Aggregation and Relevance in Deductive Databases*

S. Sudarshan **Raghu Ramakrishnan**†

Computer Sciences Department,
University of Wisconsin-Madison, WI 53706, U.S.A.

Abstract

In this paper we present a technique to optimize queries on deductive databases that use aggregate operations such as *min*, *max*, and "largest k values." Our approach is based on an extended notion of relevance of facts to queries that takes aggregate operations into account. The approach has two parts: a rewriting part that labels predicates with "aggregate selections," and an evaluation part that makes use of "aggregate selections" to detect that facts are irrelevant and discards them. The rewriting complements standard rewriting algorithms like Magic sets, and the evaluation essentially refines Semi-Naive evaluation.

1 Introduction

Recursive queries with aggregation have been considered by several people [BNR+87, MPR90]. The advantages of a rich language are clear, but unless effective optimization techniques are developed, the performance of specialized systems based on supporting a limited class of queries (for example generalized transitive closure queries) cannot be matched. In this paper we consider optimizations of recursive programs with aggregate operations.

Consider the (very naive) program shown in Figure 1, for computing shortest paths between nodes in the relation *edge*. It essentially enumerates all paths and chooses shortest paths among them. The notation $path(X, Y, min(< C >))$ in the head of rule $R2$ denotes that for each value of X, Y all possible C values that are generated by the body of the rule are collected in a set, and the *min* aggregate operation is applied on this set of values. For each value of X and Y, a *path* fact is created with the result of the *min* operation as the third argument.

A formulation of the problem in this form is desirable since it is declarative, can be queried in many different ways and is easy to write. It is easily augmented with additional constraints such as "the edges all have a given label" (for instance, flights on United Airlines alone must be considered), or "there must be no more than three hops on the flight". The standard bottom-up evaluation of such a program is extremely inefficient since it constructs every possible path in the graph. In contrast, the above problem can be solved in polynomial time using either Warshall's algorithm or Dijkstra's shortest path algorithm (see [AHU74]). It can also be evaluated efficiently if it is expressed using specialized operators for transitive closure ([RHDM86, ADJ88, CN89]).

We propose to optimize bottom-up evaluation using a notion of relevance of facts to some aggregate operations such as *min* and *max*. Our notion of relevance can be seen as an extension of the notion of relevance used in optimizations such as Magic sets rewriting [BMSU86, BR87, Ram88]. We first explain the idea informally, using Program Simple (Figure 1).

Example 1.1 Consider Program Simple (Figure 1)[1]. Aggregate operation *min* has the property that non-minimal values in a set are unnecessary for the aggregate operation on the set. Using this property, we can deduce that a fact $path(a, b, p1, c1)$ is relevant to the rule defining the query predicate *shortest_path* only if there is no fact $path(a, b, p2, c2)$ such that $c2 < c1$. We use tests called *aggregate selections* to check whether a fact is relevant; conditions such as the above are used in the tests.

The rewriting (automatically) deduces an aggregate selection on this occurrence of the predicate *path*; only facts with minimum cost values satisfy the aggregate selection. It then "pushes" this aggregate selection into rules that define *path*, and propagates the selections through the program.

The rewriting algorithm outputs a program containing aggregate selections on the predicates. In this case the output

*This paper appeared in the Int'l Conference on Very Large Databases, 1991

†The work of both authors was supported in part by a David and Lucile Packard Foundation Fellowship in Science and Engineering, an IBM Faculty Development Award and NSF grant IRI-8804319. The email addresses of the authors are {sudarsha,raghu}@cs.wisc.edu

[1] We assume that *append* is defined for us, and concentrate on the rest of the program.

$R1: shortest\_path(X,Y,P,C) \leftarrow s\_p\_length(X,Y,C), path(X,Y,P,C).$
$R2: s\_p\_length(X,Y,min(<C>)) \leftarrow path(X,Y,P,C).$
$R3: path(X,Y,P1,C1) \leftarrow path(X,Z,P,C), edge(Z,Y,EC),$
$append([edge(Z,Y)|nil],P,P1), C1 = C + EC.$
$R4: path_1(X,Y,[edge(X,Y)|nil],C) \leftarrow edge(X,Y,C).$
Query: ?-$s\_p(X,Y,P,C)$.

Figure 1: Program Simple

is essentially the same as Program Simple, except that every occurrence of *path* in the program has an aggregate selection that selects minimum cost paths. The rewritten program is shown in Figure 2, and we discuss it after introducing the notation used to express aggregate selections.

The evaluation phase of our technique makes use of the aggregate selections on *path*, and discards facts on which the aggregate selection test fails (namely the non-minimal paths). We can optimize the evaluation further by using in each iteration only the path fact with minimum cost among all newly generated *path* facts. This reduces the cost to the same as that of Dijkstra's algorithm ($O(e * log(n))$), and this is discussed in Section 5.2. The optimized evaluation also works when edge weights are negative, so long as there are no negative cost cycles. □

Recently Ganguly et al. [GGZ91] independently examined Datalog programs with *min* or *max* aggregate operations. Their work addresses problems that are similar to those that we consider, but the approaches are quite different and the techniques are complementary. We present a comparison of our techniques with those of Ganguly et al. in Section 6.1, and describe several advantages of our approach.

The rest of the paper is organized as follows. We present basic definitions in Section 2. Our notion of relevance is developed in Section 3, where we also introduce aggregate selections and constraints as a way of specifying relevance information. Techniques for propagation of aggregate selections and constraints through single rules are developed in Section 4.1. In Section 4.2 we present an algorithm to rewrite programs by propagating aggregate selections through the program, starting from the query. In Section 5 we show how to evaluate rewritten programs.

2 Definitions

We consider logic programs (an extension of Datalog programs that allows terms such as lists) extended with aggregation primitives. For simplicity, we only consider programs without negation although our results can be extended to deal with stratified negation in a straightforward manner. We also restrict the use of aggregation to be stratified. That is, if p is used to define q via a rule that uses aggregation, q cannot be used to define p. Further, we require that every variable in the head of a rule should appear in the body. This means that only ground terms can be generated, which is reasonable in a database context. Finally, we assume that program transformations such as Magic Sets have already been carried out; their use is largely orthogonal to our optimizations.

We assume standard definitions [Ull89]. We use overlines to denote tuples of terms, variables etc. We use $Vars(t)$ to denote the set of variables that occur in a term t. Similarly, $Vars(\overline{t})$ denotes the set of variables that occur in a tuple of terms $\overline{t}$.

The syntax and semantics that we use for aggregation is very similar to LDL [BNR+87]. Wlog we assume that there is at most one literal in the body of a rule that has an aggregate operation in the head. The semantics of a rule

$$p(\overline{t}, agg\_f < Y >) \leftarrow q(\ldots)$$

is as follows. We use the set of all facts that can be derived for q to instantiate $q(\ldots)$ and thus generate instantiations of variables in $Vars(\overline{t}) \cup \{Y\}$. For each value of $Vars(\overline{t})$ in this set, we first collect the set of corresponding instantiations of Y and apply aggregate operation agg_f to it to get a value $y_{\overline{t}}$, and then create a fact $p(\overline{t}, y_{\overline{t}})$.

3 Views of Relevance In Logic Programs

The idea of relevance of facts to a query is used by Prolog and other top-down evaluation techniques, as well as by program rewriting techniques such as Magic sets. Suppose we have a rule

$$R: \; p(\overline{t}) \leftarrow q_1(\overline{t_1}), q_2(\overline{t_2}), \ldots, q_n(\overline{t_n})$$

Assume for simplicity that we have a left-to-right rule evaluation (in the fashion of Prolog). Then a fact $q_i(a_i)$ is relevant if there is an instantiation

$$R': \; p(a) \leftarrow q_1(a_1), q_2(a_2), \ldots, q_i(a_i)$$

of (the head and first i body literals of) R such that the head fact $p(a)$ is relevant, and all instantiated facts $q_1(a_1), \ldots, q_{i-1}(a_{i-1})$ have been derived. Thus, the notion of relevance is local to a rule and to a set of facts that can instantiate it.

In contrast, in the shortest path problem we can decide that a particular fact $path(a,b,p1,c1)$ is irrelevant if a shorter path (fact) has been found. Such information is "global", in the sense that relevance depends on facts other than those used to instantiate a rule. We develop this notion of relevance for programs with aggregate operations in the rest of

this section, in three steps. (1) If agg_f is an aggregate function and S a set of values, we consider when some values in S can be ignored without affecting $agg\_f(S)$ (Section 3.1). (2) We use the ideas of step 1 to define when a fact is relevant (Section 3.2). (3) We introduce *aggregate selections* and *aggregate constraints* as a way of explicitly identifying irrelevant facts (Section 3.3).

3.1 Relevance and Aggregate Functions

Given a set of values and an aggregate function on the set, not all the values may be needed to compute the result of the aggregate function. For instance, if the aggregate function is *min*, no value except the minimum value is needed. We now formalize the notion of values being unnecessary for aggregate functions.

Definition 3.1 Incremental Aggregate Selector (IncSel) Functions : Let agg_f be an aggregate function $agg\_f : 2^D \rightarrow D$ on domain D. We say that agg_f is an *incremental aggregate selector* (*IncSel*) function if there exists a (nontrivial) function $unnecc : 2^D \rightarrow 2^D$ such that

1. $\forall S \subseteq D, \forall S1, (S - unnecc(S)) \subseteq S1 \subseteq S \Rightarrow$ $agg\_f(S1) = agg\_f(S)$
2. $unnecc_{agg\_f}$ is monotone. i.e., $\forall S1 \subseteq S2 \subseteq D$, $unnecc_{agg\_f}(S1) \subseteq unnecc_{agg\_f}(S2)$
3. $\forall S \subseteq D, unnecc_{agg\_f}(S) = unnecc_{agg\_f}($ $S - unnecc_{agg\_f}(S))$ □

Given a set S, Part 1 of the above condition lets us drop values in $unnecc_{agg\_f}(S)$ from S without affecting the result of $agg\_f(S)$. Part 2 of the above condition lets us detect unnecessary values before the entire set of values is computed—when we have computed some $S1 \subset S$, any value detected as unnecessary for agg_f on $S1$ is also guaranteed to be unnecessary for agg_f on S; a value that is necessary for $S1$ may however be unnecessary for S. Part 3 of this condition ensures that if a value is detected to be unnecessary for an aggregate operation on a set, it will continue to be detected as unnecessary if we discard unnecessary values from the set[2].

Consider an IncSel function agg_f on domain D. There may be more than one possible function *unnecc* as required by the definition of IncSel functions.

Definition 3.2 unnecessary$_{\text{agg_f}}$: For each incremental aggregate selector function agg_f that is allowed in our programs, a function *unnecc* (as above) is chosen, and is denoted by $unnecessary_{agg\_f}$.

The function $necessary_{agg\_f} : 2^D \rightarrow 2^D$ is defined as $necessary_{agg\_f}(S) = S - unnecessary_{agg\_f}(S)$. □

We do not consider how this choice is made, but assume it is made by the designer of the system based on the following criterion. Given two such functions f and g, we say $f \geq' g$ iff $\forall S \subseteq D, f(S) \supseteq g(S)$; clearly $>'$ (the strict version of $\geq'$) is an (irreflexive) partial order. Preferably, a function that is maximal under the (irreflexive) partial order $>'$ is chosen.

Note that $unnecessary_{agg\_f}(S)$ could be infinite. We do not construct an infinite set $unnecessary_{agg\_f}(S)$, but require that we can efficiently test for the presence of a value in $unnecessary_{agg\_f}(S)$, for finite S.

The function *min* on reals, with $unnecessary_{min}(S) = \{x \in D \mid x > min(S)\}$ is an IncSel function. The function *max* on reals with $unnecessary_{max}$ symmetrically defined is also an IncSel function. Other examples (with the functions $unnecessary_{agg\_f}$ appropriately defined), include the aggregate function that selects the kth largest element of a set for some constant k, and the aggregate function that sums up the k largest elements of a set. Although we only consider aggregate functions of the form $2^D - > D$, the ideas in this paper can be extended to aggregate functions of the form $2^D \times D \rightarrow T/F$. Examples of such functions include "select the best three results". We can also extend the ideas to aggregate functions on multisets.

In the rest of the paper, we assume that the optimization techniques are applied only on IncSel functions, and that a set of such aggregate functions and the corresponding functions $unnecessary_{agg\_f}$ are given to us.

3.2 Relevance of Facts

We now use the notion of necessity with respect to an aggregate function in defining our extended notion of relevance of facts.

Definition 3.3 Relevance of Facts : Consider a program P with a query on it. A fact $p(\overline{a})$ is *relevant to the query* iff one of the following is true:

1. $p(\overline{a})$ is an answer to the query, or
2. $p(\overline{a})$ occurs in the body of an instantiated rule without aggregation in the head such that every literal in the body is true in the least model[3], and the head fact of the rule is relevant to the query, or
3. There is a rule R in the program
   $R : q(\overline{t_1}, agg\_f(< Y >)) \leftarrow p(\overline{t_2})$
   and an instantiation R' of R,
   $R' : q(\overline{a_1}, agg\_f(< Y >)) \leftarrow p(\overline{a_2})$
   such that
   (a) Y is free in R' and all other variables are bound to ground terms, and
   (b) Let S_Y be the set of all possible instantiations b of Y such that $p(\overline{a_2})[Y/b]$ is true in the model.

[2] This is used in Theorem 5.1 to show that inferences are not repeated. None of the other results require aggregate functions to satisfy this condition.

[3] The program semantics is based upon a least model. For positive Horn logic programs, this is the least Herbrand model. In the presence of set terms, we must consider models over an extended Herbrand universe [BNR+87]. The definition can be extended to non-stratified programs.

Then $q(\overline{a_1}, agg\_f(S_Y))$ is present in the model and is relevant to the query, and

(c) $p(\overline{a}) = p(\overline{a_2})[Y/b1]$, where $b_1 \in necessary_{agg\_f}(S_Y)$. □

A fact is said to be *irrelevant to the query* if it is not relevant to the query. In future, we simply say relevant (resp. irrelevant) when we mean "relevant to the query" (resp. "irrelevant to the query").

Example 3.1 Consider a program with one rule

$R: p(X, min(< Y >)) \leftarrow q(X, Y)$

and facts $q(5,4), q(5,6)$ and $q(5,3)$. Let the query on the program be $?p(X,Y)$. Fact $p(5,3)$ is generated as an answer. With $X = 5$, the set of facts that match the body of the rule have Y values of $3, 4$ and 6, of which only 3 is necessary for *min*. Using the above definition of relevance, we find that the facts $q(5,4)$ and $q(5,6)$ are irrelevant to the query, while $q(5,3)$ is relevant. Also, by the above definition, for the shortest path program (Figure 1) all *path* facts, except those corresponding to shortest paths, are irrelevant. □

Our extended notion of relevance is very tight, and in general we may not be able to determine the relevance of a fact without actually computing the least model of the program. The techniques we present will use sufficient but not necessary conditions to test for irrelevance. During the evaluation of some programs we may generate a fact, and later discover that it is irrelevant, for instance when some other "better" fact is generated. Once a fact is found to be irrelevant, by "withdrawing" this fact, we may be able to determine that other facts generated using it can no longer be generated, and hence can also be "withdrawn". The cost of such cascading withdrawals could be very high, and so we confine ourselves to only discarding irrelevant facts. Although this could result in some additional irrelevant computation, the gains in efficiency from our optimization can still be significant.

3.3 Aggregate Constraints and Selections

We now introduce some concepts that allow us to specify relevance information. Informally, *sound aggregate selections* are used to specify tests for relevance of facts—if there is a sound aggregate selection on a predicate in our rewritten program, and a fact for the predicate does not satisfy the selection, the fact is irrelevant. Aggregate selections are introduced by our rewriting algorithm and the information is used by our evaluation algorithm. The syntax (using a variant of Starburst SQL *groupby*) and semantics of aggregate selections are described in the next few definitions.

Definition 3.4 Atomic Aggregate Selection: An *atomic aggregate selection* has the following syntax:

$c(\overline{u}) : groupby(p(\overline{t}), [\overline{X}], agg\_f(Y))$

Here $c(\overline{u})$ denotes a literal or a conjunction of literals, and $\overline{X}$ a set of variables such that $\overline{X} \subseteq Vars(\overline{t})$. We must have $Y \in Vars(\overline{t})$, and agg_f must be an IncSel function.

Consider a program P with an associated least model. Given the set of facts for predicate p in the least model of P, we have a set of instantiations of $\overline{t}$. Since $\overline{X} \subseteq Vars(\overline{t})$ and $Y \in Vars(\overline{t})$, for each value $\overline{d}$ of $\overline{X}$ in this set of instantiations, we have a corresponding set of values for Y; we denote this set by $S_{\overline{d}}$. We construct (conceptually) a relation $unnecc\_agg(\overline{X}, Y)$ with a tuple $(\overline{d}, e)$ for each $\overline{d}$, and each $e \in unnecessary_{agg\_f}(S_{\overline{d}})$.

Let $c(\overline{a})$ be a ground conjunction. We say that $c(\overline{a})$ *satisfies the atomic aggregate selection* s_i iff there exists a substitution σ such that (1) $c(\overline{a}) = c(\overline{u})[\sigma]$, (2) σ assigns ground terms to all variables in $Vars(\overline{u}) \cup \overline{X} \cup \{Y\}$, and (3) $(\overline{X}, Y)[\sigma]$ is not in $unnecc\_agg$ [4]. □

In the above definition, the variables in $[\overline{X}]$ are called *grouped variables* and the variable Y is called the *aggregated variable* in the atomic aggregate selection. The variables in the set $((Vars(\overline{t}) - \overline{X}) - \{Y\})$ are local to the groupby, and cannot be quantified or instantiated from outside the groupby.

Definition 3.5 Aggregate Selection: An *aggregate selection* s is a conjunction of atomic aggregate selections, $s = (s_1 \wedge s_2 \wedge \ldots \wedge s_n)$.

A ground conjunction $c(\overline{a})$ *satisfies an aggregate selection* $s = (s_1 \wedge s_2 \wedge \ldots \wedge s_n)$ iff it satisfies each of the atomic aggregate selections s_i individually. □

We use the short form $c(\overline{u}) : g1 \wedge g2$ to denote $(c(\overline{u}) : g1) \wedge (c(\overline{u}) : g2)$. We often say "the aggregate selection s on the body of R" to denote the aggregate selection $c(\overline{u}) : s$, where $c(\overline{u})$ is the body of rule R. Note that a conjunction of aggregate selections is also an aggregate selection.

Our approach to rewriting the program consists of placing aggregate selections on literals and rule bodies in the program in such a fashion that if a fact/rule instantiation does not satisfy the aggregate selection it is guaranteed to be irrelevant. Hence we define the concept of sound aggregate selections formally below.

Definition 3.6 Sound Aggregate Selection: An aggregate selection s is a *sound aggregate selection on the body of a rule* R iff only irrelevant facts are produced by instantiations of the body of R that do not satisfy s.

An aggregate selection s is a *sound aggregate selection for a literal* $p(\overline{t})$ in the body of a rule R iff only irrelevant facts are produced by instantiations of R that use for literal $p(\overline{t})$ any fact $p(\overline{a})$ that does not satisfy s.

An aggregate selection s is a *sound aggregate selection on a predicate* p iff any fact $p(\overline{a})$ is irrelevant if it does not satisfy s. □

[4] Note that the relation $unnecc\_agg$ could be infinite. To actually perform the test, we would take an instantiation of Y, and test if it is in $unnecessary_{agg\_f}(\overline{X})[\sigma]$ without actually constructing the whole (possibly infinite) set $unnecessary_{agg\_f}(\overline{X})[\sigma]$, or the (possibly infinite) relation $unnecc\_agg$.

Given a sound aggregate selection on a literal/rule, we can (partially) test during an evaluation whether a fact or an instantiated rule satisfies it. The extension of each predicate p at that point is a subset of the extension of p in the least model of the program. Since the aggregate functions are incremental aggregate selectors, an answer of "no" at that point means that the answer would be "no" in the least model of the program, and hence the fact/instantiation is irrelevant. However, an answer of "yes" is conservative, since the fact/instantiation may be detected to be irrelevant if all facts in the least model were available.

Example 3.2 Consider an aggregate selection

$path(X, Y, P, C) : groupby(path(X, Y, P, C), [X, Y], min(C))$

Suppose we have two facts $path(a, b, \_, 2)$ and $path(a, b, \_, 3)$ at a point in the computation. Then we know that $path(a, b, \_, 3)$ does not satisfy the selection. Later in the computation we may derive a fact $path(a, b, \_, 1)$. At this point we find that $path(a, b, \_, 2)$ also does not satisfy the selection. □

We define sound aggregate constraints next—they differ slightly from sound aggregate selections, and we use them in our rewriting algorithm to generate aggregate selections.

Definition 3.7 Sound Aggregate Constraint : An aggregate selection s is a *sound aggregate constraint* for predicate p iff every fact that can be derived for p satisfies the aggregate selection s. □

The following are technical definitions that we use primarily to ensure that the aggregate selections that we generate can be tested efficiently. The motivation is that the fact/rule instance on which we have an aggregate selection must bind all the variables in the aggregate selection.

Definition 3.8 Non-bound Variables : The *non-bound* variables of an atomic aggregate selection $c(\overline{u}) : groupby(p(\overline{t}), [\overline{X}], agg\_f(Y))$ are the variables in the set $(Vars(\overline{X}) \cup \{Y\})$. The non-bound variables of aggregate selection $s = s_1 \wedge \ldots \wedge s_n$ are those variables that are non-bound in at least one of the atomic selections aggregate s_i. □

Definition 3.9 Restrictions of Aggregate Selections : An atomic aggregate selection s_i is said to be *restricted* to a given set V of variables if every non-bound variable in s_i occurs in V. Let $s = (s_1 \wedge s_2 \wedge \ldots \wedge s_n)$. Then $restriction(s, V) = \wedge\{s_i \mid s_i \text{ is restricted to } V\}$ □

Example 3.3 Consider the following selection:

$s = c(\overline{u}) : groupby(path(X, Y, P, C), [X, P], min(C)) \wedge groupby(path(X, Y, P, C), [X, Y], min(C))$

The non-bound variables of s are X, Y, P and C, and

$restriction(s, \{X, Y, C\}) = c(\overline{u}) : groupby(path(X, Y, P, C), [X, Y], min(C))$ □

4 Aggregate Rewriting

We present a quick overview of the next few sections of the paper. We develop our algorithm for propagating relevance information in two steps. (1) In Section 4.1 we present a collection of techniques for generating sound aggregate selections. (2) In Section 4.2, we present our main rewriting algorithm, Algorithm Push_Selections, which uses these techniques as subroutines. In Section 5, we examine an evaluation mechanism that can take advantage of sound aggregate selections on predicates of the form p/s that are generated by the rewriting mechanism.

As a preview of what the techniques can achieve, consider Program Simple (Figure 1). The result of rewriting is Program Smart, shown in Figure 2. The notation $path/s1$ denotes a (new) predicate that is a version of $path$ with the sound aggregate selection $s1$ on it. The other predicates have no aggregate selections on them. This selection tells us that paths that are not of minimum length between their endpoints are irrelevant. Discarding such facts during the evaluation leads to considerable time benefits, and is discussed in Section 5.2.

4.1 Generation of Aggregate Constraints and Selections

In this section we present a collection of techniques for generating aggregate constraints and selections. The techniques are shown below. The reader may skip this section and proceed to Section 4.2 on a first reading. Technique C1 describes a way of deducing sound aggregate constraints on predicates. Techniques BS1, BS2 and BS3 describe three ways to generate sound aggregate selections on the bodies of rules. In Sections 4.1.1 and 4.1.2 we present a more sophisticated analysis that helps us to derive further sound aggregate selections on body literals. For lack of space we omit several other techniques for generating sound aggregate constraints and selections.

Technique C1: Suppose that there is only one rule defining p, and it is of the form:

$p(\overline{t}, agg\_f(< Y >)) \leftarrow q(\overline{t_b})$

Let $\overline{X} = Vars(t)$, and let agg_f be an IncSel function such that $\forall S \subseteq D, agg\_f(S) = relevant_{agg\_f}(S)$. Then $p(\overline{t}, Y) : groupby(q(\overline{t_b}), [\overline{X}], agg\_f(Y))$ is a sound aggregate constraint on p.

Technique BS1: Let R be of the form

$R : head(\overline{t_h}) \leftarrow c(\overline{t_b}), p(\overline{t})$

and suppose there is an aggregate constraint on p of the form: $p(\overline{t_1}) : s$ where all non-bound variables in s are included in $Vars(\overline{t_1})$. Suppose there exists a renaming[5] σ of variables in $\overline{t_1}$ such that $p(\overline{t}) = p(\overline{t_1})[\sigma]$. Then $s[\sigma]$ is a sound aggregate selection on the body of rule R.

Technique BS2: Suppose we have a rule of the form

$p(\overline{t}, agg\_f(< Y >)) \leftarrow q(\overline{t_b})$

with an aggregate operation in its head. Let $\overline{X} = Vars(\overline{t})$. Then $groupby(q(\overline{t_b}), [\overline{X}], agg\_f(Y))$ is a sound aggregate selection on the body of rule R.

Technique BS3: Consider a rule of the form

$R : p(\overline{t_h}) \leftarrow body(\overline{t_b})$.

$R1 : shortest\_path(X, Y, P, C) \leftarrow s\_p\_length(X, Y, C), path/s1(X, Y, P, C).$
$R2 : s\_p\_length(X, Y, min(< C >)) \leftarrow path/s1(X, Y, P, C).$
$R3 : path/s1(X, Y, P1, C1) \leftarrow path/s1(X, Z, P, C), edge(Z, Y, EC),$
$append([edge(Z, Y)|nil], P, P1), C1 = C + EC.$
$R4 : path/s1(X, Y, [edge(X, Y)|nil], C) \leftarrow edge(X, Y, C).$
Selections:: $s1 = path/s1(X, Y, P, C) : groupby(path/s1(X, Y, P, C), [X, Y], min(C)))$

Figure 2: Program Smart

Suppose the head predicate p has a sound aggregate selection $p(\overline{t}) : s$ on it, where all non-bound variables in s are included in $Vars(\overline{t})$. Suppose there exists a renaming[5] σ of variables in $\overline{t}$ such that $p(\overline{t_h}) = p(\overline{t})[\sigma]$. Then $s[\sigma]$ is a sound aggregate selection on the body of rule R.

Technique LS1: Let s be a sound aggregate selection on the body of a rule R, and let $p(\overline{t})$ be a literal in the body of R. Then $p(\overline{t}) : restriction(s, Vars(\overline{t}))$ is a sound aggregate selection on the literal $p(\overline{t})$ in the body of R.

Example 4.1 Consider Program Simple (Figure 1). Using Technique C1 and rule $R2$ we get the aggregate constraint

$s\_p\_length(X, Y, C) : groupby(path(X, Y, P, C), [X, Y], min(C))$

on the predicate s_p_length. Using this aggregate constraint with rule R1, Technique BS1 deduces the following sound aggregate selection on the body of rule $R1$:

$groupby(path(X, Y, P1, C), [X, Y], min(C)).$

Using Technique BS2 we get the following sound aggregate selection on the body of rule $R2$:

$groupby(path(X, Y, P, C), [X, Y], min(C))$

If we had a sound aggregate selection $path(X, Y, P, C) : groupby(path(X, Y, P, C), [X, Y], min(C))$ on the head predicate of rule $R3$, Technique BS3 would derive the following sound aggregate selection on the body of rule $R3$:

$groupby(path(X, Y, P1, C1), [X, Y], min(C1)).$

From these sound aggregate selections on the bodies of $R1$ and $R2$, using LS1, we deduce the sound aggregate selection

$path(X, Y, P, C) : groupby(path(X, Y, P1, C), [X, Y], min(C))$

on the literal $path(X, Y, P, C)$ in the body of the rule $R1$, and the sound aggregate selection

$path(X, Y, P, C) : groupby(path(X, Y, P, C), [X, Y], min(C))$

on the literal $path(X, Y, P, C)$ in the body of the rule $R2$. □

[5] We could allow σ to be a substitution on variables. However, to simplify the task of ensuring that our rewriting algorithm terminates, we restrict ourselves to renamings.

4.1.1 Pushing Aggregate Selections

We now look at another way of generating aggregate selections on rule body literals. But first we present some definitions. Aggregate functions such as min and ordinary functions as $+$ or $*$ interact in a particular fashion, and we use this interaction to generate sound aggregate selections on literals in the bodies of rules.

Definition 4.1 Distribution : Let fn be a total function $fn : D \times D \times \ldots \times D \rightarrow D$ that maps n-tuples of values from D to a value in D. Define $s\_fn(U) = \bigcup\{fn(\overline{t}) \mid \overline{t} \in U\}$. Let agg_f be an aggregate function $agg\_f : 2^D \rightarrow D$. Let $S_1, S_2, \ldots S_n$ be subsets of D, and let $S = S_1 \times S_2 \times \ldots \times S_n$. Let $R = necessary_{agg\_f}(S_1) \times necessary_{agg\_f}(S_2) \times \ldots \times necessary_{agg\_f}(S_n)$. Then $necessary_{agg\_f}$ is said to *distribute* over fn iff for every $S_1, \ldots, S_n$, $agg\_f(s\_fn(R)) = agg\_f(s\_fn(S))$. □

For example $necessary_{min}$ distributes over "+" for reals and integers, and over $*$ for positive reals and positive integers, but does not distribute over $*$ for arbitrary reals [6]. Technique PS1 shows a way of deriving aggregate selections on literals in rule bodies by making use of distribution of aggregate functions over ordinary functions.

Technique PS1: Let R be a rule of the form

$R : p_h(\overline{t_h}) \leftarrow \ldots, p(\overline{t}, Wi), \ldots, Y = fn(W1, \ldots, Wn)$

such that there is no aggregate operation in the head of R. Suppose

1. There is a sound atomic aggregate selection on the body of R, of the form $groupby(p_h(\overline{t_h}), [\overline{X}], agg\_f(Y))$
2. $necessary_{agg\_f}$ distributes over fn, and
3. Each of $W1, \ldots, Wn, Y$ are distinct variables, and they each occur in exactly one literal other than $Y = fn(W1, \ldots, Wn)$ in the body of R; no two W_i's appear in the same literal; further, Y does not appear in any other literal in the body of the rule.

Define the *non-repeated arguments* of $p(\overline{t}, Wi)$ as those of the form V, where V is a variable that does not appear anywhere else in the body of the rule, and

[6] We extend the notion of distribution considerably in the full version of the paper.

$V \notin Vars(\overline{X}) \cup \{Y\}$. Then the following is a sound atomic aggregate selection on the literal $p(\overline{t}, Wi)$ in the body of the rule:

$p(\overline{Z}, W_i) : groupby(p(\overline{Z}, Wi), [\overline{Z'}], agg\_f(Wi))$

where $\overline{Z}$ is a tuple of new variables, with arity the same as $\overline{t}$, and where $\overline{Z'}$ contains all variables in $\overline{Z}$ other than those that appear in non-repeated arguments of $p(\overline{t}, W_i)$.

The above technique works for a version of the shortest path program, that computes the path length but does not keep track of the path information. In the next section we see some shortcomings of this technique, and extend it.

4.1.2 Extended Techniques for Pushing Selections

Certain predicates, such as *append*, used in the bodies of rules are total functions on some types. Given any two values of type list as the first two arguments of *append*, there is guaranteed to be a third value such that the predicate is true. Such functions are said to be "*non-constraining*" on arguments of the appropriate type. Under certain conditions, if such a function appears as a literal in the body of a rule we can drop the literal before applying Technique PS1. The result of dropping such literals from a rule is the *reduction* of the rule; if we apply Technique PS1 and generate an aggregate selection s for a literal in the reduction of the rule, then s is a sound aggregate selection for the literal in the original rule. Due to lack of space, we do not give details of the technique here, but present a brief example of its use.

Example 4.2 We continue with Example 4.1. Suppose we have a sound atomic aggregate selection $groupby(path(X, Y, P1, C1), [X, Y], min(C1))$ on the body of rule $R3$. The reduction of $R3$ wrt to the atomic aggregate selection is

$$R3' : path(X, Y, P1, C1) \leftarrow path(X, Z, P, C), edge(Z, Y, EC), C1 = C + EC.$$

Using Technique PS1, on the reduction, we find that the third argument of $path(X, Z, P, C)$ is non-repeated. Hence we deduce the following sound aggregate selection on the literal *path*

$path(X, Y, P, C) : groupby(path(X, Z, P, C), [X, Z], min(C))$

and the sound aggregate selection

$edge(Z, Y, EC) : groupby(edge(Z, Y, EC), [Z, Y], min(EC))$

on the literal *edge*.

If we used Technique PS1 without the reduction step, we would get the aggregate selection

$path(X, Y, P, C) : groupby(path(X, Z, P, C), [X, Z, P], min(C))$

which is "weaker" than the selection described above. □

4.2 The Aggregate Rewriting Algorithm

In this section we present a rewriting of the program based on the propagation of sound aggregate selections. The rewriting algorithm is somewhat similar to the adornment algorithm used in Magic sets rewriting (see [Ull89]). When it detects that an occurrence of a predicate p in the body of a particular rule has a sound aggregate selection s on it, it creates a new labeled version p/s of p. That occurrence of predicate p is replaced by p/s, and by using aggregate selection s, (copies of) rules defining p are specialized to define p/s.

The rewriting algorithm is shown below. In Step 7 of the algorithm, s is a sound aggregate selection on the head of R', and this along with any aggregate constraints on body predicates may be used with techniques from Section 4.1 to generate new aggregate selections.

Algorithm Push_Selections(P, P^{as})
Input: Program P, and query predicate *query_pred*.
Output: Rewritten program P^{as}.
1) Derive sound aggregate constraints on the predicates of the program.
2) Push $query\_pred/nil$ onto *stack*.
3) While *stack* not empty do
4) Pop p/s from the stack and mark p/s as seen.
5) For each rule R defining p do
6) Set R' = a copy of R with head predicate replaced by p/s.
7) Derive sound aggregate selections for each body literal p_i of R'.
8) For each p_i in the body of R' do
9) Let si denote the conjunction of sound aggregate selections derived for p_i.
10) If a version p_i/t of p_i such that $t \leq si$ has been seen,
11) Then choose one such, and set $si = t$;
12) Else push p_i/si onto *stack*.
13) Output a copy of R', with each p_i replaced by p_i/si.
14) Output selection s on p/s.
End Algorithm.

Postprocessing 1: For each predicate p, for each version p/s of p, choose the weakest version p/t of p in the rewritten program such that $s \geq t$. Replace all occurrences of p/s in bodies of rules in the rewritten program by p/t. Finally, remove all rules that are not reachable from the query.

Postprocessing 2: Suppose we have an atomic aggregate selection $s = groupby(p(\overline{t}), [\overline{X}], agg\_f(Y))$ in the rewritten program. If p is absent from the rewritten program select version p/s of p if it exists. If not, select a version[7] $p/s1$ of p if any such version exists. If no $p/s1$ was found, p is not connected to the query predicate—drop the selection s from all predicates that use it. Otherwise rename p in the *groupby* in s to p/s or $p/s1$ as the case may be.

An aggregate selection s is *stronger than* an aggregate selection t (denoted as $s \geq t$), if whenever t classifies an instantiation as irrelevant, then so does s. We can obtain simple sufficient conditions for this, which we omit for lack of space. If in the rewritten program there are two versions of p, p/s and p/t such that $s > t$, there is no point using the stronger version p/s—all the facts computed for p/s will be computed anyway for p/t. Preprocessing to remove p/s is described in Postprocessing 1.

As a result of the renaming of predicates, predicates in aggregate selections may not be present in the rewritten program. Postprocessing 2 describes how to fix this. [8]

Algorithm Push_Selections terminates on all finite input programs, producing a finite rewritten program. The rewritten program could potentially be large, but, as is the case with the adornment algorithm for Magic sets rewriting, this is very unlikely to happen in practice—the rewritten program is likely to be not much larger than the original program. To ensure that the rewritten program is small we could adopt heuristics such as bounding the number of atomic aggregate selections in an aggregate selection to some fixed small value, or bounding the number of different aggregate selections on each predicate. We omit details here; these restrictions may increase the number of facts computed, but will not affect correctness.

Proposition 4.1 (Stratification) : If the initial program is stratified wrt aggregation, then the aggregate rewritten program is also stratified wrt aggregation. □

Lemma 4.1 (Correctness of Rewriting) : Semi-Naive Evaluation of P^{as} gives the same set of answers for $query\_pred$ as Semi-Naive evaluation of P. Further, the aggregate selections on each predicate in P^{as} are sound aggregate selections.□

Example 4.3 Applying this algorithm to Program Simple, we get the optimized program, Program Smart shown in Figure 2). The algorithm starts with the query predicate $shortest\_path$. Creation of aggregate constraints, and pushing them into rules is done as discussed in earlier examples, and the operation of Algorithm Push_Selections is fairly straightforward. As a result of the rewriting we get the rules of Program Smart, but with $path/s1$ having the following sound aggregate selection on it:

$$path/s1(X, Y, P, C) : groupby(path(X, Y, P, C), [X, Y], min(C))$$

On postprocessing, we rename predicate $path$ in the above selection to $path/s1$, to get Program Smart. To get the benefits of the rewriting, the evaluation must make use of the aggregate selections present in Program Smart. We describe how to do this in the next section. □

5 Aggregate Retaining Evaluation

In this section we see how to evaluate a rewritten program making use of aggregate selections on predicates. Essentially, once we know that a fact does not satisfy a sound aggregate selection on it we know that it is irrelevant to the computation, and can discard it.

We define *Aggregate Retaining Evaluation* as a modification to Semi-Naive evaluation (see e.g. [Ull89]): At the end of each iteration of Semi-Naive evaluation, we discard facts that have been computed for each predicate if they do not satisfy a sound aggregate selection on the predicate.

Theorem 5.1 (Correctness, Completeness, Non-Redundancy) : Evaluation of P^{as} using Aggregate Retaining evaluation gives the same set of answers for $query\_pred$ as evaluation of P using Semi-Naive evaluation, and does not repeat any inferences. Further, the Aggregate Retaining evaluation of P^{as} terminates whenever the Semi-Naive evaluation of P terminates. □

Example 5.1 Predicate $path/s1$ in Program Smart has a sound aggregate selection $s1 = path/s1(X, Y, P, C) : groupby(path/s1(X, Y, P, C), [X, Y], min(C))$. In the evaluation of Program Smart, we maintain at most one $path/s1$ fact at a time with a given value for X, Y. If a fact is generated with any value for X and Y and another fact with the same value for X and Y already exists we know that the one with the greater C value does not satisfy the aggregate selection. Hence it can be discarded. □

5.1 Pragmatic Issues Of Testing Aggregate Selections

Our selection propagating techniques ensure that all non-bound variables in a groupby of an atomic aggregate selection also appear in the corresponding literal on which the selection is applied. When testing an atomic aggregate selection on a fact f, we have a unique instantiation of the grouped variables of the selection, and the test can be performed efficiently. If the test determines that fact f is irrelevant, f is discarded, else it is retained. As the computation proceeds, the set of unnecessary values for the "group" to which f belongs (i.e., the set of facts with the same values in the grouped arguments) could change, and this might enable us to determine that f is irrelevant after all. By sorting the set of facts on the grouped arguments, this "re-testing" can be done efficiently. The cost of sorting is small for the aggregate operations we consider in this paper; in the case of max or min aggregate operations there is at most one fact stored in each set.

[7] We omit details on how to make this choice from this version of the paper.

[8] A renaming of p is a version of p with an aggregate selection on it, and is thus a subset of p. Due to monotonicity of the functions $unnecessary_{agg\_f}$, any value that is found unnecessary wrt the subset would also be unnecessary wrt the full set. Hence while the new selection may not be as strong as the original one, the renaming is guaranteed to be sound.

Proposition 5.1 (Bounds on Performance) : Given a program that uses only aggregate operations defined in this paper, and a data set, let the time for Aggregate Retaining Evaluation of the program on the dataset be t_R, and let t_O be the time taken to evaluate the original program on the dataset. There is a constant k (independent of the data set) such that $t_R \leq k * t_O$. □

This means that Aggregate Retaining evaluation of the rewritten program can do at most a constant factor worse than Semi-Naive evaluation of the original program— the converse is not true.

Using Aggregate Retaining Evaluation, Program Smart runs in time $O(EV^2)$, and the single source version of the program [9] runs in time $O(EV)$. These bounds hold even if there are negative length edges, so long as there are no negative cycles in the *edge* graph.

5.2 Ordered Search

Consider the shortest path problem with a given starting point. Dijkstra's algorithm takes $O(E * log(V))$ time if we use a heap data structure to find the minimum cost path at each stage. However, Aggregate Retaining Evaluation on the single source shortest path program takes $O(E * V)$ time. We can get the effect of Dijkstra's algorithm by extending at each stage only the shortest path that hasn't been extended yet. In other words, we use only the *path* facts that are of minimal cost among those that haven't yet been used. This important observation is made in [GGZ91] and is used in their evaluation algorithm (see Section 6.1 for a brief description) for monotonic min programs (in their notation a *min program* is one that uses only the aggregate operation *min*, and it is said to be monotonic if it is monotonically non-decreasing on a particular argument of each predicate).

We make use of this idea to derive an improved evaluation technique for stratified min programs. The basic idea is to modify Aggregate Retaining Evaluation by hiding all facts whose cost arguments are not of minimum value until no more derivations can be made. At this stage the hidden fact whose cost argument is minimum (over all hidden facts) is made visible. The whole process is repeated until there are no more hidden facts. As before, facts that do not satisfy sound aggregate selections on predicates are discarded. We omit details here due to lack of space. We call this evaluation technique as *Ordered Aggregate Retaining Evaluation*.

Theorem 5.2 *Ordered Aggregate Retaining Evaluation is sound, and is complete for and terminates on those programs on which Aggregate Retaining evaluation terminates.*

The effect of the above evaluation is exactly the same as if Ganguly et al.'s evaluation technique were used, for the case of stratified monotonic min programs. For instance, Ordered Aggregate Retaining Evaluation of the single source shortest path program would explore paths in order of increasing cost, and would have time complexity $O(Elog(V))$ which is the same as that of the technique of Ganguly et al. and Dijkstra's algorithm. Program Smart would have time complexity $O(EVlog(V))$ using any of these techniques.

Ordered Aggregate Retaining Evaluation also works on (and Theorem 5.2 holds for) min programs that are not monotonic. For instance, the shortest path program is non-monotonic if there are negative cost edges. But even in this case, Ordered Aggregate Retaining Evaluation of Program Smart functions correctly, and terminates if there are no negative cost cycles.

6 Discussion

We now see some more examples of programs to which our techniques are applicable.

Example 6.1 The following program defines the earliest finish time of a task, given the finish times of preceding tasks.

$$
\begin{array}{ll}
R1 : e\_fin(X, max(< T >)) & \leftarrow fin(X, T). \\
R2 : fin(X, T) & \leftarrow precedes(X, Y), fin(Y, T1), \\
 & \quad delay(X, D), T = T1 + D. \\
R3 : fin(X, T) & \leftarrow first(X), delay(X, T).
\end{array}
$$

This program can be optimized using our techniques, and in the resultant program fin is replaced by fin/s, where s is the aggregate selection $fin/s(X, T) : groupby(fin/s(X, T), [X], max(T))$. The rules and other predicates are the same, but $finish$ facts that don't have maximal times are deduced to be irrelevant. We can extend this program to compute the critical path, and still apply our optimizations. □

Example 6.2 With a minor modification to Technique BS1, to allow pushing aggregate selections through rules with aggregate operations in the head, we can optimize the following program. Predicate $path2(X, Y, H, C)$ denotes a path where X and Y are source and destination, H denotes hops, and C denotes cost.

$$
\begin{array}{ll}
\text{Query: } ?\text{-}p\_best(X, Y, H, C). & \\
R1 : p\_best(X, Y, H, C) & \leftarrow p\_few(X, Y, H), \\
 & \quad p\_short(X, Y, H, C) \\
R2 : p\_few(X, Y, min(< H >)) & \leftarrow p\_short(X, Y, H, C). \\
R3 : p\_short(X, Y, H, min(< C >)) & \leftarrow path2(X, Y, H, C). \\
/* \ \ldots \text{ Rules for } path2 \ \ldots \ */ &
\end{array}
$$

The program finds flights with the minimum number of hops, and within such flights, finds those with minimum cost. Our technique generates the aggregate selection $path2(X, Y, H, C) : s$ where:

$$s = groupby(path2(X, Y, H, C), [X, Y, H], min(C)) \wedge groupby(path2(X, Y, H, C), [X, Y], min(H)).$$

[9] This version is obtained automatically by using the Factoring transformation [NRSU89] on Program Dumb, before using Aggregate Rewriting. We do not show details here, but the net effect is as if the first argument of *path* becomes a fixed constant. Aggregate Rewriting optimizes the resultant program successfully. We also assume that sharing of ground lists between the body and head facts of a rule can be done, so that the *append* calls in the program can be executed in constant time.

The selection propagates unchanged through the rules defining $path2$, so that the rewritten program is the same except for having the sound aggregate selection $s1$ on $path2$ as well as aggregate selections on p_few and p_best. □

Example 6.3 The following program can be used to find the cost of the cheapest three paths, and illustrates the ability of our techniques to handle aggregate operations other than min and max. We use the aggregate operation $least3$ that selects the three least values[10].

Query: ?-$shortest3(X, Y, C)$.
$R1: shortest3(X, Y, P, least3(< C >)) \leftarrow path(X, Y, P, C)$.
/* ... Rules for $path$ as in Figure 1 ... */

Aggregate operation $least3$ is an IncSel function (under an extended definition of IncSel functions that we do not present in this paper), with $unnecessary_{least3}(S)$ defined as all values greater than the third lowest value in S. Also, $necessary_{least3}$ distributes over "+". Hence our rewriting technique proceeds on the rules for $path$ in this program exactly as it does for the earlier shortest path problem (Example 4.3) and the $path$ rules in the rewritten program are the same as in Program Smart (Example 4.3) except that min is replaced by $least3$. Evaluation of the rewritten program is very similar too, except that instead of retaining only minimum paths between pairs of points, the cheapest three paths between pairs of points are retained. □

Our optimization techniques are orthogonal to the Magic Sets transformation, and are applicable to programs that cannot be expressed using transitive closure, as the next example shows.

Example 6.4 Consider Program Nearest_Same_Generation (adapted from [GGZ91]) in Figure 3, that computes the "nearest" among all nodes in the "same generation" as a node s. Our techniques can be applied to optimize this program. This program has been rewritten using the Magic Sets transformation.

The rewriting produces essentially the same program except that there is an aggregate selection $s = sg^{bff}(X, Y, D) : groupby(sg^{bff}(X, Y, D), [X, Y], min(D))$ on predicate sg^{bff}. In the evaluation of the rewritten program, for each X, Y pair only the fact $sg^{bff}(X, Y, D)$ such that D is minimum is retained. □

6.1 Related Work

Several papers in the past [RHDM86, ADJ88] addressed optimizations of generalized forms of transitive closure that allowed aggregate operations. Cruz and Norvell [CN89] examine the same problem in a generalized algebraic framework. On the other hand, we deal with a language that can express more general recursive queries with aggregation, and do not make use of any special syntax.

Recently Ganguly et al. [GGZ91] presented optimization techniques for monotonic increasing (resp. decreasing) logic programs with min (resp. max) aggregate operations. Informally, there must be a single cost argument for each predicate in the program and the program must be monotonic on this argument. They transform such a program into a (possibly unstratified) program with negation whose stable model yields the answers to the original program, but does not contain any irrelevant facts. They also present an efficient evaluation mechanism for computing the stable model for the transformed program.

Our results were obtained independently of Ganguly et al. [GGZ91]. The results of Ganguly et al. complement this work in two important ways. Their idea of ordering of facts in the computation (which we have adapted and extended in Section 5.2) offers significant improvements in time complexity, and unlike our technique, theirs can handle monotonic min programs even if the use of min is unstratified.

Our techniques are more general than those of Ganguly et al. in several ways. (1) Our techniques are applicable to stratified programs that are not monotonic, and that can contain multiple aggregate operations including min and max. (2) For the class of stratified monotonic min programs, our rewriting techniques generate selections that are at least as strong as those generated by Ganguly et al. (3) Given a stratified monotonic min program, its evaluation using rewriting and Ordered Aggregate Retaining Evaluation computes no more facts (in an order of magnitude sense) than its evaluation using their techniques.

There are many common examples of programs that can benefit from our optimizations, although they cannot be handled by [GGZ91] since they are not appropriately monotonic. These include the shortest path problem with edges of negative weight, and the the earliest finish time problem shown in Example 6.1. [11] Further, the Magic Sets rewritten versions of many monotonic non-linear programs are non-monotonic, and our optimizations would be useful in this context.

Unlike [GGZ91] we allow aggregate operations other than max and min, for instance "least k values". We also allow predicates with multiple cost arguments and allow multiple atomic aggregate selections on the same predicate. The use of these generalizations is illustrated in Examples 6.2 and 6.3, which cannot be handled by Ganguly et al.

7 Extensions and Conclusions

We believe that evaluation with Aggregate Optimization will offer significant time benefits for a significant class of stratified programs that use aggregate operations similar to min and max. We believe that given a technique such as that of Ganguly et al., or of Beeri et al. [BRSS89] for evaluating special classes of unstratified programs, our results can

[10] This aggregate operation returns a value that is in the extended Herbrand universe[BNR+87]. Although we do not consider these in this paper due to space limitations, this causes no problems for our optimization techniques.

[11] This program uses max and is monotonically increasing, whereas Ganguly et al. require it to be monotonically decreasing.

$R1: nearest\_sg^{bff}(X, Y, min(< D >)) \leftarrow m\_nearest\_sg^{bff}(X), sg^{bff}(X, Y, D).$
$R2: sg^{bff}(X, Y, D) \leftarrow m\_sg^{bff}(X), up(X, Z1), sg^{bff}(Z1, Z2, D1), down(Z2, Y), D = D1 + 1.$
$R3: sg^{bff}(X, Y, 1) \leftarrow m\_sg^{bff}(X), flat(X, Y).$
$R4: m\_sg^{bff}(X) \leftarrow m\_nearest\_sg^{bff}(X).$
$R5: m\_sg^{bff}(Z1) \leftarrow m\_sg^{bff}(X), up(X, Z1).$
$R6: m\_nearest\_sg^{bff}(s).$

Figure 3: Program Nearest_Same_Generation

be adapted to detect irrelevant facts using aggregate selections. Our optimization techniques may be useful for optimizing (non-recursive) SQL-like queries that use aggregate operations. We believe our techniques will find use in the bottom-up evaluation of quantitative logic programs (see e.g., [SSGK89]). Our techniques can be adapted to "push" a more general class of aggregate operations through rules, so that aggregate operations can be performed on smaller intermediate relations rather than on larger final relations. This in turn could enable us to discard facts that have been used in the aggregation. Operations such as *sum* or *count*, to which the optimization techniques we described do not apply, can benefit from such adaptations.

Acknowledgements: The authors would like to thank Divesh Srivastava for his comments and suggestions.

References

[ADJ88] R. Agrawal, S. Dar, and H. V. Jagadish. On transitive closure problems involving path computations. Technical Memorandum, 1988.

[AHU74] Alfred V. Aho, John E. Hopcroft, and Jeffrey D. Ullman. *The Design and Analysis of Computer Algorithms*. Addison-Wesley, 1974.

[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 ACM Symposium on Principles of Database Systems*, pages 1–15, Cambridge, Massachusetts, March 1986.

[BNR+87] Catriel Beeri, Shamim Naqvi, Raghu Ramakrishnan, Oded Shmueli, and Shalom Tsur. Sets and negation in a logic database language. In *Proceedings of the ACM Symposium on Principles of Database Systems*, pages 21–37, San Diego, California, March 1987.

[BR87] Catriel Beeri and Raghu Ramakrishnan. On the power of magic. In *Proceedings of the ACM Symposium on Principles of Database Systems*, pages 269–283, San Diego, California, March 1987.

[BRSS89] C. Beeri, R. Ramakrishnan, D. Srivastava, and S. Sudarshan. Magic implementation of stratified programs. Manuscript, September 89.

[CN89] I. F. Cruz and T. S. Norvell. Aggregative closure: An extension of transitive closure. In *Proc. IEEE 5th Int'l Conf. Data Engineering*, pages 384–389, 1989.

[GGZ91] Sumit Ganguly, Sergio Greco, and Carlo Zaniolo. Minimum and maximum predicates in logic programming. In *Proceedings of the ACM Symposium on Principles of Database Systems*, 1991.

[MPR90] Inderpal S. Mumick, Hamid Pirahesh, and Raghu Ramakrishnan. Duplicates and aggregates in deductive databases. In *Proceedings of the Sixteenth International Conference on Very Large Databases*, August 1990.

[NRSU89] Jeffrey F. Naughton, Raghu Ramakrishnan, Yehoshua Sagiv, and Jeffrey D. Ullman. Argument reduction through factoring. In *Proceedings of the Fifteenth International Conference on Very Large Databases*, pages 173–182, Amsterdam, The Netherlands, August 1989.

[Ram88] Raghu Ramakrishnan. Magic Templates: A spellbinding approach to logic programs. In *Proceedings of the International Conference on Logic Programming*, pages 140–159, Seattle, Washington, August 1988.

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

[SSGK89] Nikolaus Steger, Helmut Schmidt, Ulrich Güntzer, and Werner Kiessling. Semantics and efficient compilation for quantitative deductive databases. In *IEEE International Symposium on Logic Programming*, pages 660–669, 1989.

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