The Magic of Duplicates and Aggregates

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 95193, 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 ([ABC+76]), 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 ([Pir89]).

1.1 Duplicates† (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 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

---

*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.

Permission to copy without fee all or part of this material is granted provided that the copies are not made or distributed for direct commercial advantage, the VLDB copyright notice and the title of the publication and its date appear, and notice is given that copying is by permission of the Very Large Data Base Endowment. To copy otherwise, or to republish, requires a fee and/or special permission from the Endowment.

Proceedings of the 16th VLDB Conference  
Brisbane, Australia 1990

---

1Here, and in the rest of the paper, we use duplicates to mean multiple copies, not necessarily two copies.
Semi-Naive — but not Naive! — fixpoint evaluation provides an equivalent operational semantics.

3. An important contribution is the adaptation of the Magic-Sets approach ([BMSU86, BR87]) 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 L.D.I. ([BNR+87]) and HiLog ([CKW89]). 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 ([KM89]). 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-union.

However, as with previous approaches to aggregate operations ([BNR+87, Kup87]), 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 ([Ros90]). 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 [MFPR90a] (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 ([MFPR90b]). As is demonstrated through performance results in [MFPR90b], 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 \textit{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 3.3), though the truth of this conjecture is not critical to our motivation. If we treat \textit{subpart} and \textit{contains} as multisets, then under the semantics that we propose, the program

\begin{verbatim}
contains(P, S) :- subpart(P, T) & contains(T, S).
\end{verbatim}

computes the correct answer. The last rule is interpreted as follows: Group together all tuples of \textit{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 \textit{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 [MFPR90b], we would write

\begin{verbatim}
CREATE VIEW contains(P, S) AS
(SELECT P, S FROM subpart) UNION
(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.
\end{verbatim}

This paper is organized as follows. Section 1.4 gives definitions related to multisets. Duplicates are introduced in Section 2, and Section 3 discusses grouping and aggregation. Related work is presented in Section 4. 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 \textit{col}). We follow [MR89]; 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 \textit{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 $M$, $set(M)$ is the set of elements of $M$, that is, $set(M) = \{x \mid x \in M\}$. 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 $C$ 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 \textit{colored set}, called $col(M)$.

The inverse operation, $col^{-1}(S)$, 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 $a_i$, $col^{-1}(a_i) = a$.

Definition 1.3 “Multiset Constructor $[\ldots]$”: 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 \textit{Select}, \textit{Project} and \textit{Join} operators that are normally defined for sets. We use the symbols $\pi^{set}$, $\pi^{mul}$, $\pi^{set}$, $\pi^{mul}$ 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 $\pi_i R$”: Let $R$ be a multiset of $n$-tuples, and let $i$ be a vector of $k$ integers in the range $1 \ldots n$. For every $n$-tuple $t \in R$, consider the $k$-tuple $(t_{i1}, \ldots, t_{ik})$. The multiset $M$, denoted by $\pi_i R$, is defined to contain every such $k$-tuple $m$, with a multiplicity equal to the number of tuples $t$ of $R$ such that $m = (t_{i1}, \ldots, t_{ik})$. □

$\pi^{mul}$ 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 \textit{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 \textit{aggregates} or \textit{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.

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 ([ISO89]), and implementations of the relational model have extended the model to allow for duplicates. The System R prototype also dealt with duplicates ([ABC+76]). Amongst the prototype implementations of the logic data model based on Datalog type languages, NAIL! ([MUG86]), and LDL ([N'88]) 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 root of 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 2.1 "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 col(Q) generates a derivation tree for col(Q(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 :- b1, b2, ..., bk), k > 0, and for each tuple (t1, t2, ..., tk) of derivation trees for atoms (d1, d2, ..., dk) that unify with (b1, b2, ..., bk) with ð as the mgu, generate a derivation tree for hð, with hð as the root label, and (t1, t2, ..., tk) as the children, in that order, and ð 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, atoms(t) is either the label or the col(P) of the label on the root of tree t. For a set of derivation trees T, atoms(T) is the multiset of atoms that includes, for every tree t ∈ T, a copy of the atom atoms(t).

If a = 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 ⊆ 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' ⊃ D that is a subset of DT(P, E) and yet satisfies Condition 1.

We define the predicate 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 1 ensures that a unique tree represents each atom of a set predicate, and Condition 2 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.

2. Yes, bills at IBM indicate the average cost!
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 $\text{dcp}(D, P, E)$. Further, if $D_1$ and $D_2$ are two distinct sets such that $\text{dcp}(D_1, P, E) \land \text{dcp}(D_2, P, E)$, then $\text{atoms}(D_1) = \text{atoms}(D_2)$. □

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

1. $\text{dcp}(D^{(h)}, P, E)$ (we modify Definition 2.3 of $\text{dcp}$ 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 $\text{dcp}(D^{(h)}, P, E)$, then $\text{atoms}(D^{(h)}) = \text{atoms}(D^{(h)})$. □

**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 $ds(P, E) = \text{atoms}(D)$. □

**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 ([MR89]) evaluation techniques, the former working on set predicates, and the latter working, in parallel, on multiset predicates.

**Definition 2.6** "Rule Application":

\begin{align*}
\text{Incr-Eval}(p, P, D, \Delta) &= \pi^{\text{new}}_{\text{new}}\{((h, r, d_1, d_2, \ldots, d_k) | \text{ } \text{r is a rule for predicate p in program P of the form } h ::= b_1, b_2, \ldots, b_k, k > 0, \text{ and } d_i, d_2, \ldots, d_k) \in \text{col}(D \cup \Delta) \wedge (\text{the mgu of } (b_1, b_2, \ldots, b_k) \text{ and } (d_1, d_2, \ldots, d_k)) \text{ ]}.
\end{align*}

The above formula is read as follows: For every rule $r$ for predicate $p$, and for every tuple $(d_1, d_2, \ldots, d_k)$ of elements of $\text{col}(D \cup \Delta)$ that unifies with the subgoals of rule $r$ in that order, such that at least one of the $d_i$s also appears in $\text{col}(\Delta)$, add $h \theta$ to the result, where $\theta$ 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 $\text{col}(U \cup \Delta)$.

For a set $T$ of predicates, define an iteration as

\begin{align*}
\text{Incr-Eval}(T, P, D, \Delta) &= \bigcup_{p \in T} \text{Incr-Eval}(p, P, D, \Delta).
\end{align*}

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 $M$ and $S$ denote their respective extensions. $D = M \cup S$ is the database. Given a program $P$ and an edb $E$, DSN generates a multiset $D = \text{dsn}(P, E)$.

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

1. $D_0 = \emptyset$; $\delta D_0 = E$.
2. $S_{n+1} = S_n \cup \delta S_n$; $M_{n+1} = M_n \cup \delta M_n$.
3. $\delta S_{n+1} = \text{set}(\text{Incr-Eval}(P_S, P, D_n, \delta D_n)) - S_{n+1}$; $\delta M_{n+1} = \text{Incr-Eval}(P_M, P, D_n, \delta D_n)$.
4. $D_n+1 = [S_{n+1}] \cup M_{n+1}$; $\delta D_{n+1} = [\delta S_{n+1}] \cup \delta M_{n+1}$.
5. $\text{dsn}(P, E) = \Delta = \lim_{n \to \infty} D_n$. □

DSN terminates (at Step $n + 1$) when $D_n+1 = D_n$. 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 $\text{Incr-Tree}$ that does incremental computation on derivation trees, mimicking the computation of $\text{Incr-Eval}$ 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, $\text{dsn}(P, E) = \text{dsn}(P, F)$. □

**DSN evaluation without coloring**

The definition of $\text{Incr-Eval}$ 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 [UL89] to multisets.

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

**Definition 2.8** "ATOV ([UL89])": Given the goal $q(t_1, \ldots, t_n)$ and a relation $Q$ for predicate $q$, define a relation $Q'$ over the variables $X_1, \ldots, X_n$ appearing in $t_1, \ldots, t_n$ as follows:

---

3Note: Each $d_i$ is an atom, a tuple of a relation.
for each tuple \( q(s_1, \ldots, s_m) \) in \( Q \) do
  if there is a term matching \( \tau \) for tuple \( q(s_1, \ldots, s_m) \)
    and goal \( q(t_1, \ldots, t_m) \) then
    add to \( Q' \) the tuple \((\tau(X_1), \ldots, \tau(X_n))\)

Let \( \text{ATOV}(q(\bar{t}), Q) = Q' \). □

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

**Definition 2.9 “\text{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 \( \Delta \). As before, we assume that the incremental relations have not been included in the main database.

Let \( D'_1 = \text{ATOV}(b_1, D_1), \Delta'_1 = \text{ATOV}(b_1, \Delta_1) \), and let \( \delta_j \) be given by either of the following two equivalent joins:

\[
\begin{align*}
D'_1 \land \ldots \land D'_{j-1} \land \Delta'_j \land (D'_{j+1} \cup \Delta'_{j+1}) \land \ldots \land (D'_k \cup \Delta'_k)
\end{align*}
\]

or

\[
\begin{align*}
(D'_1 \cup \Delta'_1) \land \ldots \land (D'_{j-1} \cup \Delta'_{j-1}) \land \Delta'_j \land D'_{j+1} \land \ldots \land D'_k
\end{align*}
\]

Define

\[
\text{Incr\_Eval}\_2(p, r, D, \Delta) = \text{VTOA}(h, \bigcup_{1 \leq j \leq k} \delta_j)
\]

\[
\text{Incr\_Eval}\_2(p, P, D, \Delta) = \bigcup_r \text{Incr\_Eval}\_2(p, r, D, \Delta)
\]

□

Let \( \text{DSN}\_2 \) be the version of the DSN algorithm using \( \text{Incr\_Eval}\_2 \) instead of \( \text{Incr\_Eval} \).

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

By building common subexpressions carefully, \( \text{Incr\_Eval}\_2 \) 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 [Ull1989] for an introduction to adornments). More complex adornments are discussed in [MFPR90a] whatsoever, preserves the duplicate semantics of a program. We then show that a magic-sets transformation ([BMSU86, BR87, Ull1989]) 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 \( p^a \) of the same type as \( p \), using the rules for predicate \( p \) in the program \( P \). \( p^a \) has exactly the same rules as \( p \), except that the head of these rules are for predicate \( p^a \) rather than for \( p \). \( p^a \) 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 \( p^{a_1} \) and \( p^{a_2} \) are members of the same predicate class in \( P \), then the relations for \( p^{a_1} \) and \( p^{a_2} \) in \( \text{ds}(P, E) \), 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 \( p^{a_1}(t) \) by \( p^{a_2}(t) \) where \( p^{a_1} \) and \( p^{a_2} \) 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 2.4. (Programs such as \( P' \) are called adorned programs). Let predicate \( p \) appear in both \( P' \) and \( P \). Then the relation for \( p \) in \( \text{ds}(P, E) \) is identical to the relation for \( p \) in \( \text{ds}(P', E) \). □

#### 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 \( \text{magred}(t') \) 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 \( \text{dcp}(D, P, E) \) 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 \( \text{dcp}(D'', P'', E) \) 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 \( \text{magred}(t'') = t \), OR
2. For every tree \( t' \) in \( D'' \), there is a derivation tree \( t \) in \( D \) such that \( \text{magred}(t') = t \).

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 [MR89], 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 \( \text{irrset} \) is a set in which no element subserves another.

The presence of non-ground tuples raises the possibility that a tuple may be subserved by another that is generated in a later iteration. This seriously complicates the computation, and we require stratification with respect to the use of \( \text{irrset} \) predicates. We can extend the declarative and computational semantics to \( \text{irrset} \)-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 \textit{average} and \textit{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 ([BNR+87]) 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 extension 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, \texttt{group.by}(r(\bar{t}), GL, AL), that takes as arguments a goal \( r \) with its attribute list \( GL \), a grouping list \( GL \) of variables appearing in \( t \), and an aggregation list \( AL \) of aggregate functions. The general form of a \texttt{group.by} subgoal is:

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

The predicate \( r \) is called the grouping predicate. The arguments, \( \bar{t} \) of \( r \) can be general terms: constants, variables, or complex terms. However, any variables in \( \bar{t} \) are local to the \texttt{group.by}, unless they also appear in the grouping list. The grouping list consists of zero or more distinct variables that must appear in \( \bar{t} \). The list of \( Z \)'s is called the aggregation list. Each \( Z \) is a new variable, \( E \) is an expression that uses variables of \( \bar{t} \), \( S \) is optional — the keyword \texttt{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 \textit{SUM}, \textit{CNT}). Within the rule body, a \texttt{group.by} subgoal represents a relation over variables \( Y \) in the grouping list and variables \( Z \) 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 \( \text{set} \) 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 \texttt{group.by} 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 \texttt{group.by} Subgoal

An ordinary subgoal \( p(\bar{t}) \) of a rule defines an ATOV mapping from a relation \( P \) for predicate \( p \) to a relation over the variables in \( \bar{t} \). 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 \( p(\bar{t}) \), checking satisfiability for a substitution \( \sigma \) simply involves checking whether the tuple \( p(\bar{t})\sigma \) 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 ATOV(p(\bar{t}), P) mapping, and checking whether the tuple corresponding to the substitution \( \sigma \) is in the mapping.

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

\textbf{Definition 3.1 “ATOV of \texttt{group.by}”:}

\[
\text{ATOV}([r(\bar{t})], [Y_1, Y_2, \ldots, Y_m], [Z_1 = A_1(S_1(E_1)), \ldots, Z_n = A_n(S_n(E_n))], R)
\]

\[= T, \]

where \( T \) is defined as follows:
1. Let $R' = \text{ATOV}(r(\bar{f}), R)$. $R'$ is a relation over the variables in $\bar{f}$.

2. Let $G = \pi_{Y_1, Y_2, \ldots, Y_m}(R')$. We use a set projection in this step even if multisets are present. If $m = \emptyset$, $G$ is a relation with no attributes, having either a single empty tuple (if $R$ is not empty), or no tuples (if $R$ is empty).\footnote{SQL does not permit $m = \emptyset$ when using \text{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 $m = \emptyset$ 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 $m = \emptyset$ 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 $m = \emptyset$ and $R = \emptyset$. Special boundary cases in some of the properties and theorems will be required as a consequence.}

3. For each tuple $\mu$ in $G$, define a tuple $f(\mu)$ as follows:
   - Let $R'_\mu = R' \bowtie \mu$. $R'_\mu$ is thus the maximal subset of $R'$ having the same values for the attributes $Y_1, Y_2, \ldots, Y_m$ as the tuple $\mu$. If $m = \emptyset$, $\mu$ 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 $S_i = \emptyset$, set $R'_\mu.i = \emptyset$.
   - Compute $Z_1 = A_1(R'_\mu.1)$, $Z_2 = A_1(R'_\mu.2)$, $\ldots$, $Z_n = A_1(R'_\mu.n)$. 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.

\[ \text{Theorem 3.1 ATOV of a group-by subgoal is non-monotonic.} \]

\[ \text{Proof: Adding a tuple to the grouping relation } R \text{ can change the number of tuples in a group (} R'_\mu \text{). As a result, an aggregate value previously derived may no longer be derivable.} \]

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

Definition 3.3 "groupset": The groupset of a relation with respect to a grouping list $\bar{Y}$, written as $\text{groupset}(R(\bar{X}, \bar{Y}), \bar{y})$, is defined to be the set of groups $\{ \text{group}(R(\bar{X}, \bar{Y}), \bar{y}) \mid \bar{y} \in \pi_{\bar{Y}}(R) \}$.\]

Several properties exist between groups and relations $R'$, $R'_\mu$, and $T$ of Definition 3.1. group$(R', \bar{Y}, \mu) = R'_\mu$. Each group $g$ in groupset$(R', \bar{Y})$ contributes exactly one tuple to $T$, so that the number of tuples in $T$ is equal to the cardinality of groupset$(R', \bar{Y})$. We can then derive the following important theorem.

\[ \text{Theorem 3.2 ATOV of a group-by subgoal is monotonic with respect to new groups. That is, for} \]

the group-by subgoal $G$, if $R_2 = R_1 \cup \Delta$, and groupset$(R'_2, \bar{Y}) = \text{groupset}(R'_1, \bar{Y}) \cup \text{groupset}(\Delta, \bar{Y})$, then $T_2 \supseteq T_1$. \]

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 $\sigma$ satisfies the group-by subgoal $G$

\[ (G): \text{group-by}(r(\bar{f}), [Y_1, Y_2, \ldots, Y_m], \{ Z_1 = A_1(S_1(E_1)), \ldots, Z_n = A_n(S_n(E_n)) \}). \]

if the tuple $(\bar{F}\sigma, \bar{Z}\sigma)$ is in

\[ \text{ATOV(group-by}(r(\bar{f}), \bar{Y}), [\bar{Z} = \bar{A}(\bar{E}), R]), \]

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. Aggregative Stratification is similar to Stratified Negation ([Ull88]), 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 ([Ull88]). The semantics of an aggregate stratified program is given by an intuitive perfect model, similar to the perfect model of a program with stratification negation.

**Definition 3.5 “Perfect Model of Aggregate Stratified Programs”:** Given an aggregate stratified program \(P\), define its perfect model, \(M\), as the minimal model of \(P\) that has the following properties:

1. If \(M'\) is another model (minimal or non) of \(P\), then for every predicate \(p\) of stratum 1, the relation for \(p\) in \(M\) is a subset of the relation for \(p\) in \(M'\).

2. If \(M'\) is another model of \(P\) that agrees with \(M\) on all predicates of stratum 1 and less, then for every predicate \(p\) of stratum \(i + 1\), the relation for \(p\) in \(M\) is a subset of the relation for \(p\) in \(M'\).

\(\Box\)

**Theorem 3.3** Every aggregate stratified program has a unique perfect model. \(\Box\)

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

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 3.2), 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 1.1 without using multisets:

\((T1):\) contains\((P, S, \text{null}, C) \rightarrow \text{subpart}\((P, S, C)\).

\((T2):\) contains\((P, S, U, C) \rightarrow \text{subpart}\((P, U, C_1) \&\) count\(\text{contains}(U, S, C_2) \& C = C_1 \ast C_2\).

\((T3):\) count\(\text{contains}(P, S, C) \rightarrow \text{group_by}(\)

\(\text{contains}(P, S, U, C),\) \([P, S], [C = SUM(M)]\)).

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 subpart relation is acyclic, program \(T\) is group stratified, with an ordering between the \((P, S)\) groups of contains defined by a topological sort on the 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 \(\text{set_owns}(C_m, C_s, S)\) 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

\((C1): \) all\(\text{controls}(C_m, C_s) \rightarrow \text{set_owns}(C_m, C_s, S),\)

\((C2): \) all\(\text{controls}(C_m, C_s) \rightarrow \text{set_bought}(C_m, C_s)\)

\& \(C_m \neq C_s \& \text{all\_controls}(C_s, C_s, S).

\((C3): \) \text{set_bought}(C_m, C_s) \rightarrow \text{group_by}(\)

\(\text{all\_controls}(C_m, C_s), [C_m, C_s], [A = SUM(S)]\))

\& \(A > 50\).

\(\Box\)

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 SQL-Log 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 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. \(\Box\)

**Definition 3.7 “Monotonic Program”:** A program is monotonic if every rule in it is monotonic. \(\Box\)

Clearly, a rule without a 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 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 \( \text{true} \) to \( \text{false} \), 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 \((S = \text{SUM}, R^+)\), where \( R^+ \) is the domain of positive reals. \( S > c \) is not monotonic with respect to \((S = \text{SUM}, R), \) where \( R \) is the domain of all reals.

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

**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 3.4, the Corporate Takeover example is monotonic, assuming that the domain of \( S \) in the relation \text{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 \((S = \text{SUM}, R^+) \).

**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 group by 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)**

\[
\begin{align*}
(M1): & \quad p(X, Y) :- m_p(X) \land t(X, Y). \\
(M2): & \quad p(X, Y) :- m_p(X) \land \text{group by}(r(X, W), [X], [Z = \text{SUM}(W)]) \land p(Z, Y). \\
(M3): & \quad r(X, Y) :- m_r(X) \land u(X, Z) \land v(Z, Y). \\
(M4): & \quad m_p(Z) :- m_p(X) \land \text{group by}(r(X, W), [X], [Z = \text{SUM}(W)]). \\
(M5): & \quad m_p(5). \\
(M6): & \quad m_r(X) :- m_p(X).
\end{align*}
\]

The dependencies \( m_r \leftarrow M_p \rightarrow m_r \rightarrow r \) make \( r \) and \( m_p \) mutually recursive. Thus, program \( P \) has recursion through grouping and is not aggregate stratified. Program \( P \) is not even monotonic; the rules \( M2 \) and \( M4 \) are not monotonic.

Program \( P \) 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. □

[Ros90] 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 $\text{group by } (p(t), [\overline{y}], \{z = A(E)\})$ 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 groupby edge, or it satisfies the following property:

In the dependency graph of component $S$, let there be a cycle with a groupby edge $p \xrightarrow{gb} t$, 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 $m.p$ (sex. $m.p$, 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 $\mu$ 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 3.3 is magical stratified. The predicates $r$, $m.p$, and $m.r$ are in one strongly connected component, and $r \xrightarrow{gb} m.p$ 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 3.6, 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 3.5.

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, $M$ of $S$ as the minimal model of $S$ that has the following properties:

1. If $M'$ 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 $M$ is a subset of the relation for the same group in $M'$.
2. If $M'$ is another model of $S$ that agrees with $M$ 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 $M$ is a subset of the relation for the same group in $M'$.

□

**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, [MP90]) method to evaluate each component of a magical stratified program. TQE is the standard evaluation strategy in Starburst, 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 3.3 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:

$$\begin{align*}
(M3') & : r(X, Y) :- m_r(X) & u(X, Y). \\
(M3'') & : r(X, Y) :- m_r(X) & u(X, Z) & r(Z, Y).
\end{align*}$$

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:

$$\begin{align*}
(Q) & : A = 5 & B = 5 & p(A, B).
\end{align*}$$

we want to use only the binding on $A$ to limit computation of $q$ (assuming $q$ is an idb predicate) in order to evaluate $p(b)$ by rule $P1$. If we knew that the second attribute of $q$ is a positive integer, we could conceivably use the binding $\text{SUM}(C) = 5$ as an early termination test during the grouping operation; terminating if the partial sum exceeded 5. However, such a use of the binding $\text{SUM}(C) = 5$ 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 [BR87, MFPR90a]. 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 transformation gives us the program $M$

$$\begin{align*}
(M1) & : p(b)(A, B) :- m_{p(b)}(A, B) \\
       & & \text{group-by}(q_{(A, C)}, [A, B = \text{SUM}(C)]).
(M2) & : m_{q_{(a)}}(10, 10).
(M3) & : r_{(b)}(A) :- m_{p(b)}(A, B))
\end{align*}$$

with $m_{A}(A)$ being used as a subgoal in the rules for $q_{(a)}$.

#### 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 group 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 ([BNR+87]) 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. □
stratified programs that have a perfect model. Mono
grouping must be followed by an aggregation.
that includes program
tonicity of programs in absence of stratification has not
the same time we have identified two classes of non-
stricts the use of groupby so as to ensure the exis-
been discussed before. Magical Stratification is a re
finement of the notions of local stratification ([Prz88])
with recursive queries, we have borrowed the idea of
aggregates, and shows that in absence of recursion, du-
and we have followed the treatment in [MR89]. Un-
notion of derivation trees extensively,
3.4 The program P

(P1): \(^p(X, Y) \leftarrow r(X, Y).\)
(P2): \(^p(X, Y) \leftarrow \text{group}\_\text{by}(\)
\(r(X, W), [X], [Z = \text{SUM}(W)]) \&
\(p(Z, Y).\)
(P3): \(^r(X, Y) \leftarrow u(X, Z) \& u(Z, Y).\)

is aggregate stratified. Its magic-sets transformation, pro-
gram \(M\) of Example 3.3, is magical stratified. □

To evaluate an aggregate stratified program \(P\) effi-
ciently, we do a magic-sets transformation to get a pro-
gram \(M\), 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 [MR89]. Un-
lke [KW89], we do not introduce object-ids, and copies
of a tuple are indistinguishable to the user and imple-
menter.

[Klu82] extends relational algebra and calculus with
aggregates, and shows that in absence of recursion, du-
plicates are not needed for expressivity. In working
with recursive queries, we have borrowed the idea of
stratification, introduced in [CH85, ABW88] to deal
with negation, to define a class of programs that re-
stricts the use of groupby so as to ensure the exist-
ence 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. Mono-
tonicity of programs in absence of stratification has not
been discussed before. Magical Stratification is a re-
finement of the notions of local stratification ([Prz88])
and modular stratification ([Ros90]). Our treatment
of grouping and aggregation differs from the approach
taken in [OOM87, BNR+87, Kup87, CKW89], 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.

[CM90] defines a class of Closed Semiring programs
that includes program \(T\) of Section 3.3.1. 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 [CM90] is similar to ours, with differences in the syn-
tax (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 ([VBKL00]) implements program \(T\) of Sec-
section 3.3.1 by a top-down evaluation.

In adapting the Magic-Sets approach, we have
extended the transformation algorithms developed in
[BMSU86, BR87] to handle both duplicates and ag-
gregates. Several other optimization algorithms for logic
programs have been proposed, based on the perfect
model semantics of logic programs. Unfortunately,
one of these optimizations is likely to be directly ap-
licable to programs with multiset predicates and ag-
gregate operations. An important area for further
research is to identify under what conditions these op-
timizations can be adapted to such programs, and to
develop optimization techniques tailored to such pro-
grams. [Ros90] gives a magic-sets algorithm for modu-
larly stratified (that includes aggregate stratified) pro-
grams, 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 [MFPR90a]
extends magic-sets to push conditions as well as bind-
ings. [MFPR90b] 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 environ-
ment for this work.

References

[ARC+76] M. Astrahan, M. Blasgen, D. Chamberlin,
K. Eswaran, J. Gray, P. Griffiths, W. King,
R. Lorie, P. McJones, J. Mehli, G. Put-
zolu, I. Traeger, B. Wade, and V. Wat-
son. System R: Relational approach to
Database Management. ACM Transactions on

a Theory of Declarative Knowledge. In


