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

[19] L. Vieille. Recursive query processing: The power of logic. *Theoretical Computer Science*, pages 1–53, 1989.

[20] L. Vieille, P. Bayer, and V. Küchenhoff. Integrity checking and materialized views handling by update propagation in the EKS-V1 system. Technical report, CERMICS - Ecole Nationale Des Ponts Et Chaussees, France, June 1991. Rapport de Recherche, CERMICS 91.1.

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

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

[5] D. Kemp and P. Stuckey. Semantics of logic programs with aggregates. In *Procs. of the International Logic Programming Symposium*, pages 387–401, San Diego, CA, U.S.A., Oct. 1991.

[6] J. Kerisit and J. Pugin. Efficient query answering on stratified databases. In *Proc. of the International Conference on Fifth Generation Computer Systems*, pages 719–725, Tokyo, Japan, November 1988.

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

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

[9] G. Phipps, M. A. Derr, and K. A. Ross. Glue-NAIL!: A deductive database system. In *Procs. of the ACM SIGMOD Conf. on Management of Data*, pages 308–317, 1991.

[10] H. Przymusinska and T. Przymusinski. Weakly perfect model semantics for logic programs. In *Proceedings of the Fifth International Conference/Symposium on Logic Programming*, 1988.

[11] R. Ramakrishnan, D. Srivastava, and S. Sudarshan. Rule ordering in bottom-up fixpoint evaluation of logic programs. In *Procs. of the International Conf. on Very Large Databases*, Aug. 1990.

[12] R. Ramakrishnan, D. Srivastava, and S. Sudarshan. Controlling the search in bottom-up evaluation. In *Joint Int'l Conf. and Symp. on Logic Programming*, 1992.

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

[14] J. Rohmer, R. Lescoeur, and J. M. Kerisit. The Alexander method — a technique for the processing of recursive axioms in deductive database queries. *New Generation Computing*, 4:522–528, 1986.

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

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

[17] S. Sudarshan and R. Ramakrishnan. Aggregation and relevance in deductive databases. In *Procs. of the International Conf. on Very Large Databases*, Sept. 1991.

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.) One way to evaluate the program would be to put the components into separate "modules" and use Magic rewriting only within a module, treating predicate defined outside the module as database predicates; a call mechanism (as in Coral [13]) can be used to evaluate inter-module queries. In fact, 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.

A final point to note is that, since Magic Sets transformations are used to mimic top-down evaluation strategies, the results of this section are also applicable to memoing, top-down strategies for evaluating monotonic programs.

7 Conclusions

We examined the issue of efficient evaluation of queries with aggregate functions. Our techniques can be used for non-recursive view maintenance and triggers as well as queries on monotonic programs with recursive aggregates. Many of the techniques described for monotonic programs with recursive aggregates have been implemented in the Coral deductive database system [13]; incremental insertions are handled, but incremental deletion, replacement and monotonic-replacement need to be incorporated.

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, and by a David and Lucile Packard Foundation Fellowship in Science and Engineering.

References

[1] N. Arni, K. Ong, S. Tsur, and C. Zaniolo. The LDL++ system: Rationale, technology and applications. (Submitted), 1993.

[2] I. Balbin, G. S. Port, K. Ramamohanarao, and K. Meenakshi. Efficient bottom-up computation of queries on stratified databases. Journal of Logic Programming. To Appear.

not have this problem. We refer to these programs as left-to-right monotonic programs, and show that the corresponding Magic Sets transformed programs are monotonic; details are omitted here because of space considerations.

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:[7]

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

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 6.1 *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.* $\Box$

The equivalence of P and MP (w.r.t. the query predicate) can be proved by showing that any complete computation of MP can be transformed to a "query-complete" computation of P, i.e., no extension of this computation can derive any more answers to the query, even though it may derive other new facts. The proofs of correctness for Magic Sets presented in [**?**] use the concept of proof trees to show that every "relevant" fact is derived. In the context of monotonic programs, not every "relevant" fact may be derived; the use of Magic Sets can reorder the computation and result in some intermediate facts not being derived, much like in the example presented in Section 4.2.1. However, our proofs use the fact that complete computations are closed under the I^+ operator to show that even if a "relevant" fact is absent, a fact "better" than it will be present. The fixpoint formulation of the semantics is much more procedural than our formulation, and showing the above result would therefore be much harder if we used the fixpoint formulation. Thus our reformulation of the monotonic semantics is the key to our proof of correctness of Magic Sets rewriting for monotonic programs.

Recall that 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

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

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. [?] have independently proposed a differential fixpoint operator that applies to a variant of monotonic programs. However, their formalism does not allow explicit aggregates. They allow 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 this approach can simulate *min* and *max* aggregate functions, it does not generalize to more general aggregate functions.

6 Magic Sets: Adding Goal Directed Behavior

One of the main optimizations performed by a bottom-up evaluation is the specialization 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. A common specialization technique is the Magic Sets transformation, which is used to imitate top-down evaluations using bottom-up evaluation. After the transformation has been performed, bottom-up techniques can be applied to evaluate the transformed program. We assume familiarity with the Magic Sets transformation, and refer the reader to [?] for more details.

Magic Sets transformations use sideways information passing (sip) strategies to propagate bindings. Intuitively, for a rule of a program, a sip strategy represents a decision about the order in which the literals of the rule body are to be evaluated. If arbitrary sip strategies are used, the Magic Sets transformation of a monotonic program may not be monotonic. Using examples, we outline the problems with the use of arbitrary sip strategies. We then describe the restrictions under which the Magic Sets transformation of a monotonic program preserves monotonicity.

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 sip strategy 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 in a fashion similar to the definition of left-to-right modularly stratified programs [15], which does 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 [16], 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. □

Note that our optimization can only reduce the number of arithmetic operations performed, and therefore cannot do worse than the fixpoint evaluation.

Variants of Semi-Naive evaluation, such as Predicate Semi-Naive (PSN) evaluation [11] can also be used to further optimize the incremental evaluation of monotonic programs. However, since PSN evaluation changes the order in which facts are derived, the proof of correctness of this optimization makes essential use of our reformulation of the monotonic semantics.

5.1 Related Work

Mumick et al. [7] 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. The cheapest path program (Example 4.1), for instance, is not r-monotonic. Although the company controls program as described in this paper is not r-monotonic, it can be rewritten (by combining the rules defining *controls* and cv_1) such that it is r-monotonic. Mumick et al. present a bottom-up fixpoint procedure to compute this semantics, and show that the Magic Sets transformation preserves r-monotonicity.

Ganguly et al. [4] define the class of *cost-monotonic* programs, which have only the *min* and *max* aggregate functions, and is incomparable with the clas of monotonic programs. For example, the cheapest path program is not cost-monotonic if edges have negative costs, although it is cost-monotonic if all edges have non-negative costs. If $b_1(C)$ (where b_1 is a base predicate) were added to the body of rule $r2$ of the cheapest path program (see Example 4.1), the program would still be cost-monotonic (if edges have non-negative costs), but it would no longer be monotonic. Ganguly et al. present an efficient evaluation procedure for the class of cost-monotonic programs, that uses a

```
Procedure IncrEvalMonotonic (P,D)
  Let I = nf(T_P(D))
  Let OldI = ∅
  While (OldI ≠ I) {
    OldI = I ;
    I = nf(T_P(D ∪ I))      /* T_P(D ∪ I) is computed incrementally as described below*/
  }
  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 preprocessed as described in Section 3. For rules without groupby literals, Semi-Naive evaluation [3] is used to perform incremental evaluation. For rules with the groupby literals, incremental evaluation is done as described in Section 3. In each iteration, only those derivation steps are made that were not performed in previous iterations.

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 $c2 \preceq c1 \land c1 \neq c2$, 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 5.1 *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 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 5.1 (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

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

c,
$7)\}$, and $\{cost(a,b,b,3), cost(b,c,c,4), cost(a,c,c,9), mincost(a,b,3), mincost(b,c,4), cos$

$mincost(a,c,7), mincost(a,c,9)\}$, respectively. Note that the normal forms of the results of these computations are identical, given by the result of computation C_3. Further, this is the same as the least fixpoint of program CP on database D, described in Example 4.1. □

4.3 Using The Reformulated Semantics for Optimization

Our reformulation of the monotonic semantics makes it easier to show that an optimization technique is correct. In order to prove an optimization technique correct, we now only need show that (a) an evaluation using the optimized program can be mapped to a computation on the original program (and therefore the optimization is sound), and (b) the evaluation is complete (at least with respect to the query on the program). Part (a) is usually easy to show based on the definition of I^+; we only need to show that every derivation made is correct according to I^+. Showing correctness by comparing each step with the fixpoint formulation would be much more difficult.

Part (b) is typically a little more complex, but a lot easier than showing equivalence with respect to the original formulations of the monotonic semantics. In particular, we only need show that every derivation that can be made using the normal form of the result of a computation has indeed been made, to show that it is a complete computation. 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?" can now be settled. The monotonicity properties of I^+ guarantee 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.

5 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 [3], 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.[6]

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.

[6]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.

Theorem 4.2 ([?]) *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 $\preceq$).* $\square$

Definition 4.10 (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. $\square$

Proposition 4.3 *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 $S' \preceq S$. $\square$ $\square$

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 4.10.

Theorem 4.4 *For any cost-consistent monotonic program P, the monotonic semantics (based on Definition 4.10) and the monotonic semantics according to* [16] *coincide.* $\square$

Example 4.3 (Cheapest Path Program: Computations) Consider the cheapest path program CP and the database from Example 4.1. Both *cost* and *mincost* have their last arguments as cost arguments with the $\preceq$ ordering given by $\geq$ over integers. Each of the following are computations from the database $D = \{edge(a, b, 3), edge(b, c, 4), edge(a, c, 9)\}$:

$$C_1 = (r1, cost(a, b, b, 3)), (r1, cost(b, c, c, 4)).$$
$$C_2 = (r1, cost(a, c, c, 9)), (r3, mincost(a, c, 9)).$$
$$C_3 = (r1, cost(a, b, b, 3)), (r1, cost(b, c, c, 4)), (r3, mincost(a, b, 3)), (r2, cost(a, b, c, 7)$$
$$(r3, mincost(a, c, 7)), (r3, mincost(b, c, 4)), (r1, cost(a, c, c, 9)).$$
$$C_4 = (r1, cost(a, b, b, 3)), (r1, cost(b, c, c, 4)), (r1, cost(a, c, c, 9)), (r3, mincost(a, b, 3)$$
$$(r3, mincost(b, c, 4)), (r3, mincost(a, c, 9)), (r2, cost(a, b, c, 7)), (r3, minc$$

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, with results being, $\{cost(a, b, b, 3)$, $cost(b, c, c, 4), cost(a, c, c, 9), mincost(a, b, 3), mincost(b, c, 4), cost(a, b, c, 7), mincost(a,$

Definition 4.7 (Derivation monotonic)
An inference rule $I : Rules \times Interpretations \rightarrow Interpretations$ is said to be *derivation monotonic* if $S_1 \preceq S_2 \;\Rightarrow\; I(R, S_1) \subseteq I(R, S_2)$. $\Box$

Proposition 4.1 *Positive inference rule I^+ (see Definition 4.5) is derivation monotonic.* $\Box$

4.2.4 Computations For Defining Semantics

We now show how to reformulate the monotonic semantics using I^+, the positive inference rule, following the framework of [?]. 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 4.8 (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. $\Box$

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 of pre-computations is specified in terms of mappings to handle transfinite pre-computations as well.

Definition 4.9 (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 $p(\overline{a}) \not\preceq \bigcup_\beta fact(C_1(\beta))$. $\Box$

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:

in the $\preceq$ ordering is present in the meaning of the program. We use a notion very much like subsumption when defining our positive inference rule, to resolve the problem. Further, we consider interpretations that may not be cost-consistent, when defining the inference rule.

Definition 4.5 ($I^+(R, S)$)
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$\wedge$ $p(\overline{a}) \in T_R(S'))$. □

The above definition essentially extends the definition of T_R by allowing the use of non-cost-consistent interpretations. We only consider cost-consistent interpretations S' for T_R, since the monotonic semantics requires that aggregation be applied only on cost-consistent interpretations.

Example 4.2 Suppose we are given program P consisting of the rule R:

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

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 $I^+(R, S)$. □

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 $\{q(1, a, 5), q(1, b, 4)\}$.

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 4.6 (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.

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

formulation of the monotonic semantics in terms of "computations", following the approach of [**?**, **?**]. This formulation imposes few restrictions on the order in which derivations are made, and it is much easier to reason about correctness of optimizations using this formulation.

4.2.2 A Proof Theoretic Approach to Semantics

Our reformulation is based on a proof-theoretic approach to defining semantics, described in [**?**, **?**]. The idea behind the technique 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 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, and 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 [**?**].

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 follow up on the idea in this paper.

4.2.3 Positive Inference Rules

A positive inference rule is used to deduce what is true, given a set of facts that have already been deduced to be true.[5]

In the least fixpoint computation of the monotonic semantics, a fact may be derived in an intermediate iteration, but may not be true in the meaning of the program. In such a case, the meaning of the program will always contain some "better" fact. For example, in the cheapest path program of Example 4.1, $mincost(a, c, 9)$ is present in $T^4_{CP}(\emptyset)$, but is not present in the meaning of CP. However, the "better" fact $mincost(a, c, 7)$ is present.

In contrast, the proof-theoretic approach to semantics presented in [**?**] requires that if a fact is derivable by a positive inference rule at some stage in a computation, it must be derivable using the rule later in the computation as well. The apparent contradiction is resolved if we realize that in the least fixpoint iteration, if an intermediate fact is deleted, a fact that is "better"

[5] In general, a positive inference rule also takes as input a set of facts that have already been deduced to be false. However, since we do not deal with negation, and only deal with monotonic aggregation, this set of facts is not important here.

$$
\begin{aligned}
T^2_{CP}(\emptyset) &= D \bigcup \{cost(a,b,b,3), cost(b,c,c,4), cost(a,c,c,9)\}.\\
T^3_{CP}(\emptyset) &= D \bigcup \{cost(a,b,b,3), cost(b,c,c,4), cost(a,c,c,9), mincost(a,b,3), mincost\\
&\qquad mincost(a,c,9)\}.\\
T^4_{CP}(\emptyset) &= D \bigcup \{cost(a,b,b,3), cost(b,c,c,4), cost(a,c,c,9), mincost(a,b,3), mincost\\
&\qquad mincost(a,c,9), cost(a,b,c,7)\}.\\
T^5_{CP}(\emptyset) &= D \bigcup \{cost(a,b,b,3), cost(b,c,c,4), cost(a,c,c,9), mincost(a,b,3), mincost\\
&\qquad cost(a,b,c,7), mincost(a,c,7)\}.\\
T^6_{CP}(\emptyset) &= T^5_{CP}(\emptyset).
\end{aligned}
$$

Hence, $T^5_{CP}(\emptyset)$ is the meaning of the cheapest path program on the given database D. Note that the fact $mincost(a,c,9)$ which is present in $T^4_{CP}(\emptyset)$ is not present in the least fixpoint of the program; it has been replaced by the "better" fact $mincost(a,c,7)$. □

4.2 Monotonic Semantics Reformulated

We now present a reformulation of the monotonic semantics after motivating the reformulation.

4.2.1 Why Reformulate the Monotonic Semantics?

The least model formulation of the monotonic semantics does not directly lend itself to computation. Although the least fixpoint formulation is amenable to computation, it does not incorporate common query optimizations such as Semi-Naive evaluation [3] and Magic Sets transformation [**?**]. The formulation is in terms of a sequence of iterations, and if the 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.

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. For example, if we controlled the order of derivations in Example 4.1 to postpone the derivation of $mincost(a,c,9)$, we would generate the fact $cost(a,b,c,7)$ and directly generate $mincost(a,c,7)$ without ever generating $mincost(a,c,9)$. The optimization techniques of [4] would actually perform precisely the above reordering of the evaluation. It is hard to determine whether such optimizations are sound for cost-consistent monotonic programs.

Evaluation optimizations are very important for efficient query evaluation. For example, many common logic programs have an infinite set of facts in their semantics, but have a finite set of answers to typical queries. Without query-directed program evaluation, such programs are not of much practical use.

How then can we apply such optimization techniques to programs under the monotonic semantics? We address the problem by presenting a new

Definition 4.4 (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. $\Box$

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.

4.1.1 Computational Definition of Monotonic Semantics

It is not obvious from the definition of the monotonic semantics how to compute it. It is shown in [16] that the least model is equivalent to the least fixpoint of T_P, which itself can be computed as follows. Define $T_P^{\alpha+1}(S) = T_P(T_P^{\alpha}(S))$, for successor ordinals $\alpha+1$, and define $T_P^{\beta}(S) = \bigcup_{\alpha<\beta} T_P^{\alpha}(S)$, for limit ordinals β. Then there is a γ (possibly transfinite) such that $T_P^{\gamma+1}(\emptyset) = T_P^{\gamma}(\emptyset)$, i.e., $T_P^{\gamma}(\emptyset)$ is a fixpoint of T_P, and is the least, in the $\preceq$ ordering, of all fixpoints of T_P. Further, the least fixpoint of T_P is the same as the least model of P.

What the above formulation of the least model provides is a way of computing the monotonic semantics. 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. Clearly if γ as above is not finite, the computation will not terminate.

Example 4.1 (Cheapest Path Program) Consider the cheapest path program CP below (from Ross and Sagiv [16]) in which we have a database relation *edge*, describing a directed graph with edge costs. A fact of the form $edge(s,d,c)$ represents the information that there is a directed edge from source s to destination d with cost c. This information can be used to compute cheapest directed paths between nodes in the graph. The program has the following rules:

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

Suppose the program also has a set of facts $D = \{edge(a,b,3), edge(b,c,4), edge(a,c,9$ The least fixpoint of the program, T_{CP}, can be iteratively computed as follows:

$$T_{CP}(\emptyset) = D.$$

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 $c_1 \preceq c_2$.

The ordering is further extended to sets of facts as follows: $S_1 \preceq S_2$ iff $\forall s_1 \in S_1, \exists s_2 \in S_2$ s.t. $s_1 \preceq s_2$. $\Box$

Note that the ordering on sets of facts is not a partial order (since it is not anti-symmetric), but is a pre-order.

Recall that the partial order $\preceq$ on domain D is the "better than" relation defined by the aggregate function applied on the cost argument, for monotonic-replacement operations; if $c_1 \preceq c_2$, then c_2 is said to be "better than" c_1. For example, let $p(X, C)$ be a predicate with cost argument C, and let the domain D of C be the domain of natural numbers. Suppose $\preceq$ is defined as $\leq$ for this domain. Then $p(1, 1) \preceq p(1, 2)$, and $\{p(1, 2), p(1, 3)\} \preceq \{p(1, 4)\}$;

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 above example, the set of facts $\{p(1, 2), p(1, 3)\}$ is not cost-consistent.

Definition 4.2 (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 $T_{R_1}(S) \cup T_{R_2}(S) \cup \ldots \cup T_{R_n}(S)$. $\Box$

The above definition can be made more precise in terms of substitutions and satisfaction, in the usual fashion. Suppose we are given program P with a cost predicate q and the rules:

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

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 4.3 (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, $T_P(S)$ is also cost-consistent. $\Box$

We have defined the class of programs for which the monotonic semantics is defined, and now we define the semantics itself. In [16] the semantics is actually defined in two ways, which are shown to be equivalent. The first is a least model approach, and the second is a least fixpoint approach.

groupby body literal. We could use the technique described above for stratified programs, but cannot guarantee that this technique will terminate. After introducing the monotonic semantics (Section 4) we describe a technique to efficiently evaluate monotonic programs (Section 5); the only update operations we support are insertion and monotonic-replacement. The general case of efficiently evaluating arbitrary programs with aggregation is harder and we do not consider it in this paper.

4 Monotonic Semantics Revisited

The monotonic semantics of Ross and Sagiv [16] provides an intuitive semantics for a wide range of programs that have recursion through aggregation, such as the cheapest path program and the company controls program. The monotonic semantics provides ease of expression for a number of problems, and the semantics is itself natural and easy to understand. Furthermore, it has a potential for efficient evaluation, unlike semantics for for more general classes of unstratified aggregation. The potential was not fully realized earlier, but the evaluation techniques we present help realize it.

In this section, we first briefly describe the original least model and least fixpoint formulations of the monotonic semantics (Section 4.1). These formulations are not amenable to common query optimizations such as Semi-Naive evaluation [3] and Magic Sets transformation [**?**]. Hence, in Section 4.2, after motivating the need for a reformulation, we present a reformulation of the monotonic semantics in terms of computations, i.e., sequences of derivation steps, following the approach outlined in [**?**, **?**]. This makes the task of proving correctness of our program optimizations much easier, as outlined in Section 4.3.

4.1 Ross and Sagiv's Monotonic Semantics

We present the intuition behind the monotonic semantics of [16]. The formal definitions may be found in [16]. The monotonic semantics is defined for a class of programs called "cost-consistent monotonic programs", which we define below.

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*. 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. For example, the predicates $cv(X, Z, Y, C)$ and $cv_1(X, Y, C)$ in the company controls program of Example 1.1 are cost predicates, and their last arguments (denoted by C in both cases) are the cost arguments.

Definition 4.1 (The $\preceq$ Ordering) For each cost predicate, a domain D

would be rewritten into the rules:

$$R_i: \quad q(\overline{A}) \qquad : - \ \ldots, p_i(\overline{X}, S), \ldots.$$
$$AR_i : p_i(\overline{X}, S) : - \ groupby(p(\overline{X}, \overline{Z}, Y), [\overline{X}], S = \mathbf{G}\langle Y \rangle).$$

where p_i is a new predicate. The program is thus partitioned into rules without groupby literals, and aggregation rules with a single groupby body literal. This rewriting simplifies incremental evaluation.

Consider a preprocessed rule of the form of rule AR_i above: during the evaluation, we maintain the extent of $p_i(\overline{X}, S)$, and with each tuple in p_i, we maintain any additional information that is needed for the incremental computation of the aggregate function $\mathbf{G}$. For example, if $\mathbf{G}$ is *average*, the additional information would include the values of *sum* and *count*.

If any updates are performed on relation p, we have to update p_i as well. For example, an insertion of fact $p(\overline{a}, \overline{b}, c)$ is handled as follows. If there exists a fact $p_i(\overline{a}, \_)$, this fact is updated using the incremental insertion operation for the aggregate function $\mathbf{G}$. If there does not exist such a fact, a new fact $p_i(\overline{a}, s)$ is created where $s = \mathbf{G}(\{c\})$. Deletions, replacements and monotonic-replacements are handled using their respective incremental update operations.

3.3 Combining Incremental Aggregation With Incremental View Maintenance

Several techniques have been proposed in the literature for incremental evaluation of programs without aggregation. For non-recursive programs, exact techniques are known for both insertion and deletion from relations [**?**]. For the case of recursive programs, insertion is handled by Semi-Naive evaluation (e.g., [3]). Deletion is harder, and several solutions have been proposed [**?**, **?**] which may repeat computation, i.e. they may delete a derived fact and rederive it during an incremental update.

The important point to note is that these techniques are *orthogonal* to our techniques for recomputing aggregates and the two can be used in conjunction. The basic idea is to use techniques for incremental program evaluation on rules other than the aggregation rules in the rewritten program, and use our incremental aggregation techniques on the aggregation rules.

Consider first the case of stratified aggregation. The incremental program evaluation technique could call the incremental aggregation technique with a multiset of updates on the relation used in the groupby body literal. The incremental aggregation technique computes a multiset of updates on the head predicate of the rule, and returns it. Since aggregation is stratified, the updates on the head of the rule do not result in any further updates on the relation in the groupby literal in the body of the rule. This technique is also directly applicable for incremental maintenance of SQL views involving aggregates.

Now consider unstratified aggregation. Updates to the head relation of an aggregation rule could result in further updates to the relation in the presenting our notation for the use of aggregation in query languages in Section 3.1, we perform a preprocessing step on the program to partition the program into rules without aggregation, and rules with aggregation that have only a single body literal (Section 3.2). Existing incremental evaluation techniques are used for incremental evaluation of rules without aggregation, and our techniques for incremental computation of aggregate functions are used for rules with aggregation (Section 3.3).

Incremental aggregation techniques are also useful for applications such as implementing triggers involving aggregates, and computing aggregates on sequence data. For example, an algorithm that computes the n-day moving average on stock-market quotes could use our techniques to advantage.

3.1 Notation

SQL provides aggregation in combination with a grouping facility, which is used to partition values into groups and aggregate on the values within each group, as illustrated by the query below.

```
SELECT type, AVG(time)
FROM process
GROUP-BY type
```

The same program can be written in deductive database notation [7] as:

$$p(Type, A) :- groupby(process(Type, Id, Time), [Type], A = average\langle Time \rangle).$$

The groupby literal specifies the relation that should be aggregated upon ($process$), the attributes of the relation to group on (the arguments enclosed in [ ]), the aggregate function ($average$) and the multiset of values to apply it on (the $Time$ values within each group).

In general, a groupby literal has the following syntax [7]: $groupby(p(\overline{X}, \overline{Z}, Y), [\overline{X}], S = \mathbf{G}\langle Y \rangle)$, where $\mathbf{G}$ is an aggregate function. The tuple of variables $\overline{X}$ denotes the group-by attributes. Variables other than S and $\overline{X}$ are quantified within the literal, and are therefore not visible outside the literal. We assume for simplicity that these variables are distinct from all other variables in the rule. The semantics of the literal can be defined by a translation to SQL groupby, and we assume that the reader is familiar with the SQL groupby semantics. We use the deductive database notation in the rest of the paper.

3.2 Implementing Incremental Aggregation For Groupby Literals

In order to keep our techniques for incremental computation of aggregate functions independent of techniques for incremental program evaluation, we perform the following preprocessing on the program. The preprocessed program is obtained from the original program by moving each groupby literal in the program into a separate rule by itself. For example, a rule of the form:

$$R_i :\ q(\overline{A}) :- \ldots, groupby(p(\overline{X}, \overline{Z}, Y), [\overline{X}], S = \mathbf{G}\langle Y \rangle), \ldots.$$

Aggregate	Updates	Incremental Complexity		Space
		Absolute	Relative	Requirement
min	{insertion, monotonic-replacement}	O(1)	O(1)	O(1)
	{insertion, deletion, replacement}	O(log(n))	O(log(n))	O(n)
max	{insertion, monotonic-replacement}	O(1)	O(1)	O(1)
	{insertion, deletion, replacement}	O(log(n))	O(log(n))	O(n)
sum	{insertion, deletion, replacement}	O(1)	O(1)	O(1)
product	{insertion, deletion, replacement}	O(1)	O(1)	O(1)
count	{insertion, deletion, replacement}	O(1)	O(1)	O(1)
average	{insertion, deletion, replacement}	O(1)	O(1)	O(1)
variance	{insertion, deletion, replacement}	O(1)	O(1)	O(1)
median	{insertion, deletion, replacement}	O(log(n))	O(log(n))	O(n)
mode	{insertion, deletion, replacement}	O(log(n))	O(1)	O(n)

Table 2: Incremental Cost of Evaluating Aggregates

value changes the count of that value by 1. Given a value, the balanced tree can be used to get logarithmic-time access to the pair $\langle value, count \rangle$. The pair may move to one of the adjoining nodes, or be placed in a new node adjoining its current node, all constant time operations. Hence insertion and deletion can be done in logarithmic time. The result of this incremental algorithm is a pointer to the node containing all the values with the maximum count.[4] Further, the $O(\log(n))$ incremental complexity for each operation can be shown to be a lower bound, using a reduction from the element disjointness problem [**?**, **?**]. This reduction can also be used to show a lower bound of $O(n * \log(n))$ for computing *mode* on a multiset of n values. This demonstrates a provably optimal, constant relative incremental complexity, although the absolute incremental complexity is logarithmic.

Theorem 2.4 *The mode aggregate function is* $(\{insert, delete, replace\}, O(\log(n)))$ *incremental.* □

Table 2 provides a summary of lower and upper bounds results for updates on a variety of aggregate functions.

Theorem 2.5 *The bounds shown in Table 2 are correct and optimal worst-case bounds.* □

3 Using Incremental Aggregation

In this section we present an overview of our approach for view maintenance for programs (either SQL views or logic programs) with aggregation. After

[4] Since the number of *mode* values can be proportional to the size of the multiset, returning all the values, instead of just the pointer to the data structure, can have complexity $O(n)$.

An example of an aggregate function that is not definable directly using structural recursion, but which can still be computed incrementally is *average*. Average can be evaluated incrementally by computing it as $\Sigma x_i/n$ from incrementally maintained sum (Σx_i) and count (n). (This point was also noted by Dar et al. [**?**].) Similarly, the *variance* aggregate function can be computed as $(n\Sigma x_i^2 - (\Sigma x_i)^2)/n^2$, from incrementally maintained Σx_i^2, Σx_i and n.[3] The *standard deviation* aggregate function can be computed as the square-root of the variance.

2.4 Other Aggregate Functions

The *median* aggregate function can be incrementally evaluated by maintaining two balanced trees and a variable which holds the median value. One tree (the "left" tree) maintains values less than or equal to the current median, and the other (the "right" tree) maintains values greater than or equal to the current median. When a new value gets added to the multiset, we compare its value with the current median, and add it to one of the two trees appropriately. At this point, the "current" median may no longer be the median of the new multiset. In this case, if the left tree is bigger, its largest element is promoted to be the new median, and the old median is inserted into the right tree. The case when the right tree is bigger is symmetric. Deletion is handled similarly. This gives a time complexity of $O(\log(n))$ for each insert/delete operation. Further, Galil [**?**] has shown that the $O(\log(n))$ complexity for each operation is a lower bound, using a reduction from sorting a collection of values. The idea behind this reduction is to sort a multiset M of n values as follows. First, add $n-1$ copies of $-\infty$ to the multiset, and obtain the median; this is the smallest value in M. Then keep adding 2 copies of $+\infty$ to the multiset, and each time obtain the median. The sequence of values obtained is in sorted order. Given a collection of n values, however, the median can be computed in linear time [**?**].

Theorem 2.3 *The median aggregate function is* $(\{insert, delete, replace\}, O(\log(n)))$ *incremental.* $\square$

The *mode* aggregate function finds all values with the maximum count in a multiset. It can be incrementally evaluated by maintaining two data structures: (1) a doubly-linked list of nodes, where each node is a set of $\langle value, count\rangle$ pairs, and (2) a balanced tree of distinct values in the multiset, along with pointers to the corresponding $\langle value, count\rangle$ pair in the doubly-linked list. All pairs in a given node of the doubly-linked list have the same *count* and the list of nodes is sorted by *count*. Each node (set of pairs) is itself maintained as a doubly-linked list. Insertion or deletion of a

[3] Alternative formulations of variance may sometimes be preferable for reasons of numerical precision.

Aggregate	Domain of multiset elements	g	f	d
sum	Reals and Integers	$+$	id	$-$
product	Non-zero Reals and Non-zero Integers	$*$	id	$/$
count	Any	$+$	1	$-$

Table 1: Invertible Aggregate Functions

(in our computational model) on any given values for its arguments, then it is straightforward to devise an $(\{insert\}, O(1))$ incremental algorithm for computing structurally recursive aggregate functions, based on the functions f and g. Examples of such aggregate functions include min, max, sum and $count$. For example, the aggregate function $count$ has $f(a) = 1$ and $g(x, y) = x + y$.

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

Definition 2.6 (Invertible Aggregates) An aggregate function $\mathbf{G} : \mathcal{M}(D) \rightarrow D'$ definable by structural recursion is said to be *invertible* if there exists a function $d : D' \times D' \rightarrow D'$ such that for all nonempty $S, T \in \mathcal{M}(D)$, $\mathbf{G}(S) = d(\mathbf{G}(S \cup T), \mathbf{G}(T))$. □

An example of an invertible aggregate function is *product* on the domain of non-zero reals, where $f(a) = a$, $g(a, b) = a * b$, and $d(p, a) = p/a$. Table 1 illustrates some invertible aggregate functions, with the associated domains for the multiset elements.

Theorem 2.2 *Every invertible aggregate function $\mathbf{G}$ is $(\{insert, delete, replace\}, O(1))$ incremental, if the function d can be computed in constant time on any values for its arguments.* □

Some aggregate functions definable by structural recursion, such as min and max, are not invertible. However, these aggregates can be incrementally evaluated in constant time in the presence of monotonic-replacement. This can be formalized using a ternary "replace" function r; we do not do so here for lack of space.

By providing functions f, g, d and r users can define their own aggregate functions, which can be evaluated incrementally. The functions provide a degree of extensibility that generalizes the extensible aggregate functions supported by LDL [8]; LDL allows users to define aggregate functions by specifying the f and g functions. For example, a user can define a *sum-of-squares* aggregate function, using $f(x) = x^2, g(x, y) = x + y$ and $d(x, y) = x - y$, and the system can automatically evaluate it incrementally under insertions, deletions and replacements.

a sequence of n values, the optimal worst case time complexity of computing the *sum* is $O(n)$, and hence such an incremental aggregate algorithm also has $(\{insert\}, O(1))$ 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 [?]: "More typically, we have a 'best known' algorithm, and our [relative incremental complexity] is relative to the complexity of that algorithm."

Definition 2.4 (Incremental Aggregate Function) An aggregate function $\mathbf{G}$ is said to be $(U, O(h))$ *incremental* if: (1) there is an incremental aggregate algorithm with $(U, O(h))$ absolute incremental complexity that computes $\mathbf{G}$, and (2) $h(n)$ is asymptotically less than $t_{\mathbf{G}}(n)$. $\square$

The above definition includes the requirement that $h(n)$ be asymptotically less than $t_{\mathbf{G}}(n)$, since we are interested only in incremental algorithms where the cost of computing the aggregate incrementally using the incremental algorithm is lower than the cost of computing the aggregate from scratch using any (possibly non-incremental) 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. For example, it may be possible to compute the new sum from just the old sum and the change Δp, thus giving a constant space requirement. On the other hand, computing the new *mode* requires storing all of p.

2.3 Extensible Incremental Aggregation

Several aggregate functions are definable by structural recursion on an associative, commutative binary operator [?]. 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 2.5 (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:

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

It follows from the above definition that g must be both associative and commutative. If, in addition, f and g can be computed in constant time 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 [**?**, **?**].

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 $\mathbf{G}$. $\square$

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 make the requirement explicit below.

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.

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.* $\square$

For example, an incremental aggregate algorithm that maintains the sum after each insertion has $(\{insert\}, O(1))$ absolute incremental complexity, since each insertion and recalculation of the sum takes constant time. Given

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 $c_1 \geq c_2$. In other words, replacement of a value in a multiset by a smaller value is a monotonic-replacement for the *min* aggregate function. Recomputing the *min* value of a multiset is easier if monotonic replacement is performed than if arbitrary replacement is performed. The partial ordering for the *max* aggregate function over multisets of reals is given by $\leq$, i.e., replacement of a value in a multiset by a larger value is a monotonic-replacement for the *max* aggregate function.

Our model of computation is the Random Access Model (RAM), with the additional assumption that each of the basic arithmetic operations of $+, -, *$ and $/$ and each of the comparison operations of $<, \leq, =, \geq, >$ and $\neq$ can be performed in constant time. Thus when we refer to notions of time complexity, we are counting the number of basic arithmetic and comparison operations needed.

We will be focusing particularly on aggregate functions where the problem input is a multiset of integers or reals, and the problem output is an integer or a real. We do not assume that the domains are finite, i.e., we assume for the purpose of our lower and upper bounds results that our integers/reals are unbounded.[2] As is the case with sorting and other algorithms, if we were to assume a finite domain, then a number of the complexity bounds mentioned later in this paper could be reduced.

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

[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.

[2]Our algorithms do not encode multiple computation steps into a single arithmetic operation by using unbounded integers.

Example 1.2 (Materialized Non-recursive View) Consider a database describing the performance of some industrial process. The base relation *process* has three arguments: the *type* of an instance of the process, the *identifier* of the process instance, and the *time* taken to complete the process instance. To track the performance of the process one may define a view with the following SQL-style query:

```
SELECT type, MAX(time), AVG (time), MEDIAN(time)
FROM process
GROUP-BY type
```

As the industrial process proceeds, new tuples may be added to the base relation *process*. Additionally, some *process* tuples may change as time estimates become more accurate. Tuples in *process* may also be deleted as they go out-of-date. We would like to keep the view up-to-date without having to recompute the entire view each time the relation *process* changes; to do so, we need to compute the aggregate functions MAX, AVG and MEDIAN "incrementally" each time the base relation changes.

Techniques for computing MAX and AVG incrementally are known (e.g. [?]). We present novel techniques for incremental computation of aggregate functions such as MEDIAN and MODE. □

2 Incremental Computation of Aggregate Functions

In this section we formalize what it means to incrementally compute an aggregate function. We start by presenting basic definitions, and our model of computation (Section 2.1. We then define notions of incrementality for aggregate functions (Section 2.2). We provide a framework for incremental computation of a large class of aggregate functions which are of a simple form (Section 2.3). Finally, we provide incremental evaluation techniques for several aggregate functions outside this class and provide upper and lower bounds for incremental computation of a variety of common aggregate functions (Section 2.4).

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 $\mathcal{M}(D)$. □

Note that we do not impose any restrictions on the range of aggregate functions. For example, *min* is an aggregate function from $\mathcal{M}(\mathcal{R})$ to $\mathcal{R}$, and *count* is an aggregate function from $\mathcal{M}(\mathcal{R})$ to $\mathcal{N}$, where $\mathcal{R}$ denotes the real numbers, and $\mathcal{N}$ denotes the non-negative integers.

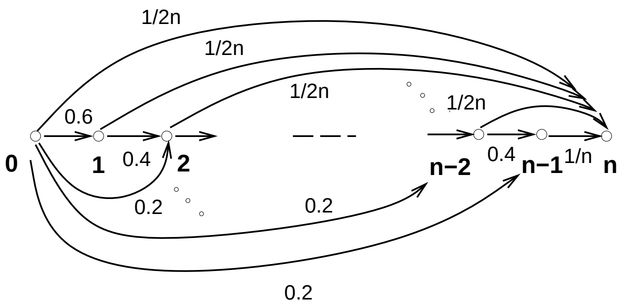

Figure 1: The *owns_stock* Relation

1.1 Motivating Examples

Example 1.1 (Company Controls Program) Consider the company controls example (modified from Mumick et al. [7]) in which we have a database relation *owns_stock* with three arguments, describing stock ownership. A fact of the form $owns\_stock(c1, c2, n)$ represents the information that company $c1$ owns fraction n of the stock of company $c2$. This information can be used to determine whether a company $c1$ owns controlling interest in company $c2$. Intuitively, a company $c1$ is defined to own controlling interest in company $c2$, if $c1$ owns (directly or indirectly through an intermediate company that $c1$ controls) more than 50% of the stock of $c2$.

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

The above program can be understood under the monotonic semantics as follows. The relation cv maintains information about (direct and indirect) stock ownership. A fact of the form $cv(X,Z,Y,N)$ means that company X has controlling interest in company Z, which directly owns fraction N of the stock of company Y; hence, X owns fraction N of the stock of Y, via Z. The relation $cv_1(Z,Y,N)$ maintains information about the total fraction N of the stock of company Y owned by company X. A fact of the form $controls(X,Y)$ indicates that this total fraction exceeds 0.5, i.e., company X is determined to have a controlling interest in company Y.

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 [16] 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 $O(n)$. $\Box$

for several aggregate functions outside this class, such as *median* and *mode*. 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 of [16] (Section 4). The least fixpoint characterization in [16] is very sensitive to the order in which facts are derived. Consequently, using these formulations, it is very difficult to show the correctness of program optimizations, such as Predicate Semi-Naive evaluation (see, e.g. [11]) and the Magic Sets transformation, that change the order in which facts are derived, resulting in a change in the set of (intermediate) facts that get computed as well.

Our reformulation of the monotonic semantics in terms of computations, following the approach of Beeri et al. [**?**], does not impose a fixed ordering on the sequence of derivation steps, and has the advantage that it allows certain reorderings of the derivation steps in a computation without changing the result of the computation, i.e., the meaning of the program. The reformulation allows for a better understanding of program optimizations, and helps us prove the correctness of the optimizations.

Our third contribution is to show how a broad class of programs with aggregate functions can be efficiently implemented (Sections 3 and 5). We perform a preprocessing step on the program to partition the program into rules without aggregation, and rules with aggregation that have only a single body literal. Existing incremental evaluation techniques, e.g., Semi-Naive evaluation [3], are used for incremental evaluation of rules without aggregation, and our techniques for incremental computation of aggregate functions are used for rules with aggregation.

The above approach provides the first technique (to our knowledge) to efficiently evaluate programs under the monotonic semantics of [16]. It also enables the efficient integration of the incremental evaluation procedure into existing deductive database systems that support Semi-Naive evaluation, such as Aditi [**?**], Coral [13], LDL and LDL++ [8, 1] and Glue-Nail! [9].

Our techniques can also be used to improve the efficiency of algorithms for view maintenance of SQL queries with aggregates (e.g., [**?**, **?**]), and activation of triggers involving aggregates.

Our final contribution is to analyze the behavior of the Magic Sets transformation, which is used for focusing the bottom-up evaluation to generate only facts relevant to a given query (Section 6). We show that under simple restrictions on the sideways information passing (sip) strategies, the Magic Sets transformed programs can be evaluated correctly and efficiently for left-to-right monotonic programs, a large sub-class of monotonic programs. The correctness proof of the Magic Sets transformation depends crucially upon our semantic reformulation of the monotonic semantics.

stratification to ensure that no fact depended on itself through aggregation [10, 15]. However, there are many useful queries that cannot be easily (if at all) expressed using stratified programs, and more recent semantics such as [7, 4, 5, 18, 16, **?**, **?**] have relaxed or removed stratification requirements.

In particular, the monotonic semantics of Ross and Sagiv [16] 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 [7], 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.)

While there has been considerable work on efficiently evaluating queries on programs with stratified aggregation, (see [6, 2, 17, 20, 19, 12], for example) not much attention has been paid to the problem of efficiently evaluating queries on programs with unstratified aggregation. The monotonic semantics covers a wider class of programs than semantics proposed earlier for unstratified aggregation [7, 4] while having much more potential for efficient evaluation than techniques proposed for more general classes of unstratified aggregation [18, **?**, **?**]. In spite of the potential, no efficient technique for evaluating monotonic programs was known.

In this paper, we present the first results (to our knowledge) on the problem of efficient evaluation of queries on programs with monotonic aggregation [16]. Thus our paper helps realize the potential for efficient evaluation inherent in the monotonic semantics. Specifically, our main contributions are as follows:

1. Results on incremental (re)computation of aggregate functions applied to a multiset of values, when that multiset is updated (Section 2).

2. A semantic reformulation of the monotonic semantics of [16] in terms of computations, following the approach of Beeri et al. [**?**] (Section 4).

3. Results on incremental evaluation of a broad class of programs with aggregation, including SQL programs and monotonic programs with non-stratified aggregation (Sections 3 and 5).

4. Results on the behavior of the Magic Sets transformation (see [**?**, 14], for example) of monotonic programs (Section 6).

We now describe each of these contributions in more detail.

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 describe novel incremental evaluation techniques

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 ????
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 consider the problem of efficiently evaluating queries on programs with monotonic aggregation, a class of programs defined by Ross and Sagiv. Our approach for solving this problem consists of the following techniques: incremental computation of aggregate functions, incremental fixpoint evaluation of monotonic programs and query-directed transformation of monotonic programs. Our techniques for the incremental computation of aggregate functions are of independent interest for incremental maintenance of materialized SQL views and triggers defined using aggregate functions.

We formalize the notion of incremental computation of aggregate functions on a multiset, develop novel incremental evaluation techniques for several aggregate functions, and provide upper and lower bounds for incremental computation of a variety of aggregate functions. We present an incremental fixpoint evaluation procedure for monotonic programs which is based on a combination of incremental aggregation and the Semi-Naive evaluation technique. We analyze the behavior of the Magic Sets transformation on monotonic programs, and describe simple restrictions on programs and sideways information passing (sip) strategies under which this transformation preserves monotonicity.

We present a proof-theoretic reformulation of the monotonic semantics in terms of computations, following the approach of Beeri et al., which greatly simplifies the task of proving the correctness of our optimizations; the reformulation is also of interest in itself.

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