Efficient Incremental Evaluation of Queries with Aggregation

Raghu Ramakrishnan
University of Wisconsin
Madison, WI 53706
raghu@cs.wisc.edu

Kenneth A. Ross
Columbia University
New York, NY 10027
kar@cs.columbia.edu

Divesh Srivastava
AT&T Bell Laboratories
Murray Hill, NJ 07974
divesh@research.att.com

S. Sudarshan
AT&T Bell Laboratories
Murray Hill, NJ 07974
sudarsha@research.att.com

Abstract

We present a technique for efficiently evaluating queries on programs with monotonic aggregation, a class of programs defined by Ross and Sagiv. Our technique consists of the following components: incremental computation of aggregate functions, incremental fixpoint evaluation of monotonic programs and Magic Sets transformation of monotonic programs. We also present a formalization of the notion of incremental computation of aggregate functions on a multiset, and upper and lower bounds for incremental computation of a variety of aggregate functions. We describe a proof-theoretic reformulation of the monotonic semantics in terms of computations, following the approach of Beeri et al.; this reformulation greatly simplifies the task of proving the correctness of our optimizations.

1 Introduction

There has been a lot of recent work in the literature on defining the semantics of complex database queries involving aggregate functions. Early work assumed some form of stratification of predicates to ensure that there was no recursion through aggregation. Subsequent proposals allowed recursion through predicates defined using aggregation, but required other forms of stratification to ensure that no fact depended on itself through aggregation (e.g., [13]). However, there are many useful queries that cannot be easily (if at all) expressed using these semantics, and more recent semantics such as [7, 9, 14, 15, 16, 17] have relaxed or removed stratification requirements.

In particular, the monotonic semantics of Ross and Sagiv [14] provides an intuitive meaning for a large class of programs that are not handled by the various stratified semantics. The semantics is a natural extension of the traditional least fixpoint semantics and is intuitive and easy to understand. The company controls program from [9], with cyclic stock ownership between companies, and the cheapest path program on a labeled, directed graph with cycles are two such programs. (We discuss the examples in more detail later.) In this paper, we present the first results (to our knowledge) on the problem of efficient evaluation of queries on programs under the monotonic semantics.

Our first contribution is on the incremental computation of aggregate functions (Section 2). We formalize what it means to incrementally compute aggregate functions on a multiset across a sequence of updates to the multiset. We present a framework for incremental computation of a large class of aggregate functions which are of a simple form; this framework also provides extensibility by allowing user-defined aggregate functions to be incrementally computed. We also provide upper and lower bounds for incremental computation of a variety of common aggregate functions.

Our second contribution is a novel reformulation of the monotonic semantics in terms of computations, following the approach of Beeri et al. [3] (Section 3). The least fixpoint characterization in [14] is very sensitive to the order in which facts are derived. Consequently, using this formulation, it is very difficult to show the correctness of program optimizations, such as the Magic Sets transformation, that change the order in which facts are derived. Our formulation allows for a better understanding of program optimizations, and helps us prove the correctness of the optimizations.

Our third contribution is to show how existing incremental evaluation techniques, e.g., Semi-Naive evaluation [1], can be combined with our techniques for incremental computation of aggregate functions, for the evaluation of monotonic programs (Section 4). This enables the efficient integration of the incremental evaluation procedure for programs with monotonic aggregation into existing deductive database systems. Many of the techniques described for monotonic programs have been implemented in the Coral deductive database system [11].

Our final contribution is to show that for left-to-right monotonic programs, which is a large sub-class of monotonic programs, the Magic Sets transformation (under simple restrictions such as using left-to-right "sips") can be applied to restrict computation to facts "relevant" to a given query. The correctness proof of the Magic Sets transformation depends crucially upon our semantic reformulation of the monotonic semantics.

1.1 Motivating Example

We assume familiarity with basic logic programming notation. Aggregate functions, such as *min*, *max*, *sum* and *count* are typically used in combination with a grouping facility, which is used to partition values into groups and aggregate on the values within each group. A groupby literal has the following syntax: $groupby(p(\overline{X}, \overline{Z}, Y), [\overline{X}], S = \mathbf{G}\langle Y\rangle)$, where $\overline{X}$ and $\overline{Z}$ denote tuples of variables, and $\mathbf{G}$ is an aggregate function on multisets. Intuitively, this literal is equivalent to the literal $p'(\overline{X}, S)$ where the relation p' is defined as follows. A tuple $p'(\overline{x}, s)$ is present in p' iff $s = \mathbf{G}(m)$, where $m = \pi_Y(\sigma_{\overline{X}=\overline{x}}(p(\overline{X}, \overline{Z}, Y)))$ is non-empty; here, π is the multiset projection operator, and σ is the selection operator of relational algebra.

Example 1.1 (Company Controls Program)
Consider the company controls example (modified from Mumick et al. [9]). A database fact of the form $owns\_stock(c1, c2, n)$ represents the information that company $c1$ owns fraction n of the stock of company $c2$. Consider the following program:

$$\begin{array}{ll} cv(X, X, Y, N) & :-\ owns\_stock(X, Y, N). \\ cv(X, Z, Y, N) & :-\ controls(X, Z), owns\_stock(Z, Y, N). \\ cv_1(X, Y, S) & :-\ groupby(cv(X, Z, Y, N), [X, Y], S = sum\langle N\rangle). \\ controls(X, Y) & :-\ cv_1(X, Y, S), S > 0.5. \end{array}$$

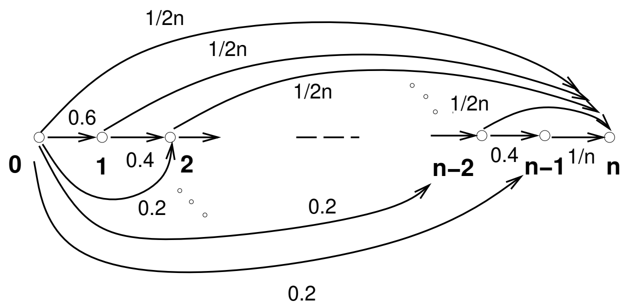

Figure 1: The *owns_stock* Relation

The above program can be understood under the monotonic semantics as follows. A fact of the form $controls(X, Y)$ indicates that company X has a controlling interest in company Y, i.e., X owns (directly or indirectly through an intermediate company that X controls) more than 50% of the stock of Y. The relation cv maintains information about (direct and indirect) stock ownership. A fact of the form $cv(X, X, Y, N)$ means that company X directly owns fraction N of the stock of company Y. A fact of the form $cv(X, Z, Y, N)$ means that company X indirectly owns fraction N of the stock of company Y via company Z, since X has a controlling interest in company Z, which directly owns fraction N of the stock of company Y. The relation $cv_1(X, Y, S)$ maintains information about the total fraction S of the stock of company Y owned by company X, by adding up the fractions of the stock of company Y owned (directly and indirectly) by company X.

Consider a dataset with the *owns_stock* relation depicted in Figure 1. The dataset is defined as follows: $owns\_stock(0, 1, 0.6)$, $owns\_stock(i, i+1, 0.4), 1 \leq i \leq n-2$, $owns\_stock(0, i, 0.2), 2 \leq i \leq n-1$, $owns\_stock(i, n, 1/2n), 0 \leq i \leq n-2$ and $owns\_stock(n-1, n, 1/n)$. The fixpoint computation of [14] recomputes the relation cv_1 each time new facts are added to the relation cv, which would result in a $\Theta(n^2)$ total cost for computing cv_1. Using our techniques of incremental computation of aggregate functions and incremental evaluation of monotonic programs, the total cost for computing cv_1 is only $O(n)$. $\Box$

2 Incremental Computation of Aggregates

2.1 Model of Computation

Definition 2.1 (Aggregate Function) Let $\mathcal{M}(D)$ denote the set of all multisets on domain D. An *aggregate function* $\mathbf{G}$ is any function whose domain is $\mathcal{M}(D)$. $\Box$

Informally, incremental computation refers to recomputing some value when the input changes "by a small amount". In our context, the input is a multiset, and we assume that the change in the input is caused by one of the following update operations on multisets: *insertion*, *deletion*, *replacement*, and *monotonic-replacement*. In all cases, only *one* element is inserted or deleted or replaced by an update operation.

The first three update types are self-explanatory. The last update type, monotonic-replacement, requires a partial ordering $\preceq$ on the domain D to be defined by the aggregate function $\mathbf{G}$, such that $\langle D, \preceq \rangle$ is a complete lattice. We say that c_2 is "better than" c_1 if $c1 \preceq c2$. Monotonic-replacement is the replacement of a value $c_1 \in D$ in a multiset by another value $c_2 \in D$, such that $c_1 \preceq c_2$.[1]

For example, in computing cheapest paths in a graph with costs on the edges, the *min* aggregate function is used; a "better" path here between two vertices is a cheaper path. Hence, the partial ordering for the *min* aggregate function over multisets of reals is given by $\geq$, i.e., $c_1 \preceq c_2$ iff $c_1 \geq c_2$.

Our model of computation is the Random Access Model (RAM), with the additional assumption that each of the basic arithmetic and comparison operations can be performed in constant time.

Our algorithms as well as the complexity analysis always assume that updates are correct, i.e., deletion, replacement and monotonic-replacement occur only on an existing value in the multiset. There are usually semantic reasons ensuring that this is always the case, e.g., if we have a relation with an aggregation function specified on an attribute, the deleted value is from that attribute of a deleted tuple. An independent check for existence of the value can be performed if updates may be incorrect; however, it would require maintaining the multiset and would take logarithmic time, which may change the incremental cost of the aggregate computation.

2.2 Incremental Aggregate Functions

The following definitions are based on [4, 12].

Definition 2.2 (Incremental Aggregate Algorithm) Let $\mathbf{G}$: $\mathcal{M}(D) \rightarrow D'$ be an aggregate function, with domain $\mathcal{M}(D)$ being the set of problem inputs, and range D' being the set of problem outputs. The size of a problem instance, i.e., the size of a multiset $p \in \mathcal{M}(D)$, is denoted by n.

Let $U \subseteq \{$insertion, deletion, replacement, monotonic-replacement$\}$ be a set of permitted update types. Let p be an input multiset, Δp (of a type in U) be an update on p, and p' be the result of update Δp on p. If, given as input p, $\mathbf{G}(p)$, Δp and possibly auxiliary information corresponding to p, algorithm A returns $q' = \mathbf{G}(p')$, and updates the auxiliary information to incorporate Δp, then A is called an *incremental aggregate algorithm* for $\mathbf{G}$. $\Box$

Obviously, any algorithm for computing $\mathbf{G}$ can be used in this situation since the entire input, p and Δp, is available to the algorithm. But in many applications, small changes in the input cause correspondingly small changes in the output, and it would be more efficient to compute the new output from the old output rather than to recompute the entire output from scratch, and we are interested in incremental algorithms that do exactly this.

We let $t_{\mathbf{G}}(n)$ denote the optimal worst-case asymptotic time complexity of computing an aggregate function $\mathbf{G}$ (when an optimal algorithm exists) on an input of size n, and let $t_A(n)$ denote the worst-case asymptotic time complexity of executing algorithm A on an input of size n.

Definition 2.3 (Incremental Complexity) Let $\mathbf{G}$ be an aggregate function. Let A be an incremental aggregate algorithm for computing $\mathbf{G}$, given a set U of update types. Let h and r be functions of n, the size of the input.

[1] Although replacement and monotonic-replacement can be modeled by the deletion of the old value followed by an insertion of the new value, there are cases where the "incremental" computation of an aggregate function is faster if we realize that the update is in fact a replacement or a monotonic-replacement, so we treat these cases separately.

Aggregate	Updates	Incremental Complexity		Space Requirement
		Absolute	Relative	
min	{ins., mon-repl.}	O(1)	O(1)	O(1)
	{ins., del., repl.}	O(log(n))	O(log(n))	O(n)
max	{ins., mon-repl.}	O(1)	O(1)	O(1)
	{ins., del., repl.}	O(log(n))	O(log(n))	O(n)
sum	{ins., del., repl.}	O(1)	O(1)	O(1)
product	{ins., del., repl.}	O(1)	O(1)	O(1)
count	{ins., del., repl.}	O(1)	O(1)	O(1)
average	{ins., del., repl.}	O(1)	O(1)	O(1)
variance	{ins., del., repl.}	O(1)	O(1)	O(1)
median	{ins., del., repl.}	O(log(n))	O(log(n))	O(n)
mode	{ins., del., repl.}	O(log(n))	O(1)	O(n)

Table 1: Incremental Cost of Evaluating Aggregates

If it can be shown that $t_A(n) = O(h(n))$, then we say that A has $(U, O(h))$ *absolute incremental complexity*. If $t_{\mathbf{G}}$ exists, and $(\Sigma_{i=1}^{n} t_A(i))/t_{\mathbf{G}}(n)$ is $O(r(n))$ for some function r, then we say that A has $(U, O(r))$ *relative incremental complexity*. □

For all aggregate functions $\mathbf{G}$ considered in this paper, $t_{\mathbf{G}}$ exists and is known. However, we should repeat the following remark from [4]: "More typically, we have a 'best known' algorithm, and our [relative incremental complexity] is relative to the complexity of that algorithm."

The *space requirement* of an incremental aggregate algorithm is the size of the part of the input (p, $\mathbf{G}(p)$, Δp and auxiliary information) that is actually required to compute q' in Definition 2.2 above.

Table 1 provides a summary of lower and upper bounds results for updates on a variety of aggregate functions. Some of these results are straightforward; details of the others, such as *median* and *mode*, are presented in the full version of the paper.

2.3 Extensible Incremental Aggregation

Several aggregate functions are definable by structural recursion on an associative, commutative binary operator [17]. This class is important for two reasons. First, this class includes a large number of standard aggregate functions such as *min*, *max*, *sum* and *count*. Second, and perhaps more importantly, it provides an extensible way to add new aggregate functions to a database query language.

Definition 2.4 (Aggregate Functions Definable by Structural Recursion) An aggregate function $\mathbf{G} : \mathcal{M}(D) \rightarrow D'$ is said to be *definable using structural recursion* if there exist functions $f : D \rightarrow D'$ and $g : D' \times D' \rightarrow D'$ such that $\mathbf{G}$ can be defined as follows:

$$\begin{aligned} \mathbf{G}(\{a\}) &= f(a), \text{ for all } a \in D \\ \mathbf{G}(S \cup T) &= g(\mathbf{G}(S), \mathbf{G}(T)), \text{ for all nonempty } S, T \in \mathcal{M}(D). \quad \Box \end{aligned}$$

It follows from the above definition that g must be both associative and commutative. For example, the aggregate function *count* has $f(a) = 1$ and $g(x, y) = x + y$.

Theorem 2.1 *Every aggregate function that is definable by structural recursion has* $(\{insert\}, O(1))$ *absolute incremental complexity, if* f *and* g *can be computed in constant time on any values for its arguments.* $\square$

Examples of aggregate functions that satisfy the above theorem include min, max, sum and $count$.

Similar to the "insert" function g, we can also define a binary "delete" function d, and a ternary "replace" function r to capture deletion and monotonic-replacement of values from a multiset. Details are presented in the full paper.

LDL [10] allows users to define new aggregate functions by specifying the f and g functions. Our formalization generalizes the LDL approach by allowing users to also provide the functions d and r to incrementally recompute aggregates under deletion and monotonic-replacement. For example, a user can define a *sum-of-squares* aggregate function, using $f(x) = x^2, g(x, y) = x + y, d(x, y) = x - y$ and $r(x, y, z) = x - y + z$, and the system can automatically evaluate it incrementally under insertions, deletions and replacements.

Some aggregate functions, such as *average* and *variance*, which cannot be directly defined using structural recursion, can still be recomputed incrementally by computing them from other incrementally computable aggregate functions, such as *sum*, *count* and *sum-of-squares*. (This point was also noted by Dar et al. [6].)

3 Monotonic Semantics Revisited

The monotonic semantics of [14] is defined for a class of programs called "cost-consistent monotonic programs". We present the intuition here; formal definitions may be found in [14].

Some predicates of a program are defined to be *cost predicates*. For such predicates, one of the arguments is distinguished from the rest, and is called a *cost argument*; the rest of the arguments are called non-cost arguments. We represent such a predicate as $p_i(\overline{X}, C)$, where C denotes the cost argument, and $\overline{X}$ the rest of the arguments. The intuition is that the cost predicate is used in a groupby literal of the program, where an aggregate function is applied on the cost argument.

Definition 3.1 (The $\preceq$ Ordering) For each cost predicate, a domain D is specified for the cost argument, and a partial order $\preceq$ is defined on the domain s.t. $\langle D, \preceq \rangle$ is a complete lattice.

The partial order on cost arguments is extended to ground facts for the cost predicates as follows: $p(\overline{a_1}, c_1) \preceq p(\overline{a_2}, c_2)$ iff $\overline{a_1} = \overline{a_2}$ and $c_1 \preceq c_2$.

For sets of facts, a pre-order $\preceq$ is defined as follows:[2] $S_1 \preceq S_2$ iff $\forall s_1 \in S_1, \exists s_2 \in S_2$ s.t. $s_1 \preceq s_2$. $\square$

A set of facts S is said to be *cost-consistent* if there are no two facts $p_i(\overline{a}, c_1)$ and $p_i(\overline{a}, c_2)$ in S, s.t. p_i is a cost predicate and $c_1 \neq c_2$; in other words, no two facts in S differ *only* on their cost argument.

In the program of Example 1.1, the predicate $cv_1(X, Y, C)$ is a cost-predicate, with cost argument C and $\preceq$ defined as $\leq$. Then $cv_1(a, b, 1) \preceq cv_1(a, b, 2)$, and $\{cv_1(a, b, 2), cv_1(a, b, 3)\} \preceq \{cv_1(a, b, 4)\}$. Note that the set of facts $\{cv_1(a, b, 2), cv_1(a, b, 3)\}$ is not cost-consistent.

[2] This is not a partial order since it is not anti-symmetric.

Definition 3.2 (T_R and T_P) For a rule R, and a cost-consistent set of facts S, let $T_R(S)$ denote the set of facts that can be derived in one step using R and S. For a program $P = \{R_1, R_2, \ldots, R_n\}$, we let $T_P(S)$ denote $T_{R_1}(S) \cup T_{R_2}(S) \cup \ldots \cup T_{R_n}(S)$. □

The above definition can be made more precise in terms of substitutions and satisfaction, in the usual fashion.

For example, suppose we are given program P with a cost predicate q and the rules:

$$\begin{array}{ll} p(X) & :- \; q(X, Y, C). \\ r(X, T) & :- \; groupby(q(X, Y, C), [X], T = sum\langle C \rangle). \end{array}$$

and a set of facts $S = \{q(1, a, 3), q(1, b, 4)\}$. Then S is cost-consistent and $T_P(S)$ is $\{p(1), r(1, 7)\}$.

Definition 3.3 (Monotonicity and Cost Consistency) A program P is said to be *monotonic* if, given cost-consistent sets of facts S_1 and S_2,

$$S_1 \preceq S_2 \;\Rightarrow\; T_P(S_1) \preceq T_P(S_2).$$

A monotonic program P is said to be *cost-consistent* if, whenever S is cost-consistent, $T_P(S)$ is also cost-consistent. □

Definition 3.4 (Monotonic Semantics) A cost-consistent set of facts S is said to be a *pre-model* for a cost-consistent monotonic program P if $T_P(S) \preceq S$, i.e., S is "better than" $T_P(S)$. A pre-model S_1 for P is said to be a *least model* for P if for all cost-consistent sets S_i that are pre-models of P, $S_1 \preceq S_i$.

The *monotonic semantics* of a cost-consistent monotonic program P is defined as the least model of P. The least model can be shown to always exist. □

While the monotonic semantics as we have defined applies to a single program component, it is straightforward to extend the results to a multi-component program.

It is not obvious from the definition of the monotonic semantics how to compute it. It is shown in [14] that the least model of a program component P is equivalent to the least fixpoint of T_P, which itself can be computed as follows. We start with the empty set, and repeatedly apply T_P until we reach a fixpoint at some (possibly transfinite) ordinal γ. If T_P is continuous, the fixpoint can be computed in at most ω steps.

3.1 Monotonic Semantics Reformulated

If the least fixpoint computation is optimized in any manner that affects the order in which facts are derived (as happens, for instance, with variants of Semi-Naive evaluation, or with query-directed evaluation techniques, such as Magic Sets), then the set of facts derived over the various iterations could change. Unlike with usual logic programming semantics, it may even be the case that an intermediate fact that was derived using the fixpoint on the original program is not derived if the order of derivations changes, even if the final set of facts is the same.

Proving correctness of program optimizations using either the least model or the least fixpoint characterizations of the monotonic semantics can hence be quite difficult. We address the problem by presenting a new formulation of the monotonic semantics in terms of "computations", following the proof-theoretic approach of [3, 15]. It is much easier to reason about correctness of optimizations using this formulation.

3.1.1 A Proof Theoretic Approach to Semantics

The idea behind the proof-theoretic approach to semantics [3, 15] is to first define rules for inferring positive information (i.e., which facts are true) and rules for inferring negative information (i.e., which facts are false). A general notion of (bottom-up) computation is then defined as a sequence of derivation (or, proof) steps, where each derivation step uses information about which facts are true and which facts are false prior to the derivation step, and uses the positive inference rule to derive a new fact and add it to the collection of true facts. The negative inference rule can then be used to determine the set of facts that are false after the derivation step.

So long as the positive and negative inference rules satisfy some simple monotonicity properties, a program can be assigned a unique semantics based on these inference rules, along with the notion of "complete" computations, i.e., computations that cannot be extended to derive new facts. This semantics can be shown to satisfy important properties such as foundedness [3].

To reformulate the monotonic semantics, we need only the positive inference rules. The negative inference rules provide a clean way to extend the monotonic semantics to handle negation. However, for lack of space we do not examine this idea in this paper.

3.1.2 Positive Inference Rules

Definition 3.5 ($I^+(R,S)$) The *positive inference rule* $I^+(R,S)$ is a function $Rules \times Interpretations \rightarrow Interpretations$, defined as follows: $p(\overline{a}) \in I^+(R,S)$ iff $\exists S'(S' \preceq S \wedge S'$ is cost-consistent $\wedge\, p(\overline{a}) \in T_R(S'))$. □

Note that, unlike the definition of T_R, the above definition allows the use of non-cost-consistent interpretations.

Example 3.1 Suppose we are given program P consisting of the rule R:

$$r(X,T){:}-groupby(q(X,Y,C),[X],T = sum\langle C\rangle).$$

and a non-cost-consistent set of facts $S = \{q(1,a,3), q(1,a,5), q(1,b,4)\}$. Also suppose that the cost argument of q has a $\preceq$ ordering defined by $\leq$ on the integers. There are many different cost-consistent interpretations S', such that $S' \preceq S$, i.e., S is "better than" S'. $\{q(1,a,3), q(1,b,4)\}$ and $\{q(1,a,5), q(1,b,4)\}$ are two possibilities. Therefore, the facts $r(1,7)$ and $r(1,9)$ (among others) are present in $I^+(R,S)$. □

The following definition is derived from [3], and is crucial to proving uniqueness of our reformulation of the monotonic semantics.

Definition 3.6 (Derivation monotonic) An inference rule $I: Rules \times Interpretations \rightarrow Interpretations$ is said to be *derivation monotonic* if $S_1 \preceq S_2 \Rightarrow I(R,S_1) \subseteq I(R,S_2)$. □

Proposition 3.1 *Positive inference rule I^+ (see Definition 3.5) is derivation monotonic.* □

An immediate question is, what about efficiency? Should we really look at every possible S' such that $S' \preceq S$ in order to compute the monotonic semantics? The answer is no, so long as the rule R is monotonic. In fact it turns out that it suffices to use only a cost-consistent interpretation that is equivalent (under the $\preceq$ ordering) to S. In terms of the above example, we need only consider the cost-consistent interpretation $\{q(1, a, 5), q(1, b, 4)\}$. However, the above formulation helps simplify understanding the semantics and proofs of correctness of optimizations.

To make formal the notion of a cost-consistent interpretation that is equivalent (under the $\preceq$ ordering) to a given (possibly non-cost-consistent) interpretation S, we have the following definition.

Definition 3.7 (Reductions and Normal Forms) Given interpretations I and I', we say that I' is a *reduction* of I if I' is cost-consistent and $I' \preceq I$.

Interpretation I' is said to be the *normal form* of I, denoted $nf(I)$, if: (1) I' is cost-consistent, and (2) $I \preceq I' \wedge I' \preceq I$, i.e., I and I' are equivalent, under the $\preceq$ ordering. □

It is not hard to show that the normal form of an interpretation is unique and always exists.

3.1.3 Computations For Defining Semantics

We now show how to reformulate the monotonic semantics using I^+, the positive inference rule, following the framework of [3]. The first step is to define sequences of derivations, which we call pre-computations; then we define computations, which are pre-computations where each derivation uses the positive inference rule I^+; finally we define the meaning of program using the notion of "complete" computations.

Definition 3.8 (Pre-computations) A *pre-computation* C is a mapping from all ordinals less than some ordinal α to the set of pairs of the form $(R, p(\overline{a}))$, where R is a rule of the program, and $p(\overline{a})$ is a fact. The ordinal α is the *length* of the pre-computation.

We call each pair in C a *step*; $C(\beta)$ denotes step β of C, where β is an ordinal less than the length of C. If $C(\beta) = (R, p(\overline{a}))$, we use $fact(C(\beta))$ to denote $p(\overline{a})$, and $rule(C(\beta))$ to denote R. □

If a pre-computation is finite, it can be simply viewed as a sequence of pairs of the form $(R, p(\overline{a}))$. The above definition is specified in terms of mappings to handle transfinite pre-computations.

Definition 3.9 (Computation) A pre-computation C is called a *computation* if for each step $C(\beta) = (R, p(\overline{a}))$ in C, $p(\overline{a}) \in I^+(R, S)$ where $S = \bigcup_{\gamma<\beta}\{fact(C(\gamma))\}$. In other words, the fact $p(\overline{a})$ can be derived using the inference rule I^+, the program rule R and the facts previously derived in the pre-computation.

A computation C_1 is said to be a *complete* computation if there is no computation C_2 such that (a) C_1 is a proper prefix of C_2 and (b) C_2 derives a "new fact", i.e., C_2 has a step $(R, p(\overline{a}))$ such that $p(\overline{a}) \not\preceq \bigcup_\beta fact(C_1(\beta))$. □

Complete computations are used as the basis for providing a meaning to programs. Given a complete computation C, let T_C denote $\bigcup_\beta fact(C(\beta))$, i.e., the collection of all facts computed in C. We call T_C the *result* of computation C for the program P. The key to proving that all complete computations define the same result (up to equivalence under $\preceq$) is to show that concatenations of computations are also computations. This is shown using the derivation monotonicity property of positive inference rules. Hence we have the following results:

Theorem 3.2 ([3]) *For each program P, there exists a complete computation of P. Further, if the positive inference rule is derivation monotonic, all complete computations of P have the same result (up to equivalence under $\preceq$).* $\Box$

Definition 3.10 (Monotonic Semantics) Given a program P, the *monotonic semantics* of program P is defined as the normal form of the result of any complete computation of P. $\Box$

Proposition 3.3 *For each program P, the monotonic semantics of P is well-defined, i.e., it exists, and is unique.*

Further, it can be computed using a complete computation, where at each step of the computation, $I^+(R,S)$ uses only $T_R(nf(S))$, and not $T_R(S')$ for all cost-consistent $S' \preceq S$. $\Box$

An interesting part of the proof of the above proposition is that if we use only the normal form reduction, a fact that can be derived using a rule with other reductions may not be derivable; however, if we consider the program as a whole, the fact, or a "better" fact, will be derivable. This is because the program as a whole is required to be monotonic, not necessarily individual rules. Note that the monotonic semantics of a program is unique, not just unique up to equivalence under $\preceq$.

The following result justifies the use of the term "monotonic semantics" in Definition 3.10.

Theorem 3.4 *For any cost-consistent monotonic program P, the monotonic semantics (based on Definition 3.10) and the monotonic semantics according to* [14] *coincide.* $\Box$

Example 3.2 (Cheapest Path Program: Computations)
Consider the cheapest path program CP below (from Ross and Sagiv [14]).

$$
\begin{aligned}
r1 &: cost(X,X,Y,C) &&:- \ edge(X,Y,C).\\
r2 &: cost(X,Z,Y,C) &&:- \ mincost(X,Z,C1), edge(Z,Y,C2), C = C1 + C2.\\
r3 &: mincost(X,Y,C) &&:- \ groupby(cost(X,Z,Y,C1),[X,Y], C = min\langle C1\rangle).
\end{aligned}
$$

For both *cost* and *mincost*, their last arguments are defined as cost arguments with the $\preceq$ ordering given by $\geq$. Each of the following are computations from the database $D = \{edge(a,b,3), edge(b,c,4), edge(a,c,9)\}$:

$$
\begin{aligned}
C_1 = {} & (r1, cost(a,b,b,3)), (r1, cost(b,c,c,4)).\\
C_2 = {} & (r1, cost(a,c,c,9)), (r3, mincost(a,c,9)).\\
C_3 = {} & (r1, cost(a,b,b,3)), (r1, cost(b,c,c,4)), (r3, mincost(a,b,3)),\\
& (r2, cost(a,b,c,7)), (r3, mincost(a,c,7)), (r3, mincost(b,c,4)),\\
& (r1, cost(a,c,c,9)).\\
C_4 = {} & (r1, cost(a,b,b,3)), (r1, cost(b,c,c,4)), (r1, cost(a,c,c,9)),\\
& (r3, mincost(a,b,3)), (r3, mincost(b,c,4)), (r3, mincost(a,c,9)),\\
& (r2, cost(a,b,c,7)), (r3, mincost(a,c,7)).
\end{aligned}
$$

Computations C_1 and C_2 are not complete computations; computation C_3, for example, is an extension of C_1 that derives new facts. Both computations C_3 and C_4 are complete computations. The result of C_3 is $\{cost(a,b,b,3), cost(b,c,c,4), cost(a,c,c,9), cost(a,b,c,7), mincost(a,b,3), mincost(b,c,4), mincost(a,c,7)\}$, and the result of C_4 contains $mincost(a,c,9)$ in addition to all the facts in the result of C_3. Note that the normal forms of the results of these computations are identical, given by the result of computation C_3. $\Box$

3.2 Using The Reformulated Semantics for Optimization

Our reformulation of the monotonic semantics makes it easier to show that an optimization technique is correct. The question "what happens if I made a change in the order of derivations, and a fact that was derived earlier is no longer derived?" is settled as follows. The monotonicity properties of I^+ are used to show that if the fact derived earlier is not derived with the change in order of derivations, a fact that is better than it (in the $\preceq$ ordering) will be derived.

4 Incremental Evaluation of Monotonic Programs

In this section we present an efficient incremental evaluation technique for monotonic programs, based on the Semi-Naive evaluation technique (see [1], for example). It is straightforward to use rule evaluation techniques developed for programs with aggregation to define the function $T_P(I)$ for a normal form interpretation I. We separate the program into two parts: a set of rules P and a set of facts D, called the database.[3]

The following procedure defines our evaluation algorithm for a single program component. Multi-component programs can be evaluated component-by-component in a straightforward way.

```
Procedure IncrEvalMonotonic (P,D)
  Let I = nf(T_P(D)) ; Let OldI = ∅
  While (OldI ≠ I)
    OldI = I ; I = nf(T_P(D ∪ I))
  return I;    /* The result of the evaluation of P on D */
```

An important point (from efficiency considerations) is that the evaluation procedure uses $T_P(D \cup I)$, not $T_P(I')$ for all $I' \preceq D \cup I$, to make new derivations. The computation of $T_P(D \cup I)$ in the above procedure is carried out incrementally as follows. We assume that the program is pre-processed by moving each groupby literal in the program into a separate rule by itself. For rules without groupby literals, Semi-Naive evaluation [1] is used to perform incremental evaluation. For rules with the groupby literals, incremental evaluation is done using incremental aggregation techniques.

The normal form of an interpretation can also be maintained incrementally during evaluation by means of an "extended subsumption check"; whenever a fact $p(\overline{a}, c1)$ is inserted, we check to see if there already exists a fact $p(\overline{a}, c2)$. If there is such a fact and $c1 \preceq c2$, we discard $p(\overline{a}, c1)$. If $c2 \preceq c1 \wedge c1 \neq c2$, we replace $p(\overline{a}, c2)$ by $p(\overline{a}, c1)$. Hence, in each iteration of the evaluation, $D \cup I$ is cost-consistent. Evaluation terminates when no "new" facts are derived in an iteration.

Theorem 4.1 *If a program P is monotonic and cost-consistent, incremental evaluation of the program using* Procedure IncrEvalMonotonic *is sound, i.e., the result of the procedure is contained in the monotonic semantics of P. Further, the evaluation is complete whenever it terminates, i.e., the monotonic semantics of P is contained in the result of the procedure.* □

The proof of soundness basically shows that the evaluation can be mapped to a computation. The idea behind the proof of completeness is to take an evaluation

[3] This is in keeping with the convention in deductive database literature, and helps distinguish between the (usually small) program and the (potentially very large) database.

that uses only normal form interpretations in each iteration, and which cannot be extended using computations that use normal form interpretations, and show that it cannot be extended using arbitrary computations either.

Example 4.1 (Company Controls Program: Revisited)
Consider again the company controls program from Example 1.1, with the same dataset (shown in Figure 1).

As the evaluation proceeds, facts of the form

$controls(0,1), controls(0,2), \ldots, controls(0,n-2), controls(0,n-1)$

are derived, each in a separate iteration. Correspondingly, facts

$cv_1(0,n,1/2n), cv_1(0,n,2/2n), cv_1(0,n,3/2n), \ldots, cv_1(0,n,(n-1)/2n)$

are derived, and finally a fact $cv_1(0,n,(n+1)/2n)$ is derived. Using the evaluation strategy of [14], the cost of computing $cv_1(0,n,i/2n), 1 \leq i \leq (n-1)$, would require computing the sum of i numbers which has cost $\Theta(i)$. Hence, the total cost of computing facts of the form $cv_1(0,n,\_)$ would be $\Theta(n^2)$.

The incremental evaluation makes use of the fact that $cv_1(0,n,(i-1)/2n)$ can be updated to $cv_1(0,n,i/2n)$ in constant time, instead of recomputing the aggregate function from scratch. Thus, the total cost of computing facts of the form $cv_1(0,n,\_)$ is only $O(n)$, as is the total cost of the incremental evaluation procedure. The use of incremental aggregates results in a reduction of the asymptotic time complexity.

Further, the incremental evaluation is similar to Semi-Naive evaluation in that it does not repeat any derivation steps. □

5 Magic Sets: Adding Goal Directed Behavior

One of the main optimizations performed in a bottom-up evaluation is the specialization (using, e.g., Magic Sets) of the program with respect to the query so that the evaluation will generate only facts that are in some way "relevant" to answering the query. We assume familiarity with the Magic Sets transformation, and refer the reader to [2] for more details.

Consider, for example, the company controls program with the following additional rule:

$$\begin{aligned} controls(X,Y) :- \ & groupby(cv(X,Z,Y,N),[X,Y],S = sum\langle N\rangle), \\ & S < 0.3, controls(Y,X), false \end{aligned}$$

where $false$ is a predicate that is always defined to fail. The resulting program is still monotonic because this rule cannot be used to compute any facts. Given a query of the form ? $controls(c1,c2)$, if a left-to-right subgoal evaluation order is used, one of the rules in the Magic Sets transformed program is:

$$\begin{aligned} m\_controls(Y,X) :- \ & m\_controls(X,Y), groupby(cv(X,Z,Y,N), \\ & [X,Y], S = sum\langle N\rangle), S < 0.3 \end{aligned}$$

This rule is not monotonic, and the presence of this rule makes the Magic Sets rewritten program non-monotonic.

We can define a sub-class of monotonic programs that does not have this problem, in a fashion similar to the definition of left-to-right modularly stratified programs [13].

We refer to these programs as left-to-right monotonic programs, and show that the corresponding Magic Sets transformed programs are monotonic; details are presented in the full paper.

A second problem arises if the sip strategy binds cost arguments of predicates. Such a sip strategy could result in a Magic Sets transformed program with the rule:[4]

$$p(X, C) :- \; m\_p(X, C), p1(X, C).$$

where both p and $p1$ are defined in the original program to have cost arguments. This rule would make the transformed program non-monotonic, even if the original program is monotonic, whether the second argument of the predicate m_p is defined to be a cost argument or not. This problem can be avoided if the Magic Sets transformation considers only sip strategies that do not bind cost arguments of predicates. We call such sip strategies as *cost-restricted* sip strategies.

The main result of this section is the following:

Theorem 5.1 *Consider a left-to-right monotonic program component P. Let MP be the Magic Sets transformation of P using left-to-right cost-restricted sip strategies. Then, MP is a left-to-right monotonic program component, and is equivalent to P w.r.t. the query predicate.* □

Our reformulation of the monotonic semantics is the key to our proof of correctness of Magic Sets rewriting for monotonic programs.

The monotonic semantics is defined for individual components of a program containing aggregation; the semantics of a multi-component program is then defined in a component-by-component fashion. However, Magic Sets transformations do not "preserve" components; given a program with two components, for example, it is possible that the Magic Sets transformed program combines the two into a single component and the resultant program may not even be monotonic. (For the same reason that the Magic Sets transformation of a stratified program may be non-stratified.) We can show that techniques used for evaluating the Magic Sets transformation of stratified programs can be used (essentially unchanged) to evaluate the Magic Sets transformation of left-to-right monotonic programs.

6 Related Work

Mumick et al. [9] define a sub-class of monotonic programs, the *r-monotonic* programs, where rules with groupby literals in the body cannot have the aggregated value appearing in the head of the rule. They also present a bottom-up fixpoint procedure to compute this semantics, and show that the Magic Sets transformation preserves r-monotonicity.

Ganguly et al. [7] define the class of *cost-monotonic* programs, which have only the *min* and *max* aggregate functions, and is incomparable with the class of monotonic programs. They also present an efficient evaluation procedure for the class of cost-monotonic programs, that uses a form of control to ensure that a derived fact does not need to be replaced by a "better" fact subsequently in the derivation.

Koestler et al. [8] have independently proposed a differential fixpoint operator, which allows the user to specify a "subsumption" meta-predicate and evaluate an aggregate-free program by eliminating tuples that are subsumed by previously derived tuples. While their formalism does not allow explicit aggregates, it can simulate *min* and *max* aggregate functions.

[4] The original rule can be obtained by deleting the $m\_p(X, C)$ literal in the rule body.

7 Discussion and Conclusions

The techniques we developed in this paper have several applications outside of efficient evaluation of queries under the monotonic semantics. First, the techniques developed for incremental computation of aggregate values can be used to enhance existing incremental view maintenance/integrity verification techniques (see, e.g., [5]).

Second, the proof-theoretic reformulation of the monotonic semantics can form the basis for extending the monotonic semantics in several directions that we are pursuing; e.g., to deal with a larger class of programs with aggregation, and to allow negation. For example, a version of the cheapest path program which also computes the actual cheapest path, in addition to the cost of the path, can be expressed as follows. An extra argument containing a list of the nodes in the path is added to the *cost* and the *mincost* predicates. However, the $\preceq$ ordering needs to be extended to ignore this path argument. (This extension enhances expressivity, but may make checking for monotonicity harder.)

The monotonic semantics does not deal with program components with negation. Deductive database systems, such as Coral, allow both aggregation and negation in program components. An interesting direction of future research is to integrate the monotonic semantics for aggregation with semantics developed for negation, to derive a more general semantics that is still efficiently computable.

Acknowledgements

We wish to thank Shaul Dar whose careful reading of an earlier version of the paper considerably improved the presentation of this paper. The research of Raghu Ramakrishnan was supported by a David and Lucile Packard Foundation Fellowship in Science and Engineering, a Presidential Young Investigator Award with matching grants from DEC, Tandem and Xerox, and NSF grant IRI-9011563. The research of Kenneth Ross was supported by NSF grant IRI-9209029, by a grant from the AT&T Foundation, by a David and Lucile Packard Foundation Fellowship in Science and Engineering, and by a Sloan Foundation Fellowship.

References

[1] I. Balbin and K. Ramamohanarao. A generalization of the differential approach to recursive query evaluation. *Journal of Logic Programming*, 4(3):259–262, Sept. 1987.

[2] C. Beeri and R. Ramakrishnan. On the power of Magic. *Journal of Logic Programming*, 10(3&4):255–300, 1991.

[3] C. Beeri, R. Ramakrishnan, D. Srivastava, and S. Sudarshan. A proof theoretic approach to semantics for logic programs. Submitted for publication. Parts of the paper appeared in 'The Valid Model Semantics for Logic Programs' (PODS'92), and in 'Extending the Well-Founded and Valid Semantics for Aggregation' (ILPS'93)., 1994.

[4] A. M. Berman, M. C. Paull, and B. G. Ryder. Proving relative lower bounds for incremental algorithms. *Acta Informatica*, 27:665–683, 1990.

[5] F. Bry, R. Manthey, and B. Martens. Integrity verification in knowledge bases. In *Logic Programming*, pages 114–139, 1992. LNAI 592.

[6] S. Dar, R. Agrawal, and H. V. Jagadish. Optimization of generalized transitive closure queries. In *Proceedings of Seventh IEEE International Conference on Data Engineering*, Kobe, Japan, 1991.

[7] S. Ganguly, S. Greco, and C. Zaniolo. Minimum and maximum predicates in logic programming. In *Proceedings of the ACM Symposium on Principles of Database Systems*, 1991.

[8] G. Koestler, W. Kiessling, H. Thone, and U. Guntzer. The differential fixpoint operator with subsumption. In *Proceedings of the International Conference on Deductive and Object-Oriented Databases*, 1993.

[9] I. S. Mumick, H. Pirahesh, and R. Ramakrishnan. Duplicates and aggregates in deductive databases. In *Proceedings of the Sixteenth International Conference on Very Large Databases*, Aug. 1990.

[10] S. Naqvi and S. Tsur. *A Logical Language for Data and Knowledge Bases.* Principles of Computer Science. Computer Science Press, New York, 1989.

[11] R. Ramakrishnan, D. Srivastava, and S. Sudarshan. CORAL: Control, Relations and Logic. In *Proceedings of the International Conference on Very Large Databases*, 1992.

[12] G. Ramalingam. *Bounded Incremental Computation.* PhD thesis, University of Wisconsin, Madison, Aug. 1993. Technical Report #1172.

[13] K. Ross. Modular Stratification and Magic Sets for DATALOG programs with negation. In *Proceedings of the ACM Symposium on Principles of Database Systems*, pages 161–171, 1990.

[14] K. Ross and Y. Sagiv. Monotonic aggregation in deductive databases. In *Proceedings of the ACM Symposium on Principles of Database Systems*, pages 114–126, 1992.

[15] S. Sudarshan, D. Srivastava, R. Ramakrishnan, and C. Beeri. Extending the well-founded and valid model semantics for aggregation. In *Proceedings of the International Logic Programming Symposium*, 1993.

[16] A. Van Gelder. The well-founded semantics of aggregation. In *Proceedings of the ACM Symposium on Principles of Database Systems*, pages 127–138, 1992.

[17] A. Van Gelder. Foundations of aggregation in deductive databases. In *Proceedings of the International Conference on Deductive and Object-Oriented Databases*, 1993.