Optimizing Recursive Queries
with Monotonic Aggregates in DeALS

Alexander Shkapsky, Mohan Yang, Carlo Zaniolo

Computer Science Department, UCLA
{shkapsky, yang, zaniolo}@cs.ucla.edu
Technical Report #140016

Abstract—The exploding demand for analytics has refocused the attention of data scientists on applications requiring aggregation in recursion. After resisting the efforts of researchers for more than twenty years, this problem is being addressed by innovative systems that are raising logic-oriented data languages to the levels of generality and performance that are needed to support efficiently a broad range of applications. Foremost among these new systems, the Deductive Application Language System (DeALS) achieves superior generality and performance via new constructs and optimization techniques for monotonic aggregates which are described in the paper. The use of a special class of monotonic aggregates in recursion was made possible by recent theoretical results that proved that they preserve the rigorous least-fixpoint semantics of core Datalog programs. This paper thus describes how DeALS extends their definitions and modifies their syntax to enable a concise expression of applications that, without them, could not be expressed in performance-conducive ways, or could not be expressed at all. Then the paper turns to the performance issue, and introduces novel implementation and optimization techniques that outperform traditional approaches, including Semi-naive evaluation. An extensive experimental evaluation was executed comparing DeALS with other systems on large datasets. The results suggest that, unlike other systems, DeALS indeed combines superior generality with superior performance.

I. INTRODUCTION

The fast-growing demand for analytics has placed renewed focus on improving the support for aggregation in recursion. Aggregates in recursive programs are essential in many important applications, including programs in new areas such as computer networking [1], social networks [2], and data mining. Significant applications include machine learning-algorithms for Markov chains and hidden Markov models, and algorithms such as Apriori that require iterating over counts, or probability-computations. Besides these new applications, we can mention a long list of traditional ones, such as Bill of Materials (BOM), a.k.a. subparts explosion: this classical recursive query for DBMS requires aggregating the various parts in the part-subpart DAG. Finally, we have problems, such as computing the shortest paths or counting the number of paths between vertices in a graph, which are now covered as foundations by most CS101 textbooks.

Although aggregates were not covered in E.F. Codd's definition of the relational calculi [3], it did not take long before early versions of relational languages such as SQL included support for aggregate functions, namely count, sum, avg, min and max, along with associated constructs such as group-by. However, a general extension of recursive query theory and implementation to allow for aggregates proved an elusive goal, and even recent versions of SQL that provide strong support for OLAP and other advanced aggregates disallow the use of aggregates in recursion and only support queries that are stratified w.r.t. to aggregates.

Yet the desirability of extending aggregates to recursive queries was widely recognized early and many partial solutions were proposed over the years for Datalog languages [4], [5], [6], [7], [8], [9], [10]. The fact that, in general, aggregates are non-monotonic w.r.t. set-containment led to proposals based on non-monotonic theories, such as locally stratified programs and perfect models [11], [12], well-founded models [13] and stable models [14]. An alternative approach was due to Ross and Sagiv [9], who observed that particular aggregates, such as continuous count, are monotonic over lattices other than set-containment and thus can be used in non-stratified programs. However practical difficulties with this approach were soon pointed out by [15] who showed that determining the correct lattices for programmers and compilers would be quite difficult, and that prevented the deployment of the monotonicity idea in practical query languages for a long time. Fortunately, we recently witnessed some dramatic developments change the situation completely. The first is that Hellerstein et al., after announcing a resurgence of Datalog, showed that monotonicity in special lattices can be very useful in proving the formal properties such as eventual consistency [16]. Then, we have seen monotonic aggregates making a strong come back in practical query languages thanks to the results published in [17], [18] and in [2], summarized below.

The formalization of monotonic aggregates proposed in [17], [18] preserves monotonicity w.r.t. set containment, and it is thus conducive to simplicity and performance that follow respectively from the facts that (i) users no longer have to deal with lattices, and (ii) the query optimization techniques, such as Semi-naive and magic sets remain applicable [17]. SocialLite [2] also made an important contribution by showing that shortest-path queries, and other algorithms using aggregates in recursion, can be implemented very efficiently so that in many situations the Datalog approach becomes preferable to that of hand-coding big-data analytics in some procedural language.

These dramatic advances represented a major source of
opportunities and challenges for our Deductive Application Language (DeAL) and its system DeALS. In fact, unlike the design of the SociaLite system where the performance of recursive graph algorithms with aggregates had played a role, DeALS has been designed as a very general system seeking to satisfy the many needs and lessons that had been learned in the course of a long experience with logic-based data languages, and the LDL [19] and LDL++ [20] experiences in particular. Thus, DeALS supports key non-monotonic constructs having formal stable model semantics, including, e.g., X-stratification and the choice construct that were found quite useful program analysis [21], and user-defined aggregates that enabled important knowledge-discovery applications [22].

In addition to a rich set of constructs, DeALS was also designed to support a roster of optimization techniques including magic sets and supplementary magic sets, non-linear recursion, and existential quantification. Introducing powerful new constructs and their optimization techniques by retrofitting a system that already supports a rich set of constructs and optimizations represented a difficult technical challenge. In this paper, we describe how this challenge was met with the introduction of new optimization techniques for monotonic aggregates. We will show that DeALS now achieves both performance and generality, and we will underscore this point by comparing not only with SociaLite but also with systems such as DLV [23] and LogicBlox [24] that realize different performance/generality tradeoffs.

Overview Preliminaries are reviewed in the next section. We then begin the first of two main parts of the paper. Section III presents the syntax and semantics for our min (mmin) and max (mmax) monotonic aggregates. Section IV discusses the evaluation and optimization of monotonic aggregate programs. Section V presents implementation details for mmin and mmax and our B+Tree based storage manager followed by its experimental validation in Section VI. The second part of the paper begins with Section VII discussing the novel optimization techniques developed for the count (mcount) and sum (msum) monotonic aggregates, followed by their implementation in Section VIII and experimental validation in Section IX. Section X provides additional DeAL program examples. Section XI presents the formal semantics on which our aggregates are based. Related work is reviewed in Section XII and we conclude in Section XIII.

II. PRELIMINARIES

Datalog Programs A Datalog program P is a finite set of rules, or Horn Clauses, where rule r in P has the form A ← A1,..., An. The atom A is the head of r. A1,..., An, the body of r, are literals, or goals, where each literal can be either a positive or negated atom. An atom has the form p(t1,...,tn) where p is a predicate and t1,...,tn are terms which can be constants, variables or functions. An r with an empty body is a fact. A successful assignment of all variables of rule body goals results in a successful derivation for the rule head predicate. Datalog programs use set semantics and are (typically) stratified (i.e. partitioned into levels based on rule dependencies) and executed in level-by-level order, in a bottom-up fashion. Datalog programs can be evaluated using an iterative approach such as Semi-naive evaluation [10].

III. MMIN AND MMAX MONOTONIC AGGREGATES

An mmin or mmax monotonic aggregate rule has the form:

\[ p(K_1,\ldots,K_n,\text{agg}(T)) \leftarrow \text{Rule Body.} \]

K1,...,Kn are the zero or more group-by arguments we also refer to as K. agg ∈ \{mmax, mmin\} is the monotonic aggregate. T is the aggregate term which is a variable.

mmin and mmax are aggregate functions that map an input set or multiset, we will call G, to an output set, we will call D. Then, given G, for each element g ∈ G mmin will put g into output set D if g is less than the least value mmin has previously computed for G. Similarly, given an input set G, for each element g ∈ G mmax will put g in output set D if g is greater than the greatest value mmax has previously computed for G. When viewed as a sequence, the values produced by mmin and mmax is monotonic. The mmin and mmax aggregates are monotonic w.r.t. set-containment and can be used in recursive rules, therefore G can be viewed as a union of rule bodies across fixpoint iterations. These aggregates memorize the most recently computed value and thus require a single pass\(^1\) over G.

A. Running Example

The All-Pairs Shortest Paths (APSP) program has received much attention in the literature [6], [9], [13], [25], [26]. APSP calculates the shortest distance path between each pair of connected vertices in a directed graph with edge costs.

\textbf{Example 1:} APSP with mmin

\begin{align*}
rl. \text{spaths}(X,Y,m\text{in}(D)) & \leftarrow \text{edge}(X,Y,D), \\
rs. \text{spaths}(X,Y,m\text{in}(D)) & \leftarrow \text{spaths}(X,Z,D1), \text{edge}(Z,Y,D2), \text{D} = D1 + D2. \\
rr. \text{shortestpaths}(X,Y,m\text{in}(D)) & \leftarrow \text{spaths}(X,Y,D).
\end{align*}

Example 1 is the DeAL APSP program with the mmin aggregate. The edge predicate denotes the edges of the graph.

The intuition for this program is as follows. In the recursion (rl, r2), an spaths fact will be derived if a path from X to Y is either i) new or ii) has length shorter than the currently known length from X to Y. r1 finds the shortest path for each edge. r2 is the left-linear recursive rule that computes new shortest paths for spaths by extending previously derived paths in spaths with an edge. Logically, this approach can result in many facts spaths for X,Y, each with a different length. Therefore, the program is stratified using a traditional (non-monotonic) min aggregate (rr) to select the shortest path for each X,Y.

\textbf{APSP By Example} Next, we walk through a Semi-naive evaluation of the Shortest Paths program from Example 1.

First, rl in Example 1, the exit rule, is evaluated on the edge facts in Figure 1. In the rule head in rl, X and Y, the

\(^1\)SQL 2003 max, min, count and sum aggregates on the unlimited preceeding window are similar to DeAL’s monotonic aggregates.
non-aggregate arguments, are the group-by arguments. The \( \text{mmin} \) aggregate is applied to each of the six edge facts and six spaths facts are successfully derived (not displayed for reasons of space) because no aggregate values had been previously computed (memorized) and each group (i.e. \((a, b)\)) was represented among the facts only once. For the spaths predicate, \( \text{mmin} \) is now initialized with a value for each group.

\[
\text{spaths}(a, c, 2) \leftarrow \text{spaths}(a, b, 1), \text{edge}(b, c, 1), 2 = 1 + 1.
\]

\[
\text{FAIL} \leftarrow \text{spaths}(a, b, 1), \text{edge}(b, d, 4), 5 = 1 + 4. \quad \text{[i]}
\]

\[
\text{FAIL} \leftarrow \text{spaths}(a, c, 3), \text{edge}(c, d, 1), 4 = 3 + 1. \quad \text{[ii]}
\]

\[
\text{spaths}(b, d, 2) \leftarrow \text{spaths}(b, c, 1), \text{edge}(c, d, 1), 2 = 1 + 1.
\]

\[
\text{spaths}(a, d, 3) \leftarrow \text{spaths}(a, c, 2), \text{edge}(c, d, 1), 3 = 2 + 1.
\]

\[
\text{shortestpaths}(a, c, 2) \leftarrow \{\text{spaths}(a, c, 3), \text{spaths}(a, c, 2)\}
\]

\[
\text{shortestpaths}(a, d, 3) \leftarrow \{\text{spaths}(a, d, 4), \text{spaths}(a, d, 3)\}
\]

\[
\text{shortestpaths}(b, d, 2) \leftarrow \{\text{spaths}(b, d, 4), \text{spaths}(b, d, 2)\}
\]

Consequence Operator (ICO) for the exit rule(s) and the recursive rule(s), respectively. The algorithm evaluates as follows. Firstly, \textbf{Semi-naive} applies \( T_E \) (i.e. the exit rules) on \( M \) to derive the first set of new \( \delta \) facts \( \delta S \) (line 2). Then, until no new facts are derived during an iteration, \textbf{Semi-naive} evaluates \( T_R \) on \( \delta S \) to derive new facts to be used in the next iteration. The new set of \( \delta \) facts \( \delta S' \) is produced only after the removal of facts found in previous steps (line 5).

Symbolic differentiation rules [10] can be applied to monotonic aggregate rules in a straightforward manner to produce rules for \textbf{Semi-naive}. We omit this in the interest of space.

Although \textbf{Semi-naive} is an efficient evaluation technique for general Datalog programs, the \textit{max-based optimization} [18] identified counting only needs to be performed on maximum (max) values if only monotonic arithmetic and boolean functions are used. In this work, we expand this observation which we refer to as the \textbf{Monotonic Optimization}. The intuition behind the \textbf{Monotonic Optimization} is that with our monotonic aggregates, monotonicity is preserved and values other than the max (\( \text{mmax} \)) or min (\( \text{mmin} \)) will add no new results and thus can be ignored. Only the max (min) intermediate values need to be used in derivations to produce the final max (min) value. In fact, the last fact produced by the aggregate for a group contains the greatest (\( \text{mmax} \)) or least (\( \text{mmin} \)) aggregate value, making this fact the only fact for the group that we need to produce for the next iteration.

\section{Monotonic Aggregate Evaluation}

In this section, we present optimized evaluation techniques for programs with monotonic aggregates. We start with a review of \textbf{Semi-naive} fixpoint evaluation, the technique that serves as the basis for our optimized evaluation approaches.

In Figure 5, the algorithm for \textbf{Semi-naive}, \( M \) is the initial model (database), \( S \) contains all facts obtained thus far, \( \delta S \) and \( \delta S' \) contain facts obtained during the previous and current iteration, respectively, and \( T_E \) and \( T_R \) are the Immediate

\[
1: S := M;
2: \delta S := T_E(M);
3: S := S \cup \delta S;
4: \textbf{while} \ \delta S \neq \emptyset \ \textbf{do}
5: \delta S' := T_R(\delta S) - S;
6: S := S \cup \delta S';
7: \delta S := \delta S';
8: \textbf{return} S;
\]

The \textbf{Monotonic Optimization} enables an optimized \textbf{Semi-naive} for monotonic aggregates we call \textbf{Monotonic Aggregate Semi-naive Evaluation (MASN)}. Figure 6 is the algorithm for MASN, which closely resembles \textbf{Semi-naive}. MASN’s differences with \textbf{Semi-naive} are as follows. Here we use \( \text{getLast}() \)

\[
\text{getLast}() \text{ supports MASN by maintaining a single fact per group in } \delta S'.
\]
to produce, from the input set, a set containing i) all facts from predicates that do not have monotonic aggregates, and ii) the last derived fact for each group from monotonic aggregate predicates. Now after the $T_E$ or $T_R$ produces a set of facts, getLast will be applied to produce the actual new $\delta^S$. Otherwise, MASN evaluation is the same as Semi-naive.

B. Eager Monotonic Aggregate Semi-naive Evaluation

MASN employs a level-by-level iteration boundary of a breadth-first search (BFS) algorithm. $\delta$ facts computed during the current iteration will be held for use until the next iteration. However, facts produced from monotonic aggregate rules can be used immediately upon derivation. Looking at the derivations in the walk-through evaluation of APSP in Section III-A one can see a case where Semi-naive, and in this case MASN as it would have evaluated the same as Semi-naive, did not capitalize on this property of monotonic aggregates.

**Example 1 r2 evaluation with Semi-naive or MASN**

\[
\text{spaths}(a, c, 2) \leftarrow \text{spaths}(a, b, 1), \text{edge}(b, c, 1), 2=1+1. \\
\text{FAIL} \leftarrow \text{spaths}(a, c, 3), \text{edge}(c, d, 1), 4=3+1.
\]

Fig. 7. Example of Iteration Boundary of MASN

Figure 7 shows the derivations of interest extracted from Figure 2. We see the second derivation performed using \text{spaths}(a, c, 3) (from $\delta S$) and result in failure because the value for (a, d) was 4. However, at the time the derivation is attempted, \text{spaths}(a, c, 2), the result of the immediately previous derivation, existed. Had \text{spaths}(a, c, 2) been used, \text{spaths}(a, d, 3) would have been derived here, rather than requiring another Semi-naive iteration (Figure 3).

To capitalize on the Monotonic Optimization, we present Eager Monotonic Aggregate Semi-naive Evaluation (EMSN). With EMSN, facts produced from monotonic aggregate rules are available to be used immediately upon derivation. EMSN evaluates recursive rules with monotonic aggregates in a fact-oriented (fact-at-a-time) manner, but still uses a set to determine which groups are used in derivations each iteration. Derivations with monotonic aggregates are always performed with the current aggregate value for a group.

Figure 8 is Eager Monotonic Aggregate Semi-naive Evaluation (EMSN). The main idea with EMSN is that recursive rules with monotonic aggregates are evaluated fact-at-a-time while the other rules are evaluated set-at-a-time like in Semi-naive. Monotonic aggregate rules are partitioned from the other rules and two sets of Immediate Consequence Operators (ICO) will be used. $T_{E_A}$ and $T_{R_A}$ are the ICO for the monotonic-aggregate exit and recursive rules, respectively, and $T_{R_A}$ will be applied on one fact at a time. $T_{E_N}$ and $T_{R_N}$ are the ICO for the remaining exit and recursive rules, respectively. $\delta S_A$ and $\delta S'_A$ contain facts obtained during the previous and current iteration, respectively, for the monotonic-aggregate rules, while $\delta S$ and $\delta S'$ contain facts obtained during the previous and current iteration, respectively, for the remaining rules. $M$ is the initial model, $S$ contains all facts obtained thus far. fact and newfact are facts from $\delta S_A$ and from a single application of $T_{R_A}$, respectively.

**Key points of Figure 8 are as follows.** We use getGroupRep() on the set produced by $T_{E_A}(M)$ to produce the initial $\delta S_A$ (line 2), getGroupRep(), produces a set containing only one aggregate predicate fact per group. For example, given \text{spaths}(a, b, 2) and \text{spaths}(a, b, 1), getGroupRep() would produce a set with either one representing (a, b) for \text{spaths}. Then, $T_{E_N}$ is applied to $M$ to produce the initial $\delta S$ (line 3). Once in the recursion, individually, each fact in $\delta S_A$ is used to retrieve its group’s current fact getCurrent(fact) from the aggregate relation, which $T_{R_A}$ is then applied to (line 7). Upon successful derivation by $T_{R_A}$, newfact is added to $\delta S'_A$ (line 8). After all facts in $\delta S_A$ are used, getGroupRep is applied to $\delta S'_A$ to get the $\delta$ facts for aggregates for the next iteration. Then, $T_{R_N}$ (remaining rules) is applied to $\delta S$ which, after duplicates are eliminated, produces $\delta S''$, the set of facts to be used in the next iteration (line 10). This process repeats until no new facts are produced during an iteration.

**Example 1 r2 evaluation with EMSN**

\[
\text{spaths}(a, c, 2) \leftarrow \text{spaths}(a, b, 1), \text{edge}(b, c, 1), 2=1+1. \\
\text{spaths}(a, c, 3) \leftarrow \text{spaths}(a, c, 2), \text{edge}(c, d, 1), 4=3+1.
\]

Fig. 9. EMSN Fact-at-a-time Efficiency

Now, consider the same scenario from Figure 7, but this time using EMSN from Figure 8. In Figure 9, now after \text{spaths}(a, c, 2) is produced, it is immediately used in the next derivation. This results in a successful derivation, and one iteration earlier than with Semi-naive or MASN. Moreover, with \text{spaths}(a, c, 2) now derived, we can ignore \text{spaths}(a, c, 3) as it will not lead to a final result (Monotonic Optimization).

**Discussion** Since EMSN evaluates the ICO for recursive monotonic aggregate rules ($T_{R_A}$) on an individual fact, rather than on a set of facts, it can use facts immediately upon derivation. Although EMSN is based on Semi-naive, and therefore BFS, EMSN has depth-first search (DFS) characteristics. Like BFS, EMSN still uses a level-at-a-time (iteration) approach guided by facts in $\delta S$ and $\delta S_A$ that were derived during the previous iteration. However, because EMSN uses the most recent aggregate value for the group the fact belongs to, regardless of when the value was computed, EMSN can evaluate deeper than a single level of the search space during an iteration of evaluation. The result is higher (min) or lower (max) aggregate values being derived earlier in evaluation.
which in turn prunes the search space to avoid derivation of facts that will not result in final values.

V. IMPLEMENTATION

In this section, we present details of DeAL’s mmin and mmax aggregate implementation.

A. System Overview

DeALS is an interpreted Datalog system with three main components — the compiler, the interpreter and the storage manager. Monotonic aggregate rules are supported by the compiler with an aggregate rewriting approach based on techniques from [27]. The interpreter uses tuple-at-a-time pipelining, evaluating rule bodies in a left-to-right fashion with backtracking. From the binding pass analysis, DeALS determines index and cursor selection. For instance, from Example 1, in r1, both X and Y are free, so edge will be scanned, while in r2, spaths will be scanned and edge will be indexed on Z (first argument) because Z is bound by spaths.

B. Storage Manager Overview

DeAL’s storage manager provides support for main memory storage and indexing for predicate relations. DeALS supports several B+Tree data structure variants for tuple storage. B+Trees stores fixed-size keys in internal and leaf nodes and fixed-size non-key attributes in leaf nodes. Leaf nodes have pointers to their right neighbors for fast linear scan of the tree. Through testing we determined our implementations perform pointers to their right neighbors for fast linear scan of the tree.

B+Tree Aggregators

Early experiments found aggregation with UHT lacking in execution time performance. The B+Tree Aggregator TupleStore (BAT) is a B+Tree design optimized for pipelined aggregation in recursive queries that provides both good read and write performance. BAT store fixed-size keys in internal and leaf nodes and fixed-size aggregate values in leaf nodes. Keys are unique and only one aggregate value per key is maintained. Leaf nodes have pointers to their right neighbors and linear key search is used in both internal and leaf nodes. In a BAT, aggregation is performed in the leaves, therefore only one search is needed to retrieve the previous value, compare it with the new value and perform the update.

Unlike with UHT, facts in our B+Trees are not easily tracked by reference or range because of node splitting. In recursive queries, after a modification is made in a leaf, BAT will insert the fact’s key into a second B+Tree\(^3\), which maintains the set of keys to process for the next iteration (\(S_{A}^{\delta}\) in Figure 8). EMSN will scan this second B+Tree using a cursor, and for each key, retrieve the fact from the BAT, which contains the current aggregate value for the key.

C. mmax and mmin Implementation

The mmax and mmin implementation tracks the greatest (mmax) or least (mmin) value computed for each group where each group has one tuple in the TupleStore. We use a single relation schema with one column for each of the predicate’s group-by argument and a column for the aggregate value. Specifically, BAT keys are the group-by arguments with the aggregate value stored in the leaf. UHT are indexed (B+Tree) on the group-by arguments. For instance, spaths in Example 1 would use BAT with keys (X, Y) and each X, Y would be stored with its current value (D) in a leaf node.

D. Operational Optimizations

No Recursive Relation Storage

Due to the Monotonic Optimization, we only need to maintain a single fact per group and when a new value for the group is successfully derived, we overwrite the previous value. If the recursive predicate and monotonic aggregate use separate stores, with EMSN and pipelining, the result is the recursive relation storage is merely being synchronized with the aggregate relation storage. Therefore, we do not allocate the recursive predicate storage, and instead have it read from the monotonic aggregate storage.

Final Results via Monotonic Aggregate

Since the monotonic aggregate maintains the value for each group in its TupleStore, when a fixpoint is reached, its TupleStore contains the final results. For instance, instead of evaluating r3 in Example 1 the recursion is materialized by the system, as it would have been by the stratified aggregate, and DeALS retrieve the final values from the monotonic aggregate’s TupleStore.

VI. MMIN & MMAX PERFORMANCE ANALYSIS

All experiments on synthetic graphs were run on a machine running Ubuntu 14.04 LTS 64-bit with an i7-4770 CPU and 32GB memory. The experiments on real-life graphs were run on a machine running Ubuntu 12.04 LTS 64-bit with four AMD Opteron 6376 CPUs and 256GB memory. The memory utilization is collected by the Linux `time` command. Execution time and memory utilization are calculated by performing the same experiment five times, discarding the highest and lowest values, and taking the average of the remaining three values. All the experiments on systems written in Java were run using Java 1.8.0 except for SociaLite (0.8.1) which doesn’t support Java 1.8.0. The experiments for SociaLite were run using Java 1.7.0.

Datasets

An n-vertex graph used in experiments has integer vertex labels ranging from 0 to n − 1. We used three kinds of synthetic graphs — 1) directed acyclic graphs (DAGs), generated by connecting each pair of vertices i and j (i < j) with (edge) probability p; 2) random graphs, generated by connecting each pair of vertices with (edge) probability p; 3) scale-free graphs, generated using GTgraph\(^4\). The graphs are shuffled after generation where one random permutation is

\(^3\)We use a B+Tree because the keys can be scanned in order, which can benefit EMSN.

\(^4\)GTgraph, http://www.cse.psu.edu/~madduri/software/GTgraph/.
applied to the vertex labels and another random permutation is applied to the edges. The real-life graphs are not shuffled. We only relabeled the graphs whose vertex labels are beyond the range of \([0, n – 1]\) while maintaining the original edge order. A text description such as “10K/20K” indicates the graph has 10,000 vertices and 20,000 edges.

**Configuration** BAT and B+Tree indexes for UHT were configured with 256 bytes allocated for keys in each node (internal and leaf). Other than experiments in Section VI-C, DeALS used EMSN with BAT.

### A. Datalog Language Implementation Comparison

DeALS is a sequential, main memory Java implementation. We compare its execution time and memory utilization on the shortest path query against the other three Datalog language implementations supporting aggregates in recursion — 1) the DLV system\(^5\), which is the state-of-the-art implementation of disjunctive logic programming; 2) the commercial LogicBlox system, which recently included support for aggregates in recursion using *staged partial fixpoint* semantics\(^6\). Based on an analysis of log files during query execution, we determined LogicBlox indeed uses an approach akin to *Semi-naive* and only uses new shortest paths found in the current iteration in derivations during the next iteration; 3) the SociaLite\(^7\) graph query language [2] which supports a left-linear recursive expression of the shortest path query and evaluates it using *Semi-naive*. SociaLite efficiently evaluates single-source shortest paths (SSSP) and queries of a similar form using an approach with Dijkstra’s algorithm-like efficiency [2]. We compared with SociaLite version 0.8.1 because it had the best execution time performance on sequential queries for SociaLite versions available to us.

\(^5\)DLV with recursive aggregates support, http://www.dbai.tuwien.ac.at/proj/dlv/dlvRecAggr/.


\(^7\)https://sites.google.com/site/socialitelang/

Among the 18 graphs, SociaLite is the fastest on two graphs, and DeALS is the fastest on the remaining 16 graphs. DeALS is more than two times faster than SociaLite on sparse graphs that the average degree of each vertex is only two (e.g., G7, G10 and G16). This advantage reduces as the average degree increases from two to ten. The main reason for this change is due to the different designs between DeALS and SociaLite. SociaLite uses an array of hash tables with an initial capacity of around 1,000 entries to maintain the delta relations, whereas DeALS uses a B+Tree. The initialization cost of a hash table is higher than that of a B+Tree, while the cost of accessing a hash table is lower than that of a B+Tree. For graphs with small average degree, the initialization cost of accessing a hash table is lower than that of a B+Tree.

For graphs with small average degree, the initialization cost may account for a large percentage of the execution time, thus DeALS is faster than SociaLite. The impact of the initialization cost reduces as the average degree increases, and thus SociaLite is faster than DeALS on denser graphs. But this speed comes at the cost of higher memory utilization. SociaLite uses more than two times memory than DeALS on all 18 graphs. Although the C-based system DLV has a significantly lower memory utilization than Java-base systems DeALS and SociaLite, it is extremely slow comparing with both DeALS and SociaLite on DAGs. To sum up, DeALS achieves the best speed and memory trade-off on sparse graphs among the three compared systems.
Real-life graphs. Table I shows the results of the APSP query on three real-life graphs from the Stanford large network dataset collection. These graphs don’t have edge costs. We assigned unit cost to each edge. The results are similar to that of synthetic graphs — DeALS is the fastest while DLV has the lowest memory utilization. These results suggest that, the B+Tree-based design (low initialization cost) adopted by DeALS is more favorable than the hash table-based design on real-life workloads.

<table>
<thead>
<tr>
<th>Evaluation Type</th>
<th>Storage Configuration</th>
<th>Time (s)</th>
<th>Memory (GB)</th>
</tr>
</thead>
<tbody>
<tr>
<td>EMSN</td>
<td>UHT with B+Tree index</td>
<td>77.743</td>
<td>4.087</td>
</tr>
<tr>
<td>EMSN</td>
<td>UHT with B+Tree index</td>
<td>68.927</td>
<td>4.131</td>
</tr>
<tr>
<td>EMSN</td>
<td>BAT</td>
<td>26.450</td>
<td>1.048</td>
</tr>
</tbody>
</table>

VIII. MONOTONIC COUNT AND SUM

DeALS also efficiently supports monotonic count and sum aggregates allowing DeAL to support many exciting applications.

A. mcount & msum Monotonic Aggregates

An mcount or msum monotonic aggregate rule has the form:

\[ p(K_1, \ldots, K_n, \text{aggr}((T, P_T))) \leftarrow \text{Rule Body} \]

\( K_1, \ldots, K_n \) are the zero or more group-by arguments \((\mathcal{K})\).

aggr \in \{mcount, msum\} is the monotonic aggregate. \((T, P_T)\)

11As in [28], \( \bar{m} \) is accepted when \( \epsilon < (0.05 \times \bar{m}) \), where \( \epsilon = (1.96 \times \sigma)/\sqrt{k} \). \( \sigma \) is standard deviation. \( k \) is the number of graphs used. 1.96, from the tables for standard normal distribution for 0.975, gives the 95% confidence coefficient.
is the aggregate term pair where $T$ is a variable passed in from the body and $P_T$ is a constant or a variable passed from the body indicating the partial count/sum contributed by $T$.

As with $\text{mmin}$ and $\text{mmax}$, the $\text{mcount}$ and $\text{msum}$ aggregates are monotonic w.r.t. set-containment and can be used in recursive rules, and memorize the most recently computed value. When viewed as a sequence, the values produced by $\text{mcount}$ and $\text{msum}$ are aggregate functions that map an input set or multiset, we will call $G$, to an output set, we will call $D$. Elements $g \in G$ have the form $(J, N_J)$, where $N_J$ indicates the partial count/sum contributed by a $J$. Note, $J$ maps to $T$ and $N_J$ maps to $P_T$ in the definition of the aggregate term.

Now, given $G$, for each element $g \in G$, if $N_J > N_{J_{prev}}$, where $N_{J_{prev}}$ is the previous count (sum) for $J$ or 0 if no previous count (sum) for $J$ exists, $\text{mcount}$ ($\text{msum}$) computes the count (sum) for $G$ by summing all maximum partial counts (sums) $N_J$ for all $J$. Since only the maximum $N_J$ for each $J$ is aggregated, no double counting (summing) occurs. Lastly, $\text{msum}$ will only compute with positive numbers, thereby ensuring its monotonicity.

**Running Example** Example 2 is the DeAL program to count the paths between pairs of vertices in an acyclic graph. This program is not expressible with Datalog with stratified aggregation [8]. We will use Example 2 as our running example for $\text{mcount}$ to explain counting in DeAL.

**Example 2:** Counting Paths in a DAG

```
r1. cpaths(X, Y, \text{mcount}((X, 1))) ← edge(X, Y).
r2. cpaths(X, Y, \text{mcount}((Z, C))) ← cpaths(X, Z, C), edge(Z, Y).
r3. cpaths(X, Y, \text{mcount}((Z, C))) ← cpaths(X, Z, C), edge(Z, Y).
```

In Example 2, $r1$ counts each edge as one path between its vertices. In $r2$, any $\text{edge}(Z, Y)$ that extends from a computed path $\text{cpath}(X, Z, C)$ establishes there are $C$ distinct paths from $X$ to $Y$ through $Z$. The $\text{mcount}((Z, C))$ aggregate in the head sums the count of paths from $X$ to $Y$ through every $Z$ to produce the count from $X$ to $Y$. Lastly, $r3$ indicates only the maximum count for each path $X, Y$ in $\text{cpaths}$ is desired. As explained in Section V-D, $r3$ does not have to be evaluated.

As an example of the interval semantics - if $r2$ in Example 2 produced $\text{cpaths}(a, b, 3)$ and then $\text{cpaths}(a, b, 4)$, we cannot sum the aggregate values to get a new count for $(a, b)$. Instead, with the counts for $\text{cpaths}(a, b, 3)$ and $\text{cpaths}(a, b, 4)$ represented with $[1, 3]$ and $[1, 4]$, respectively, $[1, 3] \cup [1, 4] = \text{max}(3, 4) = 4$. Thus $\text{cpaths}(a, b, 4)$ represents $(a, b)$’s count.

**B. mcount by Example**

To further explain the $\text{mcount}$ aggregate, we walk through an evaluation of Counting Paths (Example 2) using EMSN. The edge facts in Figure 12 is the example dataset. First, $r1$ in Example 2 is evaluated and results in the six $\text{cpaths}$ facts as shown in the Figure 13. Each $\text{cpaths}$ fact has a count of 1 indicating one path between each pair of vertices connected by edge facts. The partial count is memorized (recall $(J, N_J)$ discussed above) for each group (e.g. $(a, b)$), also displayed in the right column of Figure 13. For example, in the first derivation, $J=a$, $N_J=1$, $(a, 1)$ is memorized for $(a, b)$.

**EMSN** evaluates the recursive $r2$ rule from Example 2 using the $\text{cpaths}$ derived by $r1$. Figure 14 shows the successful derivations performed by $r2$. Four $\text{cpaths}$ facts are derived bringing the total number of $\text{cpaths}$ facts to ten. Note the derivation of $\text{cpaths}(a, d, 2)$ from joining $\text{cpaths}(a, b, 1)$ and $\text{edge}(b, d)$. It represents a count of two for $(a, d)$, even though the rule body contributed only one path count. However, looking at the *-ed entry in Figure 13, we see a partial count of $(a, 1)$ towards $(a, d)$ was accrued during evaluation of $r1$. Therefore, when computing the new count for $(a, d)$, $(a, 1)$ and the newly derived $(b, 1)$ are summed to result in $\text{cpaths}(a, d, 2)$. Next, we observe the benefits of using EMSN with the derivation of $\text{cpaths}(a, d, 4)$. Since $\text{cpaths}(a, c, 2)$ existed even though it was derived this iteration, it was used and successfully joined with $\text{edge}(c, d)$. Then the partial counts for $(a, d)$, which are $(a, 1)$, $(b, 1)$, and $(c, 2)$, are summed to produce $\text{cpaths}(a, d, 4)$. With $r2$ exhausted, a fixpoint is reached and we have our result.

**VIII. COUNTING IMPLEMENTATION**

In this section, we present details of DeAL’s monotonic count and sum aggregate implementation. We use definitions from Section VII (e.g. $G$). Note, $G$ is a single group produced from the implicit group-by for a distinct assignment of $X$, the zero or more group-by arguments.

**A. mcount and msum Implementation**

Although we use $\text{mcount}$ to present our efficient count/sum technique, it is easily generalizable to $\text{msum}$.

We present an efficient approach for count and sum using delta-maintenance ($\Delta$-Maintenance) techniques. Recalling our explanation for $\text{mcount}$ in Section VII, given a partial count $N_J > N_{J_{prev}}$, $\text{mcount}$ will sum all maximum partial counts to compute the new total count for group $G$. However, rather than recompute the count, we can instead use $\Delta$-Maintenance to increase $N$ (the current total count for $G$) by $N_J - N_{J_{prev}}$ and put the updated count, now the total current count for $G$, into output set $D$. This produces the same result as if the maximum partial count $N_J$ for all $J$ are summed
to produce the total count \( N \) for \( G \), however avoids the re-
summation of all \( N_J \) with each change in a \( N_J \). This requires
memorizing both \( N \) for \( G \) and \( N_J \) for all \( J \).

**Storage Design** Table III displays storage designs we
investigated for \( \text{mcount} \) and \( \text{msum} \). Here we use \( \mathbb{N} \) to indicate
the current count/sum for the group-by arguments \( \mathbb{R} \). As in
Section VII-A, each \( T \) contributes a partial count/sum \( P_T \)
for a distinct assignment of \( \mathbb{R} \).

**Double** uses two relations, one relation \((\mathbb{R}, \mathbb{N})\) indexed on \( \mathbb{R} \)
to store tuples containing the group’s total aggregate value and a
second relation \((\mathbb{R}, \mathbb{T}, \mathbb{N}_J)\) indexed on \( \mathbb{R}, \mathbb{T} \) to store the partial
count \( \mathbb{N}_J \) for each distinct assignment of \( \mathbb{R}, \mathbb{T} \). Early testing
showed the **Double** using **UHT** without \( \Delta\text{-Maintenance} \) to take
2-5 times longer to execute then when using \( \Delta\text{-Maintenance} \).

Next, we investigated designs using **KeyValue** type columns
as a more efficient way of managing \((\mathbb{T}, P_T)\) pairs. We de-
veloped three single relation designs \((\mathbb{R}, \mathbb{N}, \text{**KeyValue**}[(\mathbb{T}, P_T)])\),
where \( \mathbb{N} \) is the total count for \( \mathbb{R} \) and **KeyValue**\([(\mathbb{T}, P_T)]\) is a
reference to the tuple’s own **KeyValue** type data structure. The
relation is indexed on \( \mathbb{R} \) and each group has a single tuple. The
**KeyValue** types each represent a different retrieval time
complexity; a **List** \( O(\mathbb{N}) \) type, a **B+Tree** \( O(\log(\mathbb{N})) \) type,
and a **Hashtable**\(O(1)\) type. **Hashtable** is based on Linear
Hashing and stores the hashed key in the bucket to avoid
rehashing. **B+Tree** stores keys \( \mathbb{T} \) in internal and leaf nodes and
non-key attributes \( P_T \) in leaf nodes, and uses linear
search. Lastly, **List** stores \( \mathbb{T}, P_T \) pairs ordered by \( \mathbb{T} \) and uses
a linear search. These are main memory structures, so designs
attempt to limit the number of objects (e.g. **List** uses byte
arrays). **KeyValue** designs use \( \Delta\text{-Maintenance} \).

Recall our options for tuple storage from Section V-B. For
the designs shown in Table III, DeALS supports **List**, **Hashtable**
and **B+Tree** with BAT and all designs with **UHT**
indexed as shown. For example, using \( r1, r2 \) from Example
2, with **B+Tree**, the **BAT** would have \( X, Y \) as keys and the
each entry in a leaf would have the current total count \( \mathbb{N} \) and a
reference to a second **B+Tree** to store \( \mathbb{T}, P_T \) pairs. This design
is essentially a **B+Tree** of **B+Trees**.

### IX. COUNTING PERFORMANCE ANALYSIS

**Configuration** **B+Trees**, **BAT** and **B+Trees** in the **B+Tree**
design for \( \text{mcount} \) and \( \text{msum} \) were configured with 256
bytes allocated for keys in each node (internal and leaf). The
**Hashtable** design for \( \text{mcount} \) and \( \text{msum} \) used a directory and
segment size of 256, 16 initial buckets and split policy of 10.

<table>
<thead>
<tr>
<th>Name</th>
<th>Schema</th>
<th>Indexes</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Double</strong></td>
<td>((\mathbb{R}, \mathbb{N}), (\mathbb{R}, \mathbb{T}, P_T))</td>
<td>( \mathbb{R} ), ( \mathbb{R}, \mathbb{T} )</td>
</tr>
<tr>
<td><strong>List</strong></td>
<td>((\mathbb{R}, \mathbb{N}, \text{<strong>List</strong>}[(\mathbb{T}, P_T)]))</td>
<td>( \mathbb{R} )</td>
</tr>
<tr>
<td><strong>B+Tree</strong></td>
<td>((\mathbb{R}, \mathbb{N}, \text{<strong>B+Tree</strong>}[(\mathbb{T}, P_T)]))</td>
<td>( \mathbb{R} )</td>
</tr>
<tr>
<td><strong>Hashtable</strong></td>
<td>((\mathbb{R}, \mathbb{N}, \text{<strong>Hashtable</strong>}[(\mathbb{T}, P_T)]))</td>
<td>( \mathbb{R} )</td>
</tr>
</tbody>
</table>

A. Statistical Analysis of Evaluation Methods

We also performed the statistical analysis described in
Section VI-B comparing Counting Paths (Example 2) using
**EMSN** with a **Semi-naive** implementation of Counting Paths.
The experiment used randomly generated DAGs of 100-250
vertices (increments of 50) and edge probability between 0.1
and 0.9 (increments of 0.1). **EMSN** and **Semi-naive** used the
same sequence of graphs. Figures 15(a) and 15(b) show the
results of the analysis.

![Fig. 15. Ratio Semi-naive/EMSN Derivations - Counting Paths](image)

Each point on a line represents the ratio of **Semi-naive** to
**EMSN** for number of facts derived (Figure 15(a)) or number
of \( \delta \) facts (Figure 15(b)) for the size of the graph indicated
by the line and edge probability indicated by the x-axis. For
example, in Figure 15(b), **Semi-naive** produces greater than
three times as many \( \delta \) facts as **EMSN** for graphs of size 300
and 250 vertices starting around 0.2 (20%) edge probability.
Figures 15(a) and 15(b) show that for our test graphs, as
the edge probability increases, we observe Counting Paths
using **Semi-naive** requires 1.94-3.48 times as many derivations
than Counting Paths using **EMSN**. We attributed the increase
in ratio as edge probability increases to the increased edge
density allowing **EMSN** greater opportunity to use facts earlier
in derivations and thus prune the search space. These results
validate **EMSN**’s efficiency for evaluating recursive programs
with count and sum monotonic aggregates.

B. Storage Design Evaluation

In this experiment, we tested how each of the storage
designs presented in Section VIII-A would perform on DAGs.
Figure 16 shows the (geometric) average execution time and
memory utilization, along with minimum and maximum val-
ues, on 45 random 250-vertex DAGs (5 graphs for each edge
probability from 0.1 to 0.9) for each design. Designs are shown
from worst to best average execution time performance in left-
to-right order in Figures 16.

Recall the descriptions from Sections V-B and VIII-A. **D2**
is the **Double** design as described in Table III executed using
**UHT** with **B+Tree** indexes. **D1** is **Double** using a **B+Tree**
TupleStore for the \((\mathbb{R}, \mathbb{T}, P_T)\) relation\(^{12}\). **D3**-**D5** and **D6**-**D8** are
**KeyValue** designs executed using **UHT** and **BAT**, respectively.
Figure 16 shows **D6**-**D8**, the three **BAT** **KeyValue** type designs,

\(^{12}\)The \((\mathbb{R}, \mathbb{T}, P_T)\) relation contains many times more tuples than **Double’s**
\((\mathbb{R}, \mathbb{N})\) relation (still **UHT**), which is why only it was made **B+Tree** here.
able to execute a version of the Counting Paths program in
Paths program, and BOM queries such as subparts explosion,
Using SociaLite, we were unable to execute the Counting
same slow performance as we did with APSP (Section VI-A).
execute the Counting Paths program, but we experienced the
were nearly twice that of D7 and D8. We conclude the
utilization (969MB vs. 989MB) but lower maximum memory
Compared with D7, D8 had slightly better memory average
as having had the best execution time performance. D7 and
D8 had the lowest average execution time performance with
B+Tree having better maximum (51s vs. 62s) execution time.
Compared with D7, D8 had slightly better memory average
utilization (969MB vs. 989MB) but lower maximum memory
utilization (1.4GB vs. 2GB). Note, D1 and D2 had lowest
average memory utilization but their average execution times
were nearly twice that of D7 and D8. We conclude the
B+Tree design using BAT shows the most promising among
our designs for efficiently supporting mcount and msnum as
it balances good average execution time performance with
average memory utilization.

C. Discussion
Finding other system with which to perform an experimen-
tal comparison for mcount and msnum proved challenging.
Support for count and sum aggregates that can be used in
recursion is not as mature as that of min and max aggregates.
Using LogicBlox version 4, we were able to express and
eexecute the Counting Paths program, but we experienced the
same slow performance as we did with APSP (Section VI-A).
Using SociaLite, we were unable to execute the Counting
Paths program, and BOM queries such as subparts explosion,
produced results different from ground truth. Lastly, we were
able to execute a version of the Counting Paths program in
DLV, but again, the results were different from ground truth.

X. EXAMPLE PROGRAMS
This section includes additional programs showing DeAL’s
expressiveness and support for a variety of programs. More
examples are found in [29] and on the DeALS website13.

Example 3: How many days until delivery?
\[ r1.\text{delivery}(\text{Part}, \text{mmax}(\text{Days})) \leftarrow \text{basic}(\text{Part}, \text{Days}, \_). \]
\[ r2.\text{delivery}(\text{Part}, \text{mmax}(\text{Days})) \leftarrow \text{assb}(\text{Part}, \text{Sub}, \_), \text{delivery}(\text{Sub}, \text{Days}). \]
\[ r3.\text{actualDays}(\text{Part}, \text{mmax}(\text{Days})) \leftarrow \text{delivery}(\text{Sub}, \text{Days}). \]

Example 4: What is the maximum cost of a part?
\[ r1.\text{tcost}(\text{Part}, \text{msnum}(\langle \text{Part}, \text{Cost} \rangle)) \leftarrow \text{basic}(\text{Part}, \text{Cost}, \_). \]
\[ r2.\text{tcost}(\text{Part}, \text{msnum}(\langle \text{Sub}, \text{Cost} \rangle)) \leftarrow \text{assb}(\text{Part}, \text{Sub}, \_), \text{tcost}(\text{Sub}, \text{Cost}). \]
\[ r3.\text{tcost}(\text{Part}, \text{mmax}(\text{Cost})) \leftarrow \text{tcost}(\text{Part}, \text{Cost}). \]

Example 3 and 4 are the Bill of Materials (BOM) program
for finding the days required to deliver a part and program for
computing the max cost of a part from the cost of its subparts,
respectively. The assb predicate denotes each part’s required
subparts and number required and basic denotes the number
of days for a part to be received and the part’s cost.

Example 5: Viterbi Algorithm
\[ r1.\text{calcV}(0, X, \text{mmax}(L)) \leftarrow s(0, EX), p(X, EX, L1), p_i(X, L2), \]
\[ r2.\text{calcV}(T, Y, \text{mmax}(L)) \leftarrow s(T, EV), p(Y, EV, L1), T1 = T - 1, \]
\[ t(X, Y, L2), \text{calcV}(T1, X, L3), L = L1 + L2 + L3. \]
\[ r3.\text{viterbi}(T, Y, \text{mmax}(L)) \leftarrow \text{calcV}(T, Y, L). \]

Example 5 is the Viterbi algorithm for hidden Markov
models. \( t \) denotes the transition probability \( L2 \) from state
\( X \) to \( Y \). \( s \) denotes the observed sequence of length \( L4+1 \). \( p_i \)
denotes the likelihood \( L2 \) that \( X \) is the initial state. \( p \) denotes
the likelihood \( L1 \) that state \( X \) (\( Y \)) emitted \( EX \) (\( EV \)). \( r1 \) finds
the most likely initial observation for each \( X \). \( r2 \) finds the most
likely transition for observation \( T \) for each \( Y \). Lastly, \( r3 \) finds
the max likelihood for each \( T, Y \).

Example 6: Max Probability Path
\[ r1.\text{reach}(X, Y, \text{mmax}(P)) \leftarrow \text{net}(X, Y, P), \]
\[ r2.\text{reach}(X, Z, \text{mmax}(P)) \leftarrow \text{reach}(X, Y, P), \text{reach}(Y, Z, P2), \]
\[ P = P1 + P2. \]

\[ r3.\text{maxP}(X, Y, \text{mmax}(P)) \leftarrow \text{reach}(X, Y, P). \]

Example 6 is the non-linear program for computing the
maximum probability path between two nodes in a network.
\( \text{net}(X, Y, P) \) denotes the probability of reaching \( Y \) from \( X \) is \( P \).

XI. FORMAL SEMANTICS
So far we have worked with the operational semantics of our
monotonic aggregates and shown how this is conducive to
the expression of algorithms by programmers. While most
users only need to work at this level, it is important that we also
show how this coincides with the formal semantics discussed
in those two DatalogFS papers, inasmuch as properties such
as least fixpoint, and stable models will follow from it.

We can start with the inspiring example by [9] where some
people will come to the party for sure whereas others only join
in when least three of their friends will come. The basic idea
is that with cntwillcome each person watches the number of
friends that willcome grow, and once that number reaches 3
our person join the party too. For that, rather than the final
count used in [9] we can use the continuous count aggregate
mcount that enumerates all the integers until the actual
maximum, i.e. it returns the integer interval \([1, M]\) where \( M \)
is the actual maximum.

Example 7: Who will come to the party?
\[ r1.\text{willcome}(X) \leftarrow \text{sure}(X), \]
\[ r2.\text{willcome}(Y) \leftarrow \text{cntwillcome}(Y, N), N \geq 3. \]
\[ r3.\text{cntwillcome}(Y, mcount(X)) \leftarrow \text{friend}(Y, X), \text{willcome}(X). \]

The use of mcount over count is here justified on the
ground of performance, since there is no point in counting
all the friends of popular people, if only three are required.
Even more important is that while count is non-monotonic
(unless we use the special lattices suggested by [9]), mcount
is monotonic in the lattice of set containment used by the

---

standard Datalog. So no ad hoc semantic extension is needed and concepts and techniques such as magic sets, perfect models and stable models can be immediately generalized to programs with $\text{mcount}$.

A. DeAL Interval Semantics

The lessons learned with $\text{mcount}$, tells us that we can derive the monotonic counterpart of an aggregate by simply assuming that it produces an interval of integer values, rather than just one value. In the following we (i) apply this idea to max and min to obtain $\text{mmax}$ and $\text{mmin}$, and then (ii) generalize these monotonic aggregates to arbitrary numbers, and show that the least fixpoint computation under this formal interval-based semantics can be implemented using the Semi-naive semantics use in Section III, under general conditions that hold for all our examples. Because of space limitations the discussion is kept at an informal level: formal proofs are given in [17], [18].

Now, say $\text{Ub}$ represents the lower bound for all negative numbers supported by the system architecture. We can use the interval $[\text{Ub}, N]$ to represent any number regardless of its sign. The result of unifying the sets representing these numbers is a set representing the max, independent of whether this is positive or negative. Using Figure 17(b) as an example, with $N_1$ and $N_2$ represented by $[\text{Ub}, N_1]$ and $[\text{Ub}, N_2]$, respectively, then $[\text{Ub}, N_1] \cup [\text{Ub}, N_2]$ represents the larger of the two, i.e. $\text{max}(N_1, N_2) = N_2$. Thus, we can support negative numbers.

Minimum Costs Now, say $\text{Ub}$ represents the upper bound for all positive numbers supported by the system architecture. We can represent the set of all numbers between $N$ and $\text{Ub}$, i.e. the interval $[N, \text{Ub}]$, as the number $N$. Observe that a number smaller than $N$ is represented by an interval that contains the interval $[N, \text{Ub}]$. As before, if we take a union of two or more such representations, the result is the largest interval. Using Figure 17(c) as an example, with $N_1$ and $N_2$ represented by $[N_1, \text{Ub}]$ and $[N_2, \text{Ub}]$, respectively, then $[N_1, \text{Ub}] \cup [N_2, \text{Ub}]$ represents the smaller of the two, i.e., $\text{min}(N_1, N_2) = N_2$.

As another example of these semantics, consider the first derivation in Figure 2 - $\text{spath}(a, c, 2)$ was derived because the previous value for $\text{c}$ was 3. In the interval semantics, $\text{spath}(a, c, 2)$ would be represented as $[2, \text{Ub}]$, and 3 as $[3, \text{Ub}]$, thus we have $([2, \text{Ub}] \cup [3, \text{Ub}])$ is $\text{min}(2, 3) = 2$.

B. Normal Programs

DeAL programs that only use monotonic arithmetic and monotonic boolean (comparison) functions on values produced by monotonic aggregates will be called normal. All practical algorithms we have considered only require the use of normal DeAL programs. Two classes of normal programs exist.

Class 1 This class of normal programs uses monotonic aggregates to compute the max (min) values for use outside the recursion (e.g. to return the facts with the final max (min) values). Programs in this class include Examples 1, 2 which use a stratified $\text{max} (\text{min})$ aggregate at the end of the fixpoint iteration to select the max (min) value produced by the recursive rules. Examining the intermediate results of Class 1 programs: at each step of the fixpoint iteration, we have (i) the max (min) value (ii) values less than the max value (greater than the min value). However, we do not need the values in (ii), as long as the values in the head are computed from those in the body via an arithmetic function that is monotonic.

Class 2 Values produced by monotonic aggregates in Class 2 normal programs are not passed to rule heads, but are tested against conditions in the rule body. Here too, as long as the functions applied to the values are monotonic, the rules are satisfied if and only if they are satisfied for the max (min) values. Example 7 is a Class 2 normal program.

Normal Program Evaluation Recall the algorithm for MASN in Figure 6. Let us call $L$ the set produced by $T_E(M)$ or $T_R(\delta S)$, $F$ the set produced from applying $\text{getLast}()$ to $L$. For Semi-naive $\delta S$ will be $L$, whereas for MASN, $\delta S$ will be $F$. Let $W = L - F$, $W = \emptyset$ when, for facts from monotonic aggregate predicates, each group with facts derived during the iteration has only one fact. For an iteration, if $W = \emptyset$, MASN evaluates the program the same as Semi-naive. Otherwise, $W$ contains facts that will not lead to final answers for Class 1 normal programs and for Class 2 normal programs, any condition satisfied (in a rule body) by a fact in $W$ will also be satisfied by the fact of the same group in $F$. Thus, MASN does not need to evaluate $W$. In the next iteration, Semi-naive will derive all of the same facts as MASN, but also derive facts evaluating $W$. We have already established that these facts, because they were derived from $W$, will not lead to final answers or to satisfying rule bodies, and thus, MASN does not need to derive facts with these either.

Theorem 11.1: A normal program with monotonic aggregates evaluated using MASN will produce the same result as that program evaluated using Semi-naive.

Recall the algorithm for MASN in Figure 8. Assume we are evaluating a normal program with Semi-naive. Let us call $K_{sn}$
the set of facts used in derivations (∪ of all $\delta S$) by Semi-naive. Now assume we are evaluating the same normal program with EMSN. Let us call $K_{emsn}$ the set of all facts used in derivations by EMSN, which means $K_{emsn}$ contains facts that were retrieved from the aggregate’s relation, meaning they had the current value for the group at the time the fact was used in derivation. Now, let $C = K_{sn} - K_{emsn}$. If $C = \emptyset$, EMSN evaluates the program the same as Semi-naive. Otherwise, $C$ contains facts that were not used by EMSN because at the time of derivation, the values in the aggregate’s relation for those facts’ groups were greater (mmax, mcount, msum) or lesser (mmin) than the aggregate value in the fact. We know $K_{sn} \cap K_{emsn} = K_{emsn}$ and $K_{emsn} \subset K_{sn}$ because EMSN will ignore facts to use in derivations that Semi-naive will use in derivations, but EMSN will not use different facts than Semi-naive. Stated another way, Semi-naive will attempt derivations with all facts EMSN attempts derivations with. Therefore, for Class 1 normal programs, each fact in $C$ was not going to lead to final answers. For Class 2 normal programs, any condition satisfied (in a rule body) by a fact in $C$ would have also been satisfied by the value that was instead used. EMSN does not need to evaluate $C$. We have:

Theorem 11.2: A normal program with monotonic aggregates evaluated using EMSN will produce the same result as that program evaluated using Semi-naive.

XII. Related Work

We have discussed the contributions of many works including [2], [9], [23] in previous sections. Focusing on extrema aggregates, [25] proposes rewriting programs by pushing the aggregate into the recursion and using Greedy Fixpoint evaluation to select the next min/max value to execute. [26] proposes rule rewriting using aggregate selections to identify irrelevant facts that are to be discarded by an extended version of Semi-naive evaluation. Nondeterministic constructs and stable models are used to define monotone aggregates in [27]. The Bloom\textsuperscript{2} language [16] for distributed programming uses logical monotonicity via built-in and user-defined lattice types to support eventual consistency.

XIII. Conclusion

With the renaissance of Datalog, the monotonicity property has been placed at the center of its ability to provide a declarative treatment of distributed computation [30]. In this paper, we have shown how this property can be extended to the aggregate involved in recursive computations while preserving the syntax, semantics, and optimization techniques of traditional Datalog. The significance of this result follows from the fact that this problem had remained long unsolved, and that many new applications can be expressed with the proposed extensions that make them amenable to parallel execution on multiprocessor and distributed systems. Future lines of work include supporting KDD algorithms and parallel/distributed implementations of techniques from this paper.

REFERENCES