnan, Extending the well-founded and valid semantics for aggregation, Submitted for publication (1992).

[18] S.-Å. Tärnlund, Horn clause computability, BIT 17(2) (1977) 215–226.

[19] J.D. Ullman, Implementation of logical query languages for databases, ACM Trans. Database Syst. 10(3) (1985) 289–321.

[20] J.D. Ullman, *Principles of Database and Knowledge-Base Systems*, Vol. 1 (Computer Science Press, 1988).

[21] A. Van Gelder, The well-founded semantics of aggregation, *Proceedings of the Eleventh Symposium on Principles of Database Systems (PODS)*, San Diego, CA, 1992, pp. 127–138.