The Magic of Duplicates and Aggregates

Presented at VLDB 1990

Inderpal Singh Mumick*
Computer Science Department
Stanford University
Stanford, CA 94305, USA.
mumick@cs.stanford.edu

Hamid Pirahesh
IBM Almaden Research Center
650 Harry Road, K55/801
San Jose, CA 95120, USA.
pirahesh@ibm.com

Raghu Ramakrishnan†
Computer Science Department
University of Wisconsin at Madison
Madison, WI 53706, USA.
raghu@cs.wisc.edu

Abstract

We present a formal treatment of multisets (that arise when duplicates are not eliminated) and aggregate operators for deductive and relational databases. We define the semantics rigorously and extend the magic-sets technique to programs containing multisets and aggregates. The work presented here is an important step in demonstrating the applicability of the magic-sets technique for optimizing queries in commercial query languages such as SQL.

1 Introduction

Previous treatments of Datalog and proposed extensions have treated a program as a collection of definitions of *sets* of facts (tuples). On the other hand, commercial query languages such as SQL typically support the definition of sets and *multisets* of tuples, and provide aggregate operators such as SUM and COUNT over sets and multisets. The ability to deal with multisets has significance from both the standpoint of a user and an implementer. For the former, multisets often provide the more natural semantics; for the latter, computing with multisets — possibly in intermediate stages of a computation — may be the more efficient alternative, and a rigorous semantics for multiset definitions offers a sound theoretical basis for certain program optimizations. Aggregate operators offer limited second-order capabilities, and their utility is widely recognized.

Our contributions are in two areas. First, we provide a simple and intuitive semantics for logic programs containing predicates whose extensions are multisets, and show that this semantics can be supported efficiently. Second, we consider how the aggregate functions and the **group-by** construct of SQL can be introduced into recursive programs. Here again, we give an intuitive semantics, and show that the semantics can be computed efficiently. We provide an overview of our work in the following subsections.

Our work contributes to the definition and development of relational systems also. Duplicates and Aggregates have been present in relational systems from the days of System R ([?]), but their semantics has never been defined formally. A formal semantics is required for a precise language definition, more so after the introduction of recursion into relational systems and with the increasing importance of query rewrite optimization ([?]).

1.1 Duplicates[1] (Multisets)

The work on duplicates has three parts:

1. The central property of a "declarative" semantics is that it allows programs to be understood intuitively, without reference to how they are to be evaluated. Using proof-theoretic notions, in contrast to the usual model-theoretic approach, we develop a formal semantics that enjoys this important property. Models cannot describe multiset semantics

*Part of this work was done at the IBM Almaden Research Center. Work at Stanford was supported by an NSF grant IRI-87-22886, an Air Force grant AFOSR-88-0266, and a grant of IBM Corporation.

†Part of this work was done while the author was visiting IBM Almaden Research Center. Work at Wisconsin was supported by an IBM Faculty Development Award and an NSF grant IRI-88-04319.

[1] Here, and in the rest of the paper, we use duplicates to mean *multiple copies*, not necessarily *two* copies.

since a model is a *set* of atoms.

2. While a declarative semantics is necessary to provide the user with an intuitive, non-operational query language, efficient evaluation techniques must rest upon an equivalent operational semantics. We show that a straightforward extension of Semi-Naive — but not Naive! — fixpoint evaluation provides an equivalent operational semantics.

3. An important contribution is the adaptation of the magic-sets approach ([?, ?]) to work on such extended programs, thereby outlining an efficient computation method for realizing the declarative semantics.

Our treatment of duplicates and multisets can be understood intuitively as follows: If duplicates are not eliminated in evaluating rules defining a predicate, say p, then p may have several occurrences of a given tuple, say t. Thus, p is a multiset of tuples, rather than a set. The corresponding set of tuples in p is easily understood by ignoring duplicates; the subtle part is in understanding the cardinality of each tuple in the multiset p. Intuitively, a tuple t appears as often as it is derived. The number of times that a tuple is derived in SQL is not arbitrary: it is derived exactly once for each *derivation tree* that supports it. This intuition can be formalized for recursive programs as well, as we will demonstrate.

We use a device called **coloring** to distinguish different occurrences of a tuple in a multiset. Does this mean that we support some form of object ids? The answer is that we do not: Coloring is merely a technique that allows us to make the semantics precise, and to explain it intuitively. The formal semantics rests primarily upon the number of occurrences of elements in a multiset; different occurrences of a given element are indistinguishable. Note that coloring is not visible to either the user or to the implementer. Indeed, implementation of multisets is particularly efficient, since we simply omit duplicate checks. Several systems often do this for the sake of efficiency, even when the multiset semantics is not required. Our results provide a formal basis for reasoning about optimization techniques when multisets are generated as intermediate relations, independently of whether the user desires a multiset semantics.

We consider range-restricted programs in this paper, but provide a brief summary of how these results may be extended to non-range-restricted programs.

1.2 Aggregates and the Group-By Construct

Our second contribution is to consider the use of aggregate operators and the **group-by** construct of SQL. These constructs add a limited form of second-order quantification to logic programs in the form of a rich set of aggregate operators. We wish to emphasize that our use of second-order quantification is restricted greatly for the sake of efficiency: We do not allow explicit quantification over variables ranging over predicate names, nor do we allow set-valued variables. The only sets (or multisets) in our approach are conventional relations, which are sets (or multisets) of tuples. The limited second-order querying that we permit allows us to pose aggregate queries, such as SUM and COUNT, over tuples in a given (base or derived) relation, or over a subset of such tuples partitioned by the values in some set of fields. In this respect, our treatment of sets and second-order operations differs markedly from approaches such as LDL ([?]) and HiLog ([?]). The requirement that aggregation be used whenever grouping is done is novel to our work, and makes it difficult to use the first-order semantics advocated in HiLog ([?]). In essence, we have extended the facilities provided by SQL to relations that are defined recursively; our treatment ensures efficient evaluation, without the need for set-unification.

However, as with previous approaches to aggregate operations ([?, ?]), there are some difficulties in such an extension due to the combination of the **group-by** construct with recursive rules. We show that a stratification-based approach again yields an intuitive semantics, similar to *perfect models* for programs with negation. We also identify two interesting classes of non-stratified programs that have a perfect model: the first uses a notion of *rule monotonicity*, and the second, magical stratification, uses a *dependency ordering between groups* to give semantics to programs that may not be locally stratified or even modularly stratified ([?]). As in the case of programs with multisets, we show how the magic-sets transformation can be extended to programs with grouping and aggregation, thereby establishing the basis for efficient evaluation of such programs.

1.3 Summary of Our Contributions

We provide formal semantics for programs that contain multisets, aggregate operators, and the **group-by** construct, and provide a basis for efficient evaluation by extending the magic-sets technique to such programs. The results of this paper, along with those of [?] (where we extend the magic-sets technique to propagate restrictions in the form of conditions, such as $X > 5$) are critical in demonstrating that the magic-set transformation is applicable to programs in full SQL ([?]). As is demonstrated through performance results in [?], the gains of set-oriented information passing using magic-sets can be significant in terms of a simplified architecture and an efficient, stable performance over a wide range of queries.

Further, our uniform treatment of the "non-relational" features of a commercial language, such as SQL, in the presence of recursion demonstrates that deductive databases, in acquiring the power of recursion, need not sacrifice the powerful and practical features of standard relational systems. Our work shows how to introduce either or both of the two "non-relational" features discussed – multisets and aggregation – into a deductive database system. It appears that introduction of multisets increases the power of some query languages, as the following example illustrates:

EXAMPLE 1.1 Consider the following bill-of-materials query. You are given a `subpart` relation that lists for each part P all its direct subparts S. P may use more than one copy of S, and you are free to choose a representation for this. The aim is to define a `contains` relation that gives for each part P all the subparts S used in constructing P, directly or indirectly, along with an indication as to how many copies of S are used.

We conjecture that this query cannot be written in Datalog extended with stratified aggregation (Section ??), though the truth of this conjecture is not critical to our motivation. If we treat `subpart` and `contains` as multisets, then under the semantics that we propose, the program

```
contains(P, S) :- subpart(P, S).
contains(P, S) :- subpart(P, T) & contains(T, S).

count_contains(P, S, C) :-
        group_by (contains(P, S), [P, S], [C = CNT()]).
```

computes the correct answer. The last rule is interpreted as follows: Group together all tuples of `contains`(P, S) that have the same P, S values, and count the number of tuples in the group. Each group thus generates a tuple in `count_contains`(P, S, C).

The above example can be written in an SQL version that has been extended with recursion. Using SBSQL, as described in [?], we would write

```
CREATE VIEW contains(P,S)  AS
       ((SELECT P,S  FROM subpart)
         UNION  ALL
        (SELECT s.P, c.S  FROM subpart s, contains c
          WHERE s.S = c.P)).

SELECT P,S,COUNT(*) FROM contains GROUP BY P,S.
```

□

This paper is organized as follows. Section ?? gives definitions related to multisets. Duplicates are introduced in Section ??, and Section ?? discusses grouping and aggregation. Related work is presented in Section ??. Examples are written in a language similar to Datalog. The equivalent SBSQL queries can easily be derived.

1.4 Notation and Definitions

The language considered in this paper is an extension of Horn clause logic. Specifically, we use Datalog (without negation), and extend it to allow duplicates and aggregation. We define multisets and operations on them in some detail, since we use them extensively and also use some non-standard operations (such as col). We follow [?], except in the definition of multiset difference.

Definition 1.1 "Multisets": A multiset is a collection of elements that are not necessarily distinct. The number of occurrences of an element x in a multiset M is its *multiplicity* in the multiset, and is denoted by $mult(x, M)$. The cardinality $card(M)$ of a multiset M is the sum of the multiplicities of each element of M. We define $x \in M$ iff $mult(x, M) > 0$. Given a multiset M, $\mathsf{set}(M)$ is the set of elements of M, that is, $\mathsf{set}(M) = \{x \mid x \in M\}$. The difference of two multisets M_1 and M_2, denoted as $M_1 - M_2$, is a multiset in which the multiplicity of an element x is $max(mult(x, M_1)$ - $mult(x, M_2)$, 0). □

When a multiset is enumerated we use square brackets. For example, $M = [a, b, b, c]$ is a multiset with $mult(b, M) = 2$. In this paper we do not care about the representation of multisets. They could be represented by counts, or by storing multiple copies of a tuple. We now introduce a "coloring" operation on multisets that is useful for providing constructive definitions of multisets in terms of a defining property.

Definition 1.2 "Colored Sets": Let M be a multiset, and let $\mathcal{C}$ be an (infinite) ordered list of colors $c_1, c_2, \ldots$. For every element, say a, with multiplicity $n > 0$, color the copies of a with $c_1, c_2, \ldots c_n$, and denote these distinct colored elements as $a_1, a_2, \ldots, a_n$. The set of all elements obtained by thus coloring elements of M is a *colored set*, called $\mathsf{col}(M)$.

The inverse operation, $\mathsf{col}^{-1}(S)$, is defined to yield a multiset M in which the multiplicity of an element a is equal to the number of colored copies of a in the colored set S. For a colored element a_i, $\mathsf{col}^{-1}(a_i) = a$. □

Definition 1.3 "Multiset Constructor [...]": Let R be a set of n-tuples. Then $[R]$ is a multiset with multiplicity one for each element of set R. □

Finally, we introduce the multiset equivalents of the `Select`, `Project` and `Join` operators that are normally defined for sets. We use the symbols $\sigma^{set}, \sigma^{mul}, \pi^{set}, \pi^{mul}$ and $\bowtie^{set}, \bowtie^{mul}$ for the set and multiset operators, dropping the superscript where the type of the operator can be determined from the operand.

Definition 1.4 "Multiset Projection $\pi_{\bar{i}}R$": Let R be a multiset of n-tuples, and let $\bar{i}$ be a vector of k integers in the range $1 \ldots n$. For every n-tuple $\bar{t} \in R$, consider the k-tuple $(t_{i_1}, \ldots, t_{i_k})$. The multiset M, denoted by $\pi_{\bar{i}}R$, is defined to contain every such k-tuple m, with a multiplicity equal to the number of tuples $\bar{t}$ of R such that $m = (t_{i_1}, \ldots, t_{i_k})$. □

σ^{mul} and $\bowtie^{mul}$ can be defined similarly.

2 Duplicates

In this section, we consider programs in which predicates can be specified to be sets or multisets of tuples (we use the word *relation* in both cases). We develop a declarative semantics for such programs, based upon a proof-theoretic approach that determines the multiplicity of a tuple in a multiset predicate using the number of distinct proof-trees for that tuple. We present a procedural fixpoint semantics, extending the well-known Semi-Naive fixpoint evaluation technique to allow for multiset predicates. Finally, we show how to use the magic-sets transformation on such programs.

We have already seen an example that motivates the introduction of multiset predicates. Queries involving aggregates or multiset difference provide further examples where multisets are useful. We begin this section with another type of example, which appeared in the Usenet newsgroup `comp.databases`, and underlines the utility of such an extension to logic programs.

EXAMPLE 2.1 Telephone companies record for each call the source number, the called number, the start time of the call (with 1 minute granularity), and the length of the call (in multiples of 1 minute). Multiple calls of same length (1 minute) can certainly be placed between two numbers within a minute, leading to duplicates. The duplicates are important for counting the number of calls or computing the average cost of a call.[2]

It may be argued that duplicates could be avoided by using a unique ID number other than timestamp with each call, or by using a finer timestamp granularity. However duplicates are rare under this timestamping granularity, and it is more convenient just to deal with them than to have a unique id. □

The need and importance of having duplicates in a realistic implementation is widely recognized. Duplicates have been incorporated into the ISO-ANSI standard ([?]), and implementations of the relational model have extended the model to allow for duplicates. The System R prototype also dealt with duplicates ([?]). Amongst the prototype implementations of the logic data model based on Datalog type languages, NAIL! ([?], and LDL ([?]) are well-known. Neither has dealt with duplicates, choosing to stay with the traditional least Herbrand model semantics. We believe that such a limitation restricts the usability of systems based on the logic data model. Logic programming languages would be more useful as query languages if they supported the concept of duplicates.

While we want to allow duplicates for some predicates, we do not want to force every predicate to have duplicates. We therefore have two types of predicates: `set` predicates that are to be interpreted as a set of tuples, and `multiset` predicates that are to be interpreted as a multiset of tuples. The user specifies the desired interpretation using the following naming convention: The name of a `set` predicate begins with the keyword `set_` (the default), and the name of a `multiset` predicate begins with the keyword `all_`. Thus, p and `set_`p are `set` predicates and `all_`p is a `multiset` predicate.

[2] Yes, bills at IBM indicate the average cost!

2.1 A (Proof-Theoretic) Declarative Semantics

We now give the proof-theoretic semantics of logic programs without negation containing set and multiset predicates. The semantics is an extension of the traditional proof-theoretic semantics, where the number of proofs is now important. The multiplicity of an atom is the *number of proofs* for that atom. We use derivation trees, similar to the trees introduced in [?], to count the number of proofs of an atom. The semantics used in SQL and Prolog is functionally similar to the proof-theoretic semantics we give.

Definition 2.1 "Derivation Trees": Let P be a logic program without negation containing set and multiset predicates. The derivation trees of P, with respect to an edb E, $DT(P, E)$, are defined as:

- For every edb relation Q of a `multiset` predicate, every tuple h of $\mathsf{col}(Q)$ generates a derivation tree for $\mathsf{col}^{-1}(h)$, consisting of a single node with label h. For every edb relation Q of a `set` predicate, every tuple h of Q generates a derivation tree for h, consisting of a single node with label h.
- For each rule r of the form $(h :- b_1, b_2, \ldots, b_k), k > 0$, and for each tuple $(t_1, t_2, \ldots, t_k)$ of derivation trees for atoms $(d_1, d_2, \ldots, d_k)$ that unify with $(b_1, b_2, \ldots, b_k)$ with θ as the mgu, generate a derivation tree for $h\theta$, with $h\theta$ as the root label, and $(t_1, t_2, \ldots, t_k)$ as the children, in that order, and r as the edge label from the root to the children.

□

Lemma 2.1 *No two derivation trees in $DT(P, E)$ are identical. $DT(P, E)$ is thus a set.* □

Proof: By induction on the height of derivation trees, using the lemma as the inductive hypothesis. □

Each derivation tree is a representation of a proof for the atom labeling its root. (This is a slight simplification: the root label of a tree for an EDB fact may actually be a colored copy of the fact.)

Definition 2.2 "atoms": For a derivation tree t, $\mathsf{atoms}(t)$ is either the label or the col^{-1} of the label on the root of tree t. For a set of derivation trees T, $\mathsf{atoms}(T)$ is the *multiset* of atoms that includes, for every tree $t \in T$, a copy of the atom $\mathsf{atoms}(t)$. □

If $a = \mathsf{atoms}(t)$, we say that tree t is for the atom a. The set of all derivation trees for atom a in a program P with respect to an edb E is denoted by $DT(P, E, a)$.

We define the semantics of programs containing multiset predicates by a collection of derivation trees that have the following property.

Definition 2.3 "Duplicate Correctness Property": A set of derivation trees $D \subseteq DT(P,E)$ of a range-restricted program P and an edb E satisfies the Duplicate Correctness Property iff all of the following hold:

1. For every pair of nodes in D (possibly in different derivation trees), if they are both labeled with the same atom a of a `set` predicate, then the trees or subtrees rooted at these nodes are identical.
2. There is no set $D' \supset D$ that is a subset of $DT(P,E)$ and yet satisfies Condition **??**.

□

We define the predicate $\mathsf{dcp}(D,P,E)$ to be true for a set of derivation trees D that has the Duplicate Correctness Property with respect to a program P and an edb E.

Condition **??** ensures that a unique tree represents each atom of a `set` predicate, and Condition **??** ensures that all possible derivations subject to this constraint are included in the set of trees. Note that the restriction to one unique derivation tree is for `set` predicates alone. All derivation trees of multiset predicates are included in D, and are considered in building larger derivation trees.

Given a program P and an edb E, there may be multiple sets D that have the Duplicate Correctness Property. Two such sets will differ in the choice of the derivation tree for an atom of a `set` that happens to have multiple derivation trees.

Theorem 2.1 *For every program P and edb E, there exists a set D of derivation trees such that $\mathsf{dcp}(D,P,E)$. Further, if D_1 and D_2 are two distinct sets such that $\mathsf{dcp}(D_1,P,E) \wedge \mathsf{dcp}(D_2,P,E)$, then $\mathsf{atoms}(D_1) = \mathsf{atoms}(D_2)$.* □

Proof: Let $DT^{(h)}(P,E)$ be all derivation trees in $DT(P,E)$ of height $\leq h$. We can construct a set $D^{(h)} \subseteq DT^{(h)}(P,E)$, and prove, by induction on h, that

1. $\mathsf{dcp}(D^{(h)},P,E)$ (we modify Definition **??** of *dcp* slightly to consider maximal subsets of $DT^{(h)}(P,E)$, rather than maximal subsets of $DT(P,E)$ in Condition 2), and
2. If there exists another set $D'^{(h)} \subseteq DT^{(h)}(P)$ such that $\mathsf{dcp}(D'^{(h)},P,E)$, then $\mathsf{atoms}(D^{(h)}) = \mathsf{atoms}(D'^{(h)})$.

□

Definition 2.4 "Duplicate Semantics": Given a logic program P without negation, with `set` and `multiset` predicates, let $D \subseteq DT(P,E)$ be a set of derivation trees having the Duplicate Correctness Property. The duplicate semantics of the logic program P with respect to edb E is the multiset $ds(P,E) = \mathsf{atoms}(D)$. □

Definition 2.5 "Duplicate Equivalence": Programs P_1 and P_2 are duplicate equivalent iff $ds(P_1,E) = ds(P_2,E)$ for every edb E. □

2.2 Computational Semantics

A program with `set` and `multiset` predicates may be evaluated by a combination of Semi-Naive and Not-So-Naive ([?]) evaluation techniques, the former working on `set` predicates, and the latter working, in parallel, on `multiset` predicates.

Definition 2.6 "Rule Application": $\mathtt{Incr\_Eval}(p,P,D,\Delta)$ is the multiset of facts derivable from the database D (a multiset of atoms) and an incremental database Δ (that has not yet been included in D) through a single application of the rules for predicate p in program P, using at least one fact from Δ.

$$\begin{array}{l}\mathtt{Incr\_Eval}(p,P,D,\Delta) = \\ \quad \pi^{mul}_{h\theta}[\{(h\theta, r, d_1, d_2, \ldots, d_k) \mid \\ \quad r \text{ is a rule for predicate } p \text{ in program } P \text{ of the form} \\ \qquad\qquad (h :- b_1, b_2, \ldots, b_k), k > 0, \textit{ and} \\ \quad (d_1, d_2, \ldots, d_k) \in \mathsf{col}(D \cup \Delta) \wedge (\exists i)_{1 \leq i \leq k}(d_i \in \mathsf{col}(\Delta)), \textit{ and} \\ \quad (\theta \text{ is the mgu of } (b_1, b_2, \ldots, b_k) \text{ and } (d_1, d_2, \ldots, d_k)) \\ \qquad \}].\end{array}$$

□

The above formula is read as follows: For every rule r for predicate p, and for every tuple $(d_1, d_2, \ldots, d_k)$[3] of elements of $\mathsf{col}(D \cup \Delta)$ that unifies with the subgoals of rule r in that order, such that at least one of the $d_i's$ also appears in $\mathsf{col}(\Delta)$, add $h\theta$ to the result, where θ is the mgu of $(d_1, d_2, \ldots, d_k)$ and the body literals. An element appearing several times in $D \cup \Delta$ can thus contribute multiple copies of $h\theta$ to the result, as the various copies of the element will appear as differently colored elements of $\mathsf{col}(D \cup \Delta)$.

For a set T of predicates, define an *iteration* as $\mathtt{Incr\_Eval}(T,P,D,\Delta) = \bigcup_{p \in T}(\mathtt{Incr\_Eval}(p,P,D,\Delta))$.

We now define the Duplicate Semi-Naive (DSN) evaluation technique. DSN is similar to Semi-Naive evaluation, except that duplicates for `multiset` predicates are not eliminated. Let P_S and P_M be the `set` and `multiset` predicates in program P, and let $\mathcal{M}$ and $\mathcal{S}$ denote their respective extensions. $\mathcal{D} = \mathcal{M} \cup [\mathcal{S}]$ is the database. Given a program P and an edb E, DSN generates a multiset $\mathcal{D} = \mathsf{dsn}(P,E)$.

Definition 2.7 "Duplicate Semi-naive (DSN) Algorithm":

1. $\mathcal{D}_0 = \emptyset$; $\delta\mathcal{D}_0 = E$.
2. $\mathcal{S}_{n+1} = \mathcal{S}_n \cup \delta\mathcal{S}_n$; $\mathcal{M}_{n+1} = \mathcal{M}_n \cup \delta\mathcal{M}_n$.
3. $\delta\mathcal{S}_{n+1} = \mathtt{set}(\mathtt{Incr\_Eval}(P_S,P,\mathcal{D}_n,\delta\mathcal{D}_n)) - \mathcal{S}_{n+1}$; $\delta\mathcal{M}_{n+1} = \mathtt{Incr\_Eval}(P_M,P,\mathcal{D}_n,\delta\mathcal{D}_n)$.

[3] Note: Each d_i is an atom, a tuple of a relation.

4. $\mathcal{D}_{n+1} = [\mathcal{S}_{n+1}] \cup \mathcal{M}_{n+1}$; $\delta\mathcal{D}_{n+1} = [\delta\mathcal{S}_{n+1}] \cup \delta\mathcal{M}_{n+1}$.
5. $\mathsf{dsn}(P, E) = \mathcal{D} = \lim_{n\to\infty} \mathcal{D}_n$

□

DSN terminates (at Step $n+1$) when $\mathcal{D}_{n+1} = \mathcal{D}_n$. DSN may not terminate for some programs where Semi-Naive does terminate. An example is the `multiset` transitive closure of a cyclic graph, where the answer is, by definition, infinite.

The DSN algorithm can alternatively be viewed as operating on derivation trees instead of atoms. Each atom derived by DSN corresponds to a derivation tree constructed for that atom. The relationship can be made explicit by defining a function `Incr_Tree` that does incremental computation on derivation trees, mimicking the computation of `Incr_Eval` on atoms. We omit the details here, but the equivalence helps us to establish the following result:

Theorem 2.2 *The DSN algorithm correctly computes the duplicate semantics of a logic program P. That is,* $dsn(P, E) = ds(P, E)$. □

DSN evaluation without coloring

The definition of `Incr_Eval` suggests a computation where each iteration is preceded by a coloring phase. A simpler computation, without any coloring, is possible. We use multiset versions of join, selection, and projection, and extend the ATOV (argument to variable) and VTOA (variable to argument) functions of [?] to multisets.

$\mathsf{ATOV}(q(\bar{t}), Q)$ maps the relation Q for predicate q to a relation on variables in the goal $q(\bar{t})$.

Definition 2.8 "ATOV ([?])": Given the goal $q(t_1, \ldots, t_m)$ and a relation Q for predicate q, define a relation Q' over the variables $X_1, \ldots, X_n$ appearing in $t_1, \ldots, t_m$ as follows:

$Q' = \emptyset$;
for each tuple $q(s_1, \ldots, s_m)$ in Q **do**
  if there is a term matching τ for tuple $q(s_1, \ldots, s_m)$
    and goal $q(t_1, \ldots, t_m)$ **then**
   add to Q' the tuple $(\tau(X_1), \ldots, \tau(X_n))$

$\mathsf{ATOV}(q(\bar{t}), Q) = Q'$. □

VTOA is the complement of ATOV. $\mathsf{VTOA}(q(\bar{t}), Q')$ takes a relation Q' on the variables appearing in $\bar{t}$, and produces a relation for predicate q.

Definition 2.9 "Incr_Eval_2": Let r be a rule for predicate p in program P of the form $(h :- b_1, b_2, \ldots, b_k)$, $k > 0$, let $(D_1, D_2, \ldots, D_k)$ be the relations for the predicates of subgoals $(b_1, b_2, \ldots, b_k)$ in a database D, and let $(\Delta_1, \Delta_2, \ldots, \Delta_k)$ be the corresponding relations in an incremental database Δ. As before, we assume that the incremental relations have not been included in the main database.
Let $D'_i = \mathsf{ATOV}(b_i, D_i), \Delta'_i = \mathsf{ATOV}(b_i, \Delta_i)$, and let δ_j be given by either of the following two equivalent joins:

$$D'_1 \bowtie \ldots \bowtie D'_{j-1} \bowtie \Delta'_j \bowtie (D'_{j+1} \cup \Delta'_{j+1}) \ldots \bowtie (D'_k \cup \Delta'_k)$$

or

$$(D'_1 \cup \Delta'_1) \bowtie \ldots \bowtie (D'_{j-1} \cup \Delta'_{j-1}) \bowtie \Delta'_j \bowtie D'_{j+1} \ldots \bowtie D'_k$$

Define

$$\begin{aligned} \mathtt{Incr\_Eval\_2}(p, r, D, \Delta) &= \mathsf{VTOA}(h, \bigcup_{1 \le j \le k} \delta_j) \\ \mathtt{Incr\_Eval\_2}(p, P, D, \Delta) &= \bigcup_{r} \mathtt{Incr\_Eval\_2}(p, r, D, \Delta) \end{aligned}$$

□

Let DSN_2 be the version of the DSN algorithm using `Incr_Eval_2` instead of `Incr_Eval`.

Theorem 2.3 *Algorithms DSN and DSN_2 are equivalent. That is,* $dsn(P, E) = dsn\_2(P, E)$. □

By building common subexpressions carefully, `Incr_Eval_2` can be evaluated using $3k - 4$ joins.

2.3 Magic-sets Transformation

We first show that the process of adorning a program, using any adornment pattern (see [?] for an introduction to adornments. More complex adornments are discussed in [?]) whatsoever, preserves the duplicate semantics of a program. We then show that a magic-sets transformation ([?, ?, ?]) on an adorned program, preserving its duplicate semantics, can be defined.

2.3.1 Adorning a Program

Definition 2.10 "Predicate Copying": Given a set or a multiset predicate p in a program P, create another predicate p^α of the same type as p, using the rules for predicate p in the program P. p^α has exactly the same rules as p, except that the head of these rules are for predicate p^α rather than for p. p^α and p are called *predicate copies* of each other. The set of predicate copies of a predicate p forms a *predicate class*. □

Lemma 2.2 *Let P be a program obtained by one or more predicate copying operations. If p^{α_1} and p^{α_2} are members of the same predicate class in P, then the relations for p^{α_1} and p^{α_2} in $ds(P, E)$, for any edb E, are identical.* □

Theorem 2.4 *Let a rule of a program P be modified by replacing an occurrence of a subgoal $p^{\alpha_1}(\bar{t})$ by $p^{\alpha_2}(\bar{t})$ where p^{α_1} and p^{α_2} are members of the same predicate class, to get a program P'. Then P and P' are duplicate equivalent.* □

Corollary 2.1 *Let program P' be obtained from P by doing one or more predicate copies followed by one or more replacement operations as defined in Theorem ??. (Programs such as P' are called* adorned *programs). Let predicate p appear in both P' and P. Then the relation for p in $ds(P, E)$ is identical to the relation for p in $ds(P', E)$.* □

2.3.2 Magic-sets

Given an adorned program query pair (P, Q), define the magic-sets transformation for programs with multiset predicates in two steps:

1. Let P' be the magic-sets transformed program obtained without regard to duplicate semantics.

2. Define each magic predicate to be a `set` predicate. Call the program P''.

The following theorem is significant in that it ensures the correctness of the magic-sets transformation under duplicate semantics, and thereby demonstrates that the declarative duplicate semantics can be efficiently evaluated, without irrelevant computation.

The magic-sets transformation is known to preserve equivalence with respect to the minimal model semantics (for programs without duplicates). In other words, a fact has a derivation tree in the transformed program P' or P'' iff it has a derivation tree in the original program P. We must additionally prove that, with the adaptation above, a fact in P'' is supported by exactly the same number of derivation trees as in P.

Theorem 2.5 *Program P and the magic-sets transformed program P'' are duplicate equivalent to with respect to the query predicate Q.* □

Proof: For a derivation tree t' of a program containing magic predicates, let $\mathsf{magred}(t')$ denote the derivation tree obtained by removing from t' all nodes labeled by an atom of a magic predicate. Let $D \subseteq DT(P, E)$ be a set of derivation trees such that $\mathsf{dcp}(D, P, E)$ holds. We prove, by induction on the height of derivation trees in D, that there is a set of trees $D'' \subseteq DT(P'', E)$ such that $\mathsf{dcp}(D'', P'', E)$ holds, and:

1. For every tree t in D for $p(a)$, *one* of the following two conditions holds:

   - There is exactly one tree t'' in D'' for $p(a)$ such that $\mathsf{magred}(t'') = t$, OR
   - There is no tree in D'' for $\mathtt{set\_m\_}p(\pi a)$,[4] and hence there is no tree for $m\_p(\pi a)$ in $DT(P', E)$.

2. For every tree t'' in D'', there is a derivation tree t in D such that $\mathsf{magred}(t'') = t$.

[4] $\mathtt{set\_m\_}p$ is the magic predicate for p. πa denotes the projection of a onto the arguments of $\mathtt{set\_m\_}p$.

Intuitively, the first condition ensures that every derivation tree for a relevant fact is computed in the transformed program, and that no such tree is computed twice. The second condition ensures that no spurious facts are established. □

An important consequence of the magic-sets adaptation is the following: For programs that are duplicate-free [?], we need not perform any form of duplicate elimination on non-magic predicates. Thus, the result has significance even for programs in which multiset predicates are not explicitly used, since it indicates how a special property of the original program can be exploited in evaluating the transformed program, even though the transformed program may not enjoy this property.

2.4 Duplicate Semantics for Non-Range-Restricted Programs

Computation with non-range-restricted programs involves storing non-ground tuples. In addition to sets and multisets of non-ground tuples, we introduce a new data structure: An `irrset` is a set in which no element subsumes another.

The presence of non-ground tuples raises the possibility that a tuple may be subsumed by another that is generated in a later iteration. This seriously complicates the computation, and we require stratification with respect to the use of `irrset` predicates. We can extend the declarative and computational semantics to irrset-stratified programs.

3 Grouping and Aggregation

The amount of data kept in databases is frequently large and is expected to grow significantly. User queries often involve some form of data reduction, and the query language must provide operations to support this. For example, SQL supports a set of aggregate functions such as *average* and *sum*. First-order logic does not deal with grouping and aggregation since variables range over one tuple or one component of one tuple. To be able to do grouping, we need to have a single variable range over a property of a subset of columns and/or rows of a relation. The logic data model using Datalog and its various extensions does not include grouping and aggregation. LDL ([?]) allows us to construct set-terms by "grouping" all instantiations of a term in a rule body, but it does not support aggregation.

The extension to grouping and aggregation that we discuss can be used both with and without support for multisets in the system.

3.1 Syntax

We define a special second order predicate, $\mathtt{group\_by}\,(r(\bar{t}), GL, AL)$, that takes as arguments a goal r with its attribute list $\bar{t}$, a grouping list GL of variables appearing in $\bar{t}$, and an aggregation list AL of aggregate functions. The general form of a `group_by` subgoal is:

$$(G): \quad \texttt{group\_by}\,(r(\overline{t}),\,[Y_1, Y_2, \ldots, Y_m],\\ [Z_1 = A_1(S_1(E_1)), \ldots, Z_n = A_n(S_n(E_n))]).$$

The predicate r is called the grouping predicate. The arguments, $\overline{t}$ of r can be general terms: constants, variables, or complex terms. However, any variables in $\overline{t}$ are local to the `group_by` , unless they also appear in the grouping list. The grouping list consists of zero or more distinct variables that must appear in $\overline{t}$. The list of Z's is called the aggregation list. Each Z is a new variable, E is an expression that uses variables of $\overline{t}$, S is optional — the keyword `set` can be used to remove duplicates before aggregation, and A is an aggregate operator that maps a monadic relation to a single value (such as SUM, CNT). Within the rule body, a `group_by` subgoal represents a relation over variables $\overline{Y}$ in the grouping list and variables $\overline{Z}$ in the aggregation list.

For an expression E and tuple s, let $E(s)$ be the result of applying the expression E to tuple s. The operation is well-defined since the variables in E refer to attributes of s. For a set or multiset relation U, let $E(U)$ be the multiset $[E(s)|s \in U]$.

If a rule for predicate p has the groupby subgoal G in the body, we say that p depends on r through a grouping operation, and insert the edge $r \xrightarrow{gb} p$ in the dependency graph. gb is the label of the dependency edge.

3.2 Semantics

3.2.1 Semantics of a `group_by` Subgoal

An ordinary subgoal $p(\overline{t})$ of a rule defines an ATOV mapping from a relation P for predicate p to a relation over the variables in $\overline{t}$. Satisfiability of a rule in an interpretation requires testing the satisfiability of each of the subgoals for each substitution of the variables of the rule. For the subgoal $p(\overline{t})$, checking satisfiability for a substitution σ simply involves checking whether the tuple $p(\overline{t})\sigma$ is in the relation P. (Or, for a negated subgoal, it involves checking that the tuple is not in the relation P). Equivalently, satisfiability can be tested by doing the $\mathsf{ATOV}(p(\overline{t}), P)$ mapping, and checking whether the tuple corresponding to the substitution σ is in the mapping.

Within a rule we give the `group_by` subgoal a semantics similar to any other subgoal. The `group_by` subgoal G defines an ATOV mapping from the grouping relation R to a relation over the variables $(\overline{Y}, \overline{Z})$.

Definition 3.1 "ATOV of `group_by` ":

$$\mathsf{ATOV}(\texttt{group\_by}\,(r(\overline{t}),\,[Y_1, Y_2, \ldots, Y_m],\\ [Z_1 = A_1(S_1(E_1)), \ldots Z_n = A_n(S_n(E_n))]), R)\\ = T$$

where T is defined as follows:

1. Let $R' = \mathsf{ATOV}(r(\overline{t}), R)$. R' is a relation over the variables in $\overline{t}$.
2. Let $G = \pi^{set}_{Y_1,Y_2,\ldots,Y_m}(R')$. We use a set projection in this step even if multisets are present. If $m = 0$, G is a relation with no attributes, having either a single empty tuple (if R is not empty), or no tuples (if R is empty).[5]
3. For each tuple μ in G, define a tuple $f(\mu)$ as follows:
   - Let $R'_\mu = R' \bowtie \mu$. R'_μ is thus the maximal subset of R' having the same values for the attributes $Y_1, Y_2, \ldots, Y_m$ as the tuple μ. If $m = 0, \mu$ will be the empty tuple, and $R' \bowtie \mu = R'$.
   - Compute the multisets $R'_\mu\_1 = E_1(R'_\mu)$, $R'_\mu\_2 = E_2(R'_\mu), \ldots, R'_\mu\_n = E_n(R'_\mu)$.
   - If $S_i = \mathsf{set}$, let $R'_\mu\_i = \mathsf{set}(R'_\mu\_i)$.
   - Compute $Z_1 = A_1(R'_\mu\_1)$, $Z_2 = A_2(R'_\mu\_2)$, $\ldots$, $Z_n = A_n(R'_\mu\_n)$. Each Z_i will be a single value.
   - Let $f(\mu) = \mu \times Z_1 \times Z_2 \times \ldots \times Z_n$
4. Let $T = \{f(\mu) \mid \mu \in G\}$. Note that T is always a `set`.

□

Theorem 3.1 ATOV *of a* `group_by` *subgoal is nonmonotonic.* □

Proof: Adding a tuple to the grouping relation R can change the number of tuples in a group (R'_μ). As a result, an aggregate value previously derived may no longer be derivable. □

Definition 3.2 "group": A group of a relation $R(\overline{X}, \overline{Y})$ with respect to the grouping list $\overline{Y}$ and values $\overline{y}$, written as $\mathsf{group}(R(\overline{X}, \overline{Y}), \overline{Y}, \overline{y})$, is defined to be the relation $\sigma_{(\overline{Y}=\overline{y})}R$. $\overline{y}$ is the grouping value of the group. □

Definition 3.3 "groupset": The groupset of a relation with respect to a grouping list $\overline{Y}$, written as $\mathsf{groupset}(R(\overline{X}, \overline{Y}), \overline{Y})$, is defined to be the *set* of groups $\{\mathsf{group}(R(\overline{X}, \overline{Y}), \overline{Y}, \overline{y}) \mid \overline{y} \in \pi^{set}_{\overline{Y}}(R)\}$. □

Several properties exist between groups and the relations R', R'_μ, and T of Definition ??. $\mathsf{group}(R', \overline{Y}, \mu) = R'_\mu$. Each group g in $\mathsf{groupset}(R', \overline{Y})$ contributes exactly one tuple to T, so that the number of tuples in T is equal to the cardinality of $\mathsf{groupset}(R', \overline{Y})$. We can then derive the following important theorem.

[5] SQL does not permit $m = 0$ when using `group_by` . However SQL allows the aggregation operators to be used without grouping. The semantics of the aggregation without grouping in SQL is identical to our grouping with $m = 0$ if $R \neq \emptyset$, but differs when $R = \emptyset$. SQL assumes that a group consisting of the full relation R always exists when grouping is not done, and the aggregate operators are applied to this one group. With $R = \emptyset$, an empty group is generated, while grouping with $m = 0$ according to our definition will generate no groups. To make grouping with $m = 0$ equivalent to SQL aggregation without grouping, we would need to define G to have the single empty tuple when $m = 0$ and $R = \emptyset$. Special boundary cases in some of the properties and theorems will be required as a consequence.

Theorem 3.2 ATOV *of a* `group_by` *subgoal is monotonic with respect to new groups. That is, for the* `group_by` *subgoal* G*, if* $R_2 = R_1 \cup \Delta$*, and* $\mathsf{groupset}(R_2', \overline{Y}) = \mathsf{groupset}(R_1', \overline{Y}) \cup \mathsf{groupset}(\Delta', \overline{Y})$*, then* $T_2 \supseteq T1$. □

3.2.2 Model Theoretic Semantics

Definition 3.4 "Model": A model of a logic program with grouping and aggregation is an interpretation that interprets each edb predicate as the given edb relation and satisfies all the rules. A rule is satisfied if for every substitution that satisfies each of the subgoals, the corresponding head atom is in the interpretation. A substitution σ satisfies the `group_by` subgoal G

$$(G): \quad \texttt{group\_by}\,(r(\bar{t}),\, [Y_1, Y_2, \ldots, Y_m],\\ [Z_1 = A_1(S_1(E_1)), \ldots, Z_n = A_n(S_n(E_n))]).$$

if the tuple $(\overline{Y}\sigma, \overline{Z}\sigma)$ is in

$$\mathsf{ATOV}(\texttt{group\_by}\,(r(\bar{t}),\, [\overline{Y}],\, [\overline{Z} = \overline{A}(\overline{E})], R))\,,$$

where R is the relation for predicate r in the interpretation. □

Several observations are in order: There are non-recursive programs that have multiple minimal models; if the domain provides an infinite number of constants, even range-restricted programs without function symbols may not have a finite minimal model — the aggregate operators behave like function symbols, so that every minimal model is infinite; the union of two minimal models may not even be a model of the program; and there are programs that have no intuitive minimal model.

We identify classes of programs for which the model-theoretic semantics can be defined in terms of a perfect model. *Aggregate Stratification* is similar to Stratified Negation ([?]), and disallows recursions through `group_by` . A local stratification analog (Group Stratification) can also be defined. *Monotonic Programs* and *Magical Stratified* programs are two interesting classes of non-stratified programs that are closed under the magic-set transformation. Non-stratified programs involve recursion through the grouping operator.

It is straightforward to extend the results of this section to provide a (proof-theoretic) declarative semantics for programs that allow multiset predicates and aggregate operations. We call such programs `SQLog` programs.

3.2.3 Computational Semantics

In the following subsections we provide computational semantics separately for each of the classes of programs we consider.

3.3 Aggregate Stratified Programs

The programs that have no intuitive minimal model all involve a recursion through a `group_by` subgoal. That is, such programs have mutually recursive predicates, p and q (not necessarily distinct), such that q is the grouping relation of a `group_by` subgoal in a rule for p.

Therefore, a sufficient syntactic condition for the existence of an intuitive model is the absence of recursions through `group_by` . The resulting class of programs is said to be aggregate stratified. A stratification of an aggregate stratified program can be defined in a manner similar to the stratification of a negation stratified program ([?]). The semantics of an aggregate stratified program is given by an intuitive *perfect model*, similar to the perfect model of a program with stratified negation.

Definition 3.5 "Perfect Model of Aggregate Stratified Programs": Given an aggregate stratified program P, define its perfect model, $\mathcal{M}$, as the minimal model of P that has the following properties:

1. If $\mathcal{M}'$ is another model (minimal or not) of P, then for every predicate p of stratum 1, the relation for p in $\mathcal{M}$ is a subset of the relation for p in $\mathcal{M}'$.
2. If $\mathcal{M}'$ is another model of P that agrees with $\mathcal{M}$ on all predicates of stratum i and less, then for every predicate p of stratum $i+1$, the relation for p in $\mathcal{M}$ is a subset of the relation for p in $\mathcal{M}'$.

□

Theorem 3.3 *Every aggregate stratified program has a unique perfect model.* □

Proof: Let P_i be the subprogram of an aggregate stratified program P consisting of predicates of stratum i or less. We prove, by induction on i, that P_i has a unique perfect model. □

We can extend the bottom-up computational semantics to aggregate stratified programs using a straightforward layer-by-layer approach (similar to the computational semantics for stratified negation), and prove that it is equivalent to the perfect model semantics.

There are examples that involve recursion through grouping and yet have an intuitive minimal model semantics. The magic-sets transformation of aggregate stratified programs often leads to programs that are not aggregate stratified, though they do have an intuitive minimal model. In the following subsections we will define semantics for some of the interesting recursive examples, and give a class of programs that is closed under the magic-sets transformation.

3.3.1 Group Stratified Programs

In an aggregate stratified program, a predicate cannot be defined by grouping over itself. Since `group_by` is monotonic across groups (Theorem ??), what we really want is that a *group of the predicate* should not depend on itself; it may well depend on *another group* of the same predicate. Programs having this property are said to be group stratified. The idea is analogous to local stratification for negation. The perfect model for group stratified programs can be defined using a prioritized minimization of groups.

The following program was suggested by a referee as a way to express the query of Example ?? without using multisets:

$(T1)$: $\texttt{contains}(P, S, \texttt{null}, C)$:– $\texttt{subpart}(P, S, C)$.
$(T2)$: $\texttt{contains}(P, S, U, C)$:– $\texttt{subpart}(P, U, C_1)$ & $\texttt{count\_contains}(U, S, C_2)$ & $C = C_1 * C_2$.

$(T3)$: $\texttt{count\_contains}(P, S, C)$:– $\texttt{group\_by}$ ($\texttt{contains}(P, S, U, M), [P, S], [C = \mathsf{SUM}(M)]$).

$\texttt{contains}(P, S, U, C)$ means that P has C units of S by virtue of having U as a direct subpart. Program T is not aggregate stratified. However, if the $\texttt{subpart}$ relation is acyclic, program T is group stratified, with an ordering between the (P, S) groups of $\texttt{contains}$ defined by a topological sort on the $\texttt{subpart}$ relation. If the group ordering is known, the preferred model of T can be computed by a semi-naive evaluation where rule $T3$ is fired for groups in the given order.

3.4 Monotonic Programs

We now consider a class of non-stratified programs for which an intuitive model exists. The following example is illustrative.

EXAMPLE 3.1 (Corporate Takeovers): We are given a relation $\texttt{set\_owns}(C_m, C_s, S)$ with the interpretation that company C_m directly owns S% of the stock of company C_s.

A company C_m is said to have bought another company C_s if C_m controls more than 50% of the stock of C_s. C_m controls the stock it directly owns. C_m also controls stock controlled by any other company C_m has bought. Consider the program

$(C1)$: $\texttt{all\_controls}(C_m, C_s, S)$:– $\texttt{set\_owns}(C_m, C_s, S)$.
$(C2)$: $\texttt{all\_controls}(C_m, C_s, S)$:– $\texttt{set\_bought}(C_m, C_i)$ & $C_m \neq C_i$ & $\texttt{all\_controls}(C_i, C_s, S)$.

$(C3)$: $\texttt{set\_bought}(C_m, C_s)$:– $\texttt{group\_by}$ ($\texttt{all\_controls}(C_m, C_s, S), [C_m, C_s], [A = \mathsf{SUM}(S)]$) & $A > 50$.

□

The above program is not aggregate stratified. (It is not even group stratified.) However, it has an intuitive minimal model, and a bottom-up evaluation will compute the intuitive model. We thus need a weaker restriction than group stratification on SQLog programs. A semantic condition satisfied here is that if a group derives another tuple in the same group, thereby possibly changing the ATOV of the group-by subgoal for that group, the head atom derived earlier from the rule is still derivable. We formalize this semantic condition.

Definition 3.6 "Monotonic Rule": We call a rule monotonic if adding new tuples to the relations for its ordinary subgoals, or to the grouping relations of its $\texttt{group\_by}$ subgoals, can only add tuples to the head (that is, cannot invalidate a deduction) regardless of the relations for other subgoals in the rule. □

Definition 3.7 "Monotonic Program": A program is monotonic if every rule in it is monotonic. □

Clearly, a rule without a $\texttt{group\_by}$ subgoal is monotonic. A rule will not be monotonic if any variable from the aggregation list is used in the head or in an ordinary subgoal in the body, because adding tuples to the grouping relation will change the aggregate value. However, consider an element $Z = A(E)$ in the aggregation list. We can state a sufficient condition for the rule to be monotonic in terms of the literals in which Z appears, assuming that the range of the expression E is known. For Datalog rules the condition is necessary and sufficient.

Definition 3.8 "Monotonic Literal": Let a rule r have as subgoals the literal $l(Z)$ containing variable Z and the $\texttt{group\_by}$ literal G, with the element $Z = A(E)$ in the aggregation list. $l(Z)$ is said to be *monotonic* with respect to a domain D and the element $(Z = A)$ if $l(Z)$ is built-in and adding tuples to the grouping relation never changes the truth value of $l(Z)$ from *true* to *false*, provided that the range of expression E is a subset of domain D. □

EXAMPLE 3.2 The literals $S > c$, where c is a constant, is monotonic with respect to $(S = \mathsf{SUM}, R^+)$, where R^+ is the domain of positive reals. $S > c$ is not monotonic with respect to $(S = \mathsf{SUM}, R)$, where R is the domain of all reals.

Similarly, $M > c$ is monotonic with respect to $(M = \mathsf{MAX}, R^+)$, $(M = \mathsf{MIN}, R^-)$ and $(M = \mathsf{COUNT}, R)$. $M < c$ is monotonic with respect to $(M = \mathsf{MIN}, R^+)$. □

Theorem 3.4 *Let a rule r contain a* group-by *subgoal with element $Z = A(E)$ in the aggregation list, and let D be the range of expression E. Then, a sufficient (and, for Datalog rules with $>, \geq, <, \leq, =, \neq$ as the only built-in predicates, necessary) condition for the rule r to be monotonic is that Z must only appear in body literals that are monotonic with respect to $(Z = A, D)$. (This condition must hold for all elements of the aggregation list.)* □

By Theorem ??, the Corporate Takeover example is monotonic, assuming that the domain of S in the relation $\texttt{set\_owns}(C_m, C_s, S)$ is limited to positive reals. However, if we change the condition in rule $C3$ to $S < 50$, rule $C3$ is no longer monotonic since $S < c$ is not monotonic with respect to $(S = \mathsf{SUM}, R^+)$.

Theorem 3.5 *Every monotonic program has a perfect model that can be computed by a bottom-up evaluation.*
□

Proof: We use the following idea: If we add new tuples to a relation q we are grouping on, any deductions made in the previous iteration from a rule r doing a groupby on q will be repeated in the next iteration since r is monotonic. □

Stratified Monotonic Programs

The ideas of monotonicity and stratification can be combined to define a perfect model for a class of *stratified monotonic programs.* We consider the strongly connected components of a program P. Let q be the grouping relation in a groupby subgoal G of a rule r for relation p, and let p and q be mutually recursive (in the same strongly connected component). For P to be stratified monotonic, we require that the rule r be monotonic with respect to the groupby subgoal G. The perfect model of such a program can be defined by a prioritized minimization of the strongly connected components.

3.5 Magical Stratified Programs

We consider another class of recursions through groupby for which a perfect model can be defined. We define the class of Magical Stratified Programs by a semantic condition (unlike the syntactic conditions used to define aggregate stratified and monotonic programs). The following example motivates the magical stratified class:

EXAMPLE 3.3 (Magical Stratification)

$(M1)$: $p(X,Y)$:- $m\_p(X)$ & $t(X,Y)$.
$(M2)$: $p(X,Y)$:- $m\_p(X)$ & $\texttt{group\_by}\,(r(X,W),[X],[Z=\mathsf{SUM}(W)])$ & $p(Z,Y)$.

$(M3)$: $r(X,Y)$:- $m\_r(X)$ & $u(X,Z)$ & $v(Z,Y)$.

$(M4)$: $m\_p(Z)$:- $m\_p(X)$ & $\texttt{group\_by}\,(r(X,W),[X],[Z=\mathsf{SUM}(W)])$.
$(M5)$: $m\_p(5)$.

$(M6)$: $m\_r(X)$:- $m\_p(X)$.

The dependencies $r \xrightarrow{gb} m\_p \rightarrow m\_r \rightarrow r$ make r and m_p mutually recursive. Thus, program M has recursion through grouping and is not aggregate stratified. Program M is not even monotonic; the rules $M2$ and $M4$ are not monotonic.

Program M does have an intuitive model. Consider a variant of the bottom-up evaluation technique where application of rules $M2$ and $M4$ that do a grouping over r is delayed until no new tuples can be derived in an iteration. Let us assume that u and v are edb's. Then, after $m\_r(5)$ is derived, all tuples of $r(X,Y)$ in the group $X=5$ can be deduced in one iteration. At this point, no new tuples for any relation can be derived. We therefore activate rules $M2$ and $M4$ and do the grouping on r. The grouping may recursively derive new tuples of m_r, and hence new tuples in r. If $m\_r(6)$ is derived recursively, the new tuples derived for r will all be in a new group $X=6$. If $m\_r(5)$ is recursively derived, no new tuples will be derived for r, since all r tuples in group $X=5$ were derived in a previous iteration. In either case, the grouping operation done earlier for the group $X=5$ is not invalidated by recursively derived r tuples.

The reader can probably recognize program M as the result of a magic-sets transformation. If we add another base rule for m_r, M will no longer be a magic-sets transformation, but it will continue to have an intuitive semantics. □

[?] gives semantics for *modular stratified programs* in which each strongly connected component is locally stratified once all instantiated rules with a false subgoal that is defined in a lower component are removed. The definition is given for programs with negation, but can be extended to programs with grouping in a natural way, requiring that a group not determine a tuple in the same group through the grouping operation, once all instantiated rules with a false subgoal that is defined in a lower component are removed. In the above program M, the group $X=5$ of r can derive the tuple $r(5,5)$ in the same group through the grouping operation; so M is not modularly stratified. However, there is also a derivation for the same tuple $r(5,5)$ without the grouping operation, and this allows M to have an intuitive model.

If every ground tuple in a program has at least one derivation from the "lower" ground tuples, regardless of any additional cyclic derivations, we could define a perfect model for the program. The program M above has this important property.

We identify a subclass (Magical Stratified) of programs that have such a property. A component of a magical stratified program may have a relation p defined recursively in terms of grouping over p. To avoid incorrect derivations due to grouping over an incomplete relation, we require that the grouping operation be applied to a group of p only after the full group has been computed. It can be difficult to test whether a group has been computed fully. We therefore require that each group of p either be fully derivable without using the grouping operation, or no tuple of the group be derivable without using the grouping operation. Then, if we suspend the grouping operation until no tuples for p can be derived without it, we can be sure that the grouping operation will be correct.

An attribute of a predicate p is a *grouping attribute* if the attribute is used in the grouping list of any groupby operation on relation p. We define derivation trees for programs with grouping, in a manner similar to derivation trees for Datalog programs. A ground groupby subgoal $\texttt{group\_by}(p(\overline{t}), [\overline{y}], [\overline{z} = \overline{A}(\overline{E})])$ is supported by all tuples in the group $\overline{Y} = \overline{y}$ of p that have a derivation tree.

Definition 3.9 "Magical Stratification": A program P is magical stratified if every strongly connected component S of P either does not have a gb edge, or it satisfies the following property:

In the dependency graph of component S, let there be a cycle with a groupby edge $p \stackrel{gb}{\rightarrow} t$, and let $\overline{X}$ be the grouping attributes of p. If there are more than one groupby edges out of p in component S, let X be the intersection (maybe empty) of the grouping lists of each groupby. Then, there is a predicate m_p (`set_`m_p, if duplicate semantics is used) with attributes $\overline{Y} \subseteq \overline{X}$ such that

1. (**Syntactic Condition**) Every rule for p has m_p as a subgoal, and
2. (**Semantic Condition**) All tuples of p in the perfect model of S that match with a given tuple μ of m_p should have a derivation tree that does not use the grouping operation over p, provided we consider $m\_p(\mu)$ to be a base fact.

m_p is called the magical predicate of p. □

Program M of Example ?? is magical stratified. The predicates r, m_p, and m_r are in one strongly connected component, and $r \stackrel{gb}{\rightarrow} m\_p$ is the groupby edge. m_r satisfies the syntactic condition for a magical predicate of r. If we initialize m_r to the tuple $m\_r(5)$, all matching r tuples can be derived by one application of rule $M3$ (without using rule $M4$). The semantic condition is thus satisfied. Note that M will remain magical stratified if we add a base rule for m_r.

Due to the semantic condition, it may not be decidable to determine whether a program is magical stratified or not. However, aggregate stratification implies magical stratification. In Section ?? we show that the class of magical stratified programs is closed under the magic-sets transformation. At the very least, this allows us to do the magic-sets transformation on aggregate stratified programs written by the user. We give a perfect model semantics of magical stratified programs and outline an evaluation strategy, Table Queue Evaluation (TQE), that computes the perfect model of a magical stratified program.

Perfect Model

We define the semantics for a single strongly connected component S of a magical stratified program with recursion through grouping. Extension to the full program can be made along the lines of Definition ??.

We use derivation trees for tuples of the magical predicates, m_p, to define an ordering between the tuples. The tuple $m\_p(\mu)$ is in level 1 if it can be derived without using a grouping operation on a predicate of component S. If the tuple $m\_p(\mu)$ can be derived using magical tuples of level n or lower, with any grouping operation being on tuples derived using magical tuples of level $(n-1)$ or lower, the tuple $m\_p(\mu)$ is placed in level n, provided $m\_p(\mu)$ cannot be placed in a lower level.

A magical tuple $m\_p(\mu)$ defines a group $p(\mu)$ for the predicate p that is grouped on. The level of the group $p(\mu)$ is the same as the level of $m\_p(\mu)$. The perfect model of component S can be defined by a prioritized minimization of the groups, with lower levels getting higher priority.

Definition 3.10 "Perfect Model of a Strongly Connected Component of Magical Stratified Programs": Given a strongly connected component S of a magical stratified program P, define the perfect model, $\mathcal{M}$ of S as the minimal model of S that has the following properties:

1. If $\mathcal{M}'$ is another model (minimal or not) of S, then for every group $p(\mu)$ of level 1, the relation for the group $p(\mu)$ in $\mathcal{M}$ is a subset of the relation for the same group in $\mathcal{M}'$.
2. If $\mathcal{M}'$ is another model of S that agrees with $\mathcal{M}$ on all groups of level i and less, then for every group $p(\mu)$ of level $i+1$, the relation for the group $p(\mu)$ in $\mathcal{M}$ is a subset of the relation for the same group in $\mathcal{M}'$.

□

Computational Semantics

One can evaluate the strongly connected components of a magical stratified program in a bottom-up fashion, computing lower components before starting computation of higher components. Evaluation of each strongly connected component is however difficult, since a component can have recursion through grouping, and we want to ensure that a grouping operation is never applied until a full group has been evaluated.

We use the Table Queue Evaluation (TQE, [?]) method to evaluate each component of a magical stratified program. TQE is the standard evaluation strategy in Starburst,[6] and is a natural generalization of the bottom-up technique for components that have recursion through grouping (on components that do not involve recursion through grouping, bottom-up evaluation and TQE are identical). TQE works top-down in the sense that evaluation of relations is demand driven. However no information (bindings/selections) is passed top-down. The basic idea is to delay evaluation of a rule R with a groupby over a relation in the same component until all other rules have been evaluated and no more tuples can be derived without evaluating rule R. Example ?? explained TQE of program M. To see how TQE differs from the usual bottom-up evaluation, let program M be modified by defining r to be the transitive closure of u, so that we have the rules:

$(M3')$: $r(X,Y) \text{ :- } m\_r(X) \ \& \ u(X,Y).$
$(M3'')$: $r(X,Y) \text{ :- } m\_r(X) \ \& \ u(X,Z) \ \& \ r(Z,Y).$

[6] TQE was chosen in Starburst even *before* we discovered its usefulness in computing with magical stratified programs.

$(M7')$: $m\_r(Z) \coloneq m\_r(X) \ \& \ u(X, Z).$

Bottom-up evaluation may compute the groupby on r before all the r tuples in group $X = 5$ are computed. TQE will not apply the groupby on r until the rules $M7'$, $M3'$ and $M3''$ have been iterated on and all r tuples in group $X = 5$ have been computed.

Theorem 3.6 *Given a strongly connected component S of a magical stratified program, Table Queue Evaluation correctly computes the perfect model of S.* □

Proof: By induction on level l of groups in the component S, with the inductive hypothesis that TQE correctly computes the groups of level l. □

3.6 Magic-sets Transformation

The aim of the magic-sets transformation is to push information down into the lower strongly connected components, so that a bottom-up evaluation of the query will be able to use the information normally available to a top-down goal driven evaluator. The groupby subgoal is a second order predicate that cannot use the bindings on rule variables directly, but program evaluation can benefit if the bindings are pushed *"through"* the groupby subgoal into the predicate being grouped upon.

We use the magic-sets transformation to push information through a groupby subgoal into the grouping predicate. The groupby subgoal defines a relation over the grouping and aggregation variables, and the information available for passing down may be over any of these variables. Our magic-sets transformation will only pass down bindings on the grouping variables. We do not attempt to pass down bindings on the aggregation variables. Thus, for the query:

(Q): ?- $A = 5 \ \& \ B = 5 \ \& \ p(A, B).$

$(P1)$: $p(A, B) \coloneq \texttt{group\_by}\,(q(A, C), [A], B = \mathsf{SUM}(C)).$

we want to use only the binding on A to limit computation of q (assuming q is an idb predicate) in order to evaluate p^{bb} by rule $P1$. If we knew that the second attribute of q is a positive integer, we could conceivably use the binding $\mathsf{SUM}(C) = 5$ as an early termination test during the grouping operation; terminating if the partial sum exceeded 5. However, such a use of the binding $\mathsf{SUM}(C) = 5$ is beyond the scope of the magic-sets transformation.

The magic-sets transformation on programs with aggregation and grouping is essentially the same as the transformations discussed in [?, ?]. A slight modification is needed to handle the groupby subgoal — instead of generating magic-sets for the groupby subgoal, we generate magic-sets for the grouping relations, pushing only the bindings on grouping variables. As an example, for the query Q and program P above, magic-sets transformation gives us the program M

(MQ): ?- $A = 5 \ \& \ B = 5 \ \& \ p^{bb}(A, B).$

$(M1)$: $p^{bb}(A, B) \coloneq m\_p^{bb}(A, B) \ \&$
$\qquad \texttt{group\_by}\,(q^{bf}(A, C), [A], [B = \mathsf{SUM}(C)]).$
$(M2)$: $m\_p^{bb}(5, 5).$
$(M3)$: $m\_q^{bf}(A) \coloneq m\_p^{bb}(A, B)).$

with $m\_q^{bf}(A)$ being used as a subgoal in the rules for q^{bf}.

3.6.1 Magic-sets Transformation of Monotonic Programs

Theorem 3.7 *Let P be a monotonic program, and let M be the magic-sets transformation of P. Then M is monotonic.* □

Proof: An original rule r for predicate p gets no new `group_by` subgoal during magic-sets transformation. The magic predicate m_p added to r cannot refer to an aggregation variable, because no aggregation variable appears in the head of r (since r is monotonic). r will thus remain monotonic after magic-sets transformation.

A groupby subgoal G in a new rule r' for a magic predicate m_p must appear in the original rule r where p is used. Since r is monotonic with respect to G, r' must also be monotonic with respect to G. □

3.6.2 Magic-sets Transformation of Aggregate Stratified Programs

It is well known from work on stratified negation and stratified sets ([?]) that the magic-sets transformation of a stratified program may not be stratified. The following theorem allows us to apply the magic-sets transformation to aggregate stratified programs.

Theorem 3.8 *Let P be an aggregate stratified program, and let M be the magic-sets transformation of P. Then M is magical stratified.* □

Proof: The magic predicate m_p for a grouping predicate p serves as the magical predicate for p. The syntactic condition for M to be a magical stratified program is satisfied as the attributes of magic predicate m_p are a subset of the grouping attributes of p. That the semantic condition is satisfied can be proven using the fact that (1) the original program is aggregate stratified, and (2) the magic-sets transformation preserves relevant derivation trees. □

We have an even stronger result:

Theorem 3.9 *Let P be a magical stratified program, and let M be the magic-sets transformation of P. Then M is magical stratified.* □

EXAMPLE 3.4 The program P

$$
\begin{array}{ll}
(P1): & p(X,Y) :- t(X,Y). \\
(P2): & p(X,Y) :- \texttt{group\_by}(\\
 & \qquad r(X,W),[X],[Z = \mathsf{SUM}(W)]) \ \& \\
 & \quad p(Z,Y). \\
(P3): & r(X,Y) :- u(X,Z) \ \& \ v(Z,Y).
\end{array}
$$

is aggregate stratified. Its magic-sets transformation, program M of Example ??, is magical stratified. □

To evaluate an aggregate stratified program P efficiently, we do a magic-sets transformation to get a program M, and evaluate M using TQE.

4 Related Work

In defining the semantics of programs with duplicates, we have used the notion of derivation trees extensively, and we have followed the treatment in [?]. Unlike [?], we do not introduce object-ids, and copies of a tuple are indistinguishable to the user and implementer.

[?] extends relational algebra and calculus with aggregates, and shows that in absence of recursion, duplicates are not needed for expressivity. In working with recursive queries, we have borrowed the idea of stratification, introduced in [?, ?] to deal with negation, to define a class of programs that restricts the use of groupby so as to ensure the existence of a preferred minimal model, in the spirit of the "perfect" model for programs with negation. At the same time we have identified two classes of non-stratified programs that have a perfect model. Monotonicity of programs in absence of stratification has not been discussed before. Magical Stratification is a refinement of the notions of local stratification ([?]) and modular stratification ([?]). Our treatment of grouping and aggregation differs from the approach taken in [?, ?, ?, ?], where constructs such as "set grouping" are used to construct set-valued terms. We have chosen to adopt the SQL approach, in which terms cannot be set-valued since every grouping must be followed by an aggregation.

[?] defines a class of *Closed Semiring* programs that includes program T of Section ??. However, no syntactic or semantic characterization of the closed semiring class is given. An ordering between the groups is not determined, so that a semi-naive evaluation cannot be carried out. The computation is defined through Naive evaluation. The treatment of aggregates in [?] is similar to ours, with differences in the syntax (grouping is specified in the head of a rule) and computational semantics (Naive Evaluation is required). Multiset predicates are not allowed. The EKS system at ECRC ([?]) implements program T of Section ?? by a top-down evaluation.

In adapting the magic-sets approach, we have extended the transformation algorithms developed in [?, ?] to handle both duplicates and aggregates. Several other optimization algorithms for logic programs have been proposed, based on the perfect model semantics of logic programs. Unfortunately, none of these optimizations is likely to be directly applicable to programs with multiset predicates and aggregate operations. An important area for further research is to identify under what conditions these optimizations can be adapted to such programs, and to develop optimization techniques tailored to such programs. [?] gives a magic-sets algorithm for modularly stratified (that includes aggregate stratified) programs, but the transformed program is not modularly stratified, and several meta-predicates as well as the concept of iterations are built into the magic program.

The ground magic-sets transformation of [?] extends magic-sets to push conditions as well as bindings. [?] uses the results of this current paper to (1) adapt the ground magic-sets algorithm to work in presence of duplicates, and (2) to develop a magic-sets transformation that works in SQL based relational systems.

5 Acknowledgements

Yehoshua Sagiv made the suggestion that we look at the number of proof trees for an atom in defining the duplicate semantics. We have benefited from comments and discussions with Mariano Consens, Shel Finkelstein, Ashish Gupta, Håkan Jakobsson, Michael Kifer, Alberto Mendelzon, Ken Ross, Yehoshua Sagiv, Ulf Schreier, Jeff Ullman and Moshe Vardi. We thank the referees for their insightful comments. The Starburst project at IBM Almaden Research Center and the *NAIL!* project at Stanford University provided a stimulating environment for this work.

References

[ABC+76] M. Astrahan, M. Blasgen, D. Chamberlin, K. Eswaran, J. Gray, P. Griffiths, W. King, R. Lorie, P. McJones, J. Mehl, G. Putzolu, I. Traiger, B. Wade, and V. Watson. `System R: Relational approach to Database Management`. *ACM Transactions on Database Systems*, 1(2), June 1976.

[ABW88] K. Apt, H. Blair, and A. Walker. `Towards a Theory of Declarative Knowledge`. In J. Minker, editor, *Foundations of Deductive Databases and Logic Programming*, pages 89–148, Washington D.C., 1988. Morgan Kaufmann.

[BMSU86] Francois Bancilhon, David Maier, Yehoshua Sagiv, and Jeffrey D. Ullman. `Magic Sets and other Strange Ways to Implement Logic Programs`. In *Proceedings of the Fifth Symposium on Principles of Database Systems*

(PODS), Cambridge, MA, pages 1–15. March 1986.

[BNR+87] Catriel Beeri, Shamim Naqvi, Raghu Ramakrishnan, Oded Shmueli, and Shalom Tsur. `Sets and Negation in a Logic Database Language (LDL1)`. In *Proceedings of the Sixth Symposium on Principles of Database Systems (PODS), San Diego, CA*, pages 21–37. March 1987.

[BR87] Catriel Beeri and Raghu Ramakrishnan. `On the Power of Magic`. In *Proceedings of the Sixth Symposium on Principles of Database Systems (PODS), San Diego, CA*, pages 269–283. March 1987.

[CH85] Ashok Chandra and D. Harel. `Horn Clause Queries and Generalizations`. *Journal of Logic Programming*, 2(1):1–15, 1985.

[CKW89] Weidong Chen, Michael Kifer, and David S. Warren. `HiLog: A First Order Semantics for Higher-Order Logic Programming Constructs`. In *Second International Workshop on Database Programming Languages*. Morgan-Kaufmann, San Mateo, CA, June 1989.

[CM90] Mariano P. Consens and Alberto O. Mendelzon. `Low Complexity Aggregation in Graphlog and Datalog`. *Unpublished Manuscript*, 1990.

[ISO89] ISO_ANSI. `ISO-ANSI Working Draft: Database Language SQL2 and SQL3; X3H2; ISO/IEC JTC1/SC21/WG3`, 1989.

[Klu82] A. Klug. `Equivalence of Relational Algebra and Relational Calculus Query Languages Having Aggregate Functions`. *Journal of the ACM*, 29(3):699–717, July 1982.

[KM89] Michael Kifer and Inderpal Singh Mumick. `Personal Communications`, December 1989.

[Kup87] Gabriel M. Kuper. `Logic Programming with Sets`. In *Proceedings of the Sixth Symposium on Principles of Database Systems (PODS), San Diego, CA*, pages 11–20. March 1987.

[KW89] Michael Kifer and James Wu. `A Logic for Object-Oriented Logic Programming`. In *Proceedings of the Eighth Symposium on Principles of Database Systems (PODS), Philadelphia, PA*, 1989.

[MFPR90a] Inderpal Singh Mumick, Sheldon J. Finkelstein, Hamid Pirahesh, and Raghu Ramakrishnan. `Magic Conditions`. In *Proceedings of the Ninth Symposium on Principles of Database Systems (PODS), Nashville, TN*, pages 314–330. April 2-4 1990.

[MFPR90b] Inderpal Singh Mumick, Sheldon J. Finkelstein, Hamid Pirahesh, and Raghu Ramakrishnan. `Magic is Relevant`. In *Proceedings of ACM SIGMOD 1990 International Conference on Management of Data, Atlantic City, NJ*, pages 247–258, May 23-25 1990.

[MP90] John Mcpherson and Hamid Pirahesh. `Table Queue Evaluation Strategy`. Research report, to be published, IBM Research Division, Computer Science, Almaden Research Center, San Jose, California 95120-6099, 1990.

[MR89] Michael Maher and Raghu Ramakrishnan. `DéjàVu in Fixpoints of Logic Programs`. In *North American Conference on Logic Programming (NACLP), Cleveland, Ohio*, October 16-20 1989.

[MUVG86] Katherine A. Morris, Jeffrey D. Ullman, and Allen Van Gelder. `Design Overview of the NAIL! System`. In *Logic Programming: Proceedings of the Third International Conference, London*, pages 554–568, 1986.

[NT88] Shamim Naqvi and Shalom Tsur. `A Logic Language for Data and Knowledge Bases`. Computer Science Press, 1988.

[OOM87] G. Özsoyoğlu, M. Özsoyoğlu, and V. Matos. `Extending Relational Algebra and Relational Calculus with Set-Valued Attributes and Aggregate Functions`. *ACM Transactions on Database Systems*, 12(4):566–592, December 1987.

[Pir89] Hamid Pirahesh. `Early experience with rule-based query rewrite optimization`. In *Optimization workshop, SIGMOD*, May 1989.

[Prz88] T. C. Przymusinski. `On the declarative semantics of deductive databases and logic programs`. In J. Minker, editor, *Foundations of Deductive Databases and Logic Programming*, pages 89–148, Washington D.C., 1988. Morgan Kaufmann.

[Ros90] Kenneth A. Ross. `Modular Stratification and Magic Sets for Datalog Programs with Negation`. In *Proceedings of the Ninth Symposium on Principles of Database Systems (PODS), Nashville, TN*, pages 161–171. April 2-4 1990.

[Ull88] Jeffrey D. Ullman. `Principles of Database and Knowledge-Base Systems, Volume 1`. Computer Science Press, 1988.

[Ull89] Jeffrey D. Ullman. `Principles of Database and Knowledge-Base Systems, Volume 2`. Computer Science Press, 1989.

[VBKL90] L. Vieille, P. Bayer, V. Küchenhoff, and A. Lefebvre. `EKS-VI, A Short Overview`. *Unpublished Manuscript, distributed at SIGMOD 90 Technical Exhibition*, May 23-25 1990.