Optimizing Recursive Queries with Monotonic Aggregates in $DeALS$

Alexander Shkapsky, Mohan Yang, Carlo Zaniolo
University of California, Los Angeles
{shkapsky, yang, zaniolo}@cs.ucla.edu

***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 support for aggregation in recursion. Aggregates in recursive queries are essential in many important applications and are increasingly being applied in areas such as computer networking [1] and social networks [2]. Many significant applications require iterating over counts or probability computations, including machine learning algorithms for Markov chains and hidden Markov models, and data mining algorithms such as Apriori. 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 hierarchy. 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]–[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, namely that determining the correct lattices by programmers and compilers would be quite difficult [15], and this 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. Firstly, Hellerstein et al., after announcing a resurgence of Datalog, showed that monotonicity in special lattices can be very useful in proving formal properties such as eventual consistency [16]. Secondly, we see monotonic aggregates making a strong comeback 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]. SociaLite [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 ***De**ductive **A**pplication **L**anguage* ($DeAL$) 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.,

XY-stratification and the choice construct that were found quite useful in 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, supplementary magic sets 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 by comparing not only with SociaLite but also with systems such as DLV [23] and LogicBlox [24] that realize different performance/generality tradeoffs.

Overview. The first of two main parts of this paper begins with Section II which presents the syntax and semantics for the min (mmin) and max (mmax) monotonic aggregates. Section III discusses the evaluation and optimization of monotonic aggregate programs. Section IV presents implementation details for mmin and mmax and the $DeALS$ storage manager. Section V presents experimental results for Sections II-IV. The second part of this paper begins with Section VI discussing the count (mcount) and sum (msum) monotonic aggregates, followed by their implementation in Section VII and experimental validation in Section VIII. Section IX provides additional $DeAL$ program examples. Section X presents the formal semantics on which our aggregates are based. Additional related works are reviewed in Section XI and we conclude in Section XII.

Preliminaries. A Datalog program P is a finite set of rules, or Horn Clauses, where rule r in P has the form $A \leftarrow A_1, \ldots, A_n$. The *atom* A is the $head$ of r. $A_1, \ldots, A_n$, the $body$ of r, are *literals*, or *goals*, where each literal can be either a positive or negated atom. An atom has the form $p(t_1, \ldots, t_j)$ where p is a *predicate* and $t_1, \ldots, t_j$ 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].

II. MMIN AND MMAX MONOTONIC AGGREGATES

An mmin or mmax monotonic aggregate rule has the form:

$$\mathtt{p(K_1, \ldots, K_m, aggr\langle T\rangle)} \leftarrow \mathtt{Rule\ Body.}$$

In the rule head, $\mathtt{K_1}, \ldots, \mathtt{K_m}$ are the zero or more *group-by arguments* we also refer to as $\overline{\mathtt{K}}$, $\mathtt{aggr} \in \{\mathtt{mmax}, \mathtt{mmin}\}$ is the *monotonic aggregate*, and T, the *aggregate term*, is a variable.

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

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 length of the shortest path between each pair of connected vertices in a weighted directed graph.

Example 1: APSP with mmin

$$r1.\ \mathtt{spaths(X,Y,mmin\langle D\rangle)} \leftarrow \mathtt{edge(X,Y,D)}.$$
$$r2.\ \mathtt{spaths(X,Y,mmin\langle D\rangle)} \leftarrow \mathtt{spaths(X,Z,D1), edge(Z,Y,D2), D = D1 + D2}.$$
$$r3.\ \mathtt{shortestpaths(X,Y,min\langle D\rangle)} \leftarrow \mathtt{spaths(X,Y,D)}.$$

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 (*r1*, *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 (*r3*) to select the shortest path for each X, Y.

APSP By Example. Next, we walk through a *Semi-naive* evaluation of the APSP program from Example 1.

$$\mathtt{edge(a,b,1).}\ \ \mathtt{edge(a,c,3).}\ \ \mathtt{edge(a,d,4).}$$
$$\mathtt{edge(b,c,1).}\ \ \mathtt{edge(b,d,4).}\ \ \mathtt{edge(c,d,1).}$$

Fig. 1. edge Facts for Example 1

First, *r1* in Example 1, the exit rule, is evaluated on the edge facts in Fig. 1. In the rule head in *r1*, X and Y, the non-aggregate arguments, are the *group-by arguments*. The mmin aggregate is applied to each of the six edge facts and six spaths facts are successfully derived (not displayed to conserve 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, mmin is now initialized with a value for each group.

$$\mathtt{spaths(a,c,2)} \leftarrow \mathtt{spaths(a,b,1), edge(b,c,1), 2=1+1.}$$
$$\mathtt{FAIL} \leftarrow \mathtt{spaths(a,b,1), edge(b,d,4), 5=1+4.}\ \ [\mathrm{i}]$$
$$\mathtt{FAIL} \leftarrow \mathtt{spaths(a,c,3), edge(c,d,1), 4=3+1.}\ \ [\mathrm{ii}]$$
$$\mathtt{spaths(b,d,2)} \leftarrow \mathtt{spaths(b,c,1), edge(c,d,1), 2=1+1.}$$

Fig. 2. Derivations of Example 1, *r2* - Iteration 1

Semi-naive evaluates the recursive *r2* rule from Example 1 using the six spaths derived by *r1*. Fig. 2 displays four

[1]SQL 2003 max, min, count and sum aggregates on the unlimited preceding window are similar to $DeAL$'s monotonic aggregates.

derivations attempted by *r2* in its first iteration. Derivations not displayed failed to join spaths and edge facts. The first attempt results in a new spaths fact because $\texttt{spaths}(\texttt{a},\texttt{c},2)$ has an aggregate value less than the previous value for $(\texttt{a},\texttt{c})$, which was 3 (from *r1*). The failures denoted [i] and [ii] occurred because the facts to be derived would have aggregate values not less than the previous value for $(\texttt{a},\texttt{d})$, which is 4. Finally, $\texttt{spaths}(\texttt{b},\texttt{d},2)$ is derived ($2 < 4$ for $(\texttt{b},\texttt{d})$).

$$\texttt{spaths}(\texttt{a},\texttt{d},3) \leftarrow \texttt{spaths}(\texttt{a},\texttt{c},2), \texttt{edge}(\texttt{c},\texttt{d},1), 3{=}2{+}1.$$

Fig. 3. Derivations of Example 1, *r2* - Iteration 2

Using the two facts derived in Fig. 2, *Semi-naive* performs a second iteration using *r2*. As displayed in Fig. 3, $\texttt{spaths}(\texttt{a},\texttt{d},3)$ is derived because $(3 < 4)$ for $(\texttt{a},\texttt{d})$. Now, no new facts can be derived and a fixpoint is reached.

$$\texttt{shortestpaths}(\texttt{a},\texttt{c},2) \leftarrow \{\texttt{spaths}(\texttt{a},\texttt{c},3), \texttt{spaths}(\texttt{a},\texttt{c},2)\}$$
$$\texttt{shortestpaths}(\texttt{a},\texttt{d},3) \leftarrow \{\texttt{spaths}(\texttt{a},\texttt{d},4), \texttt{spaths}(\texttt{a},\texttt{d},3)\}$$
$$\texttt{shortestpaths}(\texttt{b},\texttt{d},2) \leftarrow \{\texttt{spaths}(\texttt{b},\texttt{d},4), \texttt{spaths}(\texttt{b},\texttt{d},2)\}$$

Fig. 4. Derivations of Example 1, *r3*

Lastly, *r3* is evaluated over the spaths facts derived during recursion and uses a stratified min aggregate to derive only the fact with the shortest path for each group. Fig. 4 displays derivations of *r3* on groups that had multiple facts derived in recursion showing why rules like *r3* are necessary with our semantics. In Section IV, we will discuss optimizations so rules such as *r3* do not have to be evaluated.

III. Monotonic Aggregate Evaluation

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

In Fig. 5, the algorithm for *Semi-naive*, M is the initial model (database), S contains all facts obtained thus far, δS and $\delta S'$ contain facts obtained during the previous and current iteration, respectively, and T_E and T_R are the Immediate Consequence Operator (ICO) for the exit rule(s) and the recursive rule(s), respectively. The algorithm evaluates as follows. Firstly, *Semi-naive* applies T_E (i.e. the exit rules) on M to derive the first set of new δ facts δS (line 2). Then, until no new facts are derived during an iteration, *Semi-naive* evaluates T_R on δS to derive new facts to be used in the next iteration. The new set of δ facts ($\delta S'$) is produced only after the removal of facts found in previous steps (line 5).

```
1: S := M;
2: δS := T_E(M);
3: S := S ∪ δS;
4: while δS ≠ ∅ do
5:     δS' := T_R(δS) − S;
6:     S := S ∪ δS';
7:     δS := δS';
8: return S;
```

Fig. 5. *Semi-naive Evaluation*

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

Although *Semi-naive* efficiently evaluates general Datalog programs, monotonic aggregate programs can be evaluated with even greater efficiency than *Semi-naive* provides. The *max-based optimization* [18] identified that 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 *Monotonic Optimization*. The intuition behind the *Monotonic Optimization* is that with our monotonic aggregates, monotonicity is preserved and values other than the max (mmax) or min (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 (mmax) or least (mmin) aggregate value, making this fact the only fact for the group that we need to produce for the next iteration.

A. Monotonic Aggregate Semi-naive Evaluation

```
1: S := M;
2: δS := getLast(T_E(M));
3: S := S ∪ δS;
4: while δS ≠ ∅ do
5:     δS' := getLast(T_R(δS)) − S;
6:     S := S ∪ δS';
7:     δS := δS';
8: return S;
```

Fig. 6. *Monotonic Aggregate Semi-naive Evaluation* (*MASN*)

The *Monotonic Optimization* enables an optimized *Semi-naive* for monotonic aggregates we call *Monotonic Aggregate Semi-naive Evaluation* (*MASN*). Fig. 6 is the algorithm for *MASN*, which closely resembles *Semi-naive*. *MASN*'s differences with *Semi-naive* are as follows. Here we use $getLast()$[2] 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* 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 where δ facts derived 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 II-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.

Fig. 7 shows the derivations of interest extracted from Fig. 2. We see the second derivation performed using

[2] $DeALS$ supports *MASN* by maintaining a single fact per group in $\delta S'$.

Example 1 *r2* evaluation with *Semi-naive* or *MASN*
spaths(a, c, 2) ← spaths(a, b, 1), edge(b, c, 1), 2=1+1.
FAIL ← **spaths(a, c, 3)**, edge(c, d, 1), 4=3+1.

Fig. 7. Example of Iteration Boundary of *MASN*

spaths(a, c, 3) (from δS) and resulting in failure because the value for (a, d) was 4. However, at the time the derivation is attempted, spaths(a, c, 2), the result of the immediately previous derivation, existed. Had spaths(a, c, 2) been used, spaths(a, d, 3) would have been derived here, instead of requiring another iteration (Fig. 3).

To further capitalize on the *Monotonic Optimization*, we propose *Eager Monotonic Aggregate Semi-naive Evaluation* (*EMSN*). With *EMSN*, facts produced from monotonic aggregate rules are immediately available to be used in derivations. *EMSN* evaluates recursive rules with monotonic aggregates in a fact-oriented (fact-at-a-time) manner and the facts to use in an iteration are determined by the set of groups (keys) that had aggregate facts derived during the previous iteration. With *EMSN*, derivations with monotonic aggregates are always performed with the current aggregate value for the group.

Fig. 8 is *EMSN*. In *EMSN*, recursive rules with monotonic aggregates are evaluated fact-at-a-time and all other recursive rules are evaluated using *Semi-naive*. Rules are partitioned into two sets, each with its own ICOs – T_{E_A} and T_{R_A} are the ICO for the monotonic aggregate exit and recursive rules, respectively, and T_{E_N} and T_{R_N} are the ICO for the remaining exit and recursive rules, respectively. T_{R_A} will be applied on one fact at a time. δS_A and $\delta S'_A$ are the sets of facts obtained during the previous and current iteration, respectively, for the monotonic aggregate rules, while δS and $\delta S'$ are the sets of facts obtained during the previous and current iteration, respectively, for the remaining rules. $\delta S_{A_{Keys}}$ is the set of keys for the aggregate groups that had facts derived during the previous iteration. M is the initial model, S contains all facts obtained thus far. $newfact$ is a fact derived from a single application of T_{R_A}.

Important points of Fig. 8 are as follows. We use $getKeys()$ to project out the aggregate value from the aggregate facts to produce the set of facts representing groups (keys) to use in derivations in the next iteration. For example, $getKeys(\{\texttt{spaths(a, b, 1)}\})$ would produce $\{\texttt{spaths(a, b)}\}$.

```
1:  S := M;
2:  δS_A := T_{E_A}(M);
3:  δS_{A_Keys} := getKeys(δS_A);
4:  δS := T_{E_N}(M);
5:  S := S ∪ δS ∪ δS_A;
6:  while δS ≠ ∅ and δS_{A_Keys} ≠ ∅ do
7:      for all key ∈ δS_{A_Keys} do
8:          while (newfact := T_{R_A}(getFact(key))) do
9:              δS'_A := δS'_A ∪ {newfact};
10:     δS' := T_{R_N}(δS) − S;
11:     S := S ∪ δS' ∪ δS'_A;
12:     δS_{A_Keys} := getKeys(δS'_A);
13:     δS := δS'; δS'_A = ∅
14: return S;
```

Fig. 8. *Eager Monotonic Aggregate Semi-naive Evaluation* Sketch (*EMSN*)

$getKeys()$ is applied to the set produced by $T_{E_A}(M)$ to produce the initial $\delta S_{A_{Keys}}$ (line 3). T_{E_N} (remaining rules) is applied to M to produce the initial δS (line 4). Once in the recursion, individually, each key in $\delta S_{A_{Keys}}$ is used to retrieve its group's current fact from the aggregate relation ($getFact(key)$), which T_{R_A} is then applied to (line 8). Successful derivations result in $newfact$ being added to $\delta S'_A$ (line 9). Then, T_{R_N} (remaining rules) is applied to δS and duplicates are eliminated producing $\delta S'$, the set of non-monotonic-aggregate facts to be used in the next iteration (line 10). $getKeys(\delta S'_A)$ produces the set of keys to be used in derivations in the next iteration (line 12). This process repeats until no new facts are produced during an iteration.

Example 1 *r2* evaluation with *EMSN*
spaths(a, c, 2) ← spaths(a, b, 1), edge(b, c, 1), 2=1+1.
spaths(a, d, 3) ← **spaths(a, c, 2)**, edge(c, d, 1), 3=2+1.

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

Now, consider the same scenario from Fig. 7, but this time using *EMSN*. As shown in Fig. 9, now after spaths(a, c, 2) is produced, it is immediately used in the next derivation resulting in spaths(a, d, 3) being derived an iteration earlier than with *Semi-naive* or *MASN*. Moreover, spaths(a, c, 3) will not be used in any further derivations as it would result in the derivations of facts that will not lead to a final answer now with the existence of spaths(a, c, 2).

Discussion. With the application of the ICO for recursive monotonic aggregate rules (T_{R_A}) on an individual fact, rather than on a set of facts, *EMSN* 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 δS and δS_A that were derived during the previous iteration. However, because *EMSN* uses the most recent aggregate value for the group, 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 (mmax) or lower (mmin) 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.

We considered an alternative approach for *EMSN* that instead maintains the set of aggregate facts derived during an iteration ($\delta S'_A$) where modification to the aggregate relation results in either an update or insert to $\delta S'_A$. However, with each iteration $\delta S'_A$ would become δS_A and a new $\delta S'_A$ would be started, therefore every modification to the aggregate relation would require both δS_A and the new $\delta S'_A$ to be searched to be updated with the new value, even if the aggregate group is not present in δS_A. Although δS_A and $\delta S'_A$ are generally smaller than the aggregate relation, if an aggregate group has many new results during an iteration, efficiency gained from searching smaller sets instead of searching a larger aggregate relation to retrieve the aggregate value when needed (line 8 in Fig. 8) would be offset by searching these sets many times. Furthermore, this requires a more complicated implementation to properly synchronize facts in multiple data structures.

IV. MMIN AND MMAX IMPLEMENTATION

This section contains details of our system implementation for supporting the `mmin` and `mmax` aggregates.

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 compiler produces an instantiated AND/OR Graph [20] representing the given program, which the interpreter evaluates to produce query results. The interpreter uses tuple-at-a-time pipelining and evaluates nodes of the AND/OR graph representing goals in rule bodies in a left-to-right fashion with intelligent backtracking [19]. The interpreter uses nested loops joins but will create an index and use an index nested loops join if the argument binding analysis identifies a bound join argument. For instance, for *r2* in Example 1, the interpreter will use an index nested loops join – `edge` will be indexed on `Z` (first argument) which is bound by `spaths`.

B. Storage Manager Overview

The $DeALS$ storage manager provides support for main memory storage and indexing for predicate relations. $DeALS$ supports several B+Tree data structure variants for tuple storage and indexing. A B+Tree stores fixed-size keys in internal and leaf nodes and non-key attributes in leaf nodes. Leaf nodes have pointers to their right neighbors to enable fast scanning of the tree. Through testing we determined our implementations perform best on average using a linear key search at both internal and leaf nodes with 256 bytes allocated for keys (e.g., 32 64-bit `long` keys) in each node, which results in shallow trees. $DeALS$ supports *B+Tree TupleStores*, which store tuples in a B+Tree. $DeALS$ also supports an *Unordered Heap TupleStore* (*UHT*) where tuples are stored as fixed-size entries in insertion order in large byte arrays. *UHT* can be given multiple indexes (e.g., B+Tree), which they remain synchronized with at all times. *UHT* enable a highwatermark approach for *Semi-naive* where each iteration is a contiguous range of tuples.

1) B+Tree Aggregators: Early experimentation found aggregation using either i) *UHT* with B+Tree or Linear Hashing-based secondary indexes or ii) B+Tree TupleStores lacking in execution time performance. The *B+Tree Aggregator TupleStore* (*B+AT*) is a B+Tree TupleStore optimized for pipelined aggregation in recursive queries that provides both good read and write performance. *B+AT* 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 search is used in both internal and leaf nodes. In a *B+AT*, aggregation is performed in the leaves, therefore only one search of the tree is needed to retrieve the previous value, compare it with the new value and perform the update.

Unlike with *UHT*, facts in *B+AT* are not easily tracked by reference or range because of node splitting. Therefore, during evaluation of recursive queries with *EMSN*, after an aggregate value is modified, the *B+AT* inserts the modified entry's key into the set of keys, which is also a B+Tree[3], to process for the next iteration ($\delta S_{A_{Keys}}$ in Fig. 8). This approach requires two tree searches with an aggregate value modification – one *B+AT* search which results in a modified aggregate value and one $\delta S_{A_{Keys}}$ search to record the key. Should no modification occur, then only the *B+AT* is searched. To retrieve aggregate facts to use in derivations (line 8 in Fig. 8), a specialized cursor scans $\delta S_{A_{Keys}}$, using each key to retrieve the key's aggregate value from the *B+AT*.

Hash table approaches can be an appealing alternative to B+Trees. In Section V we present experimental results comparing $DeALS$ with a system that utilizes a hash table approach and highlight some of key differences between using B+Trees and hash tables.

C. `mmin` and `mmax` Implementation

The `mmin` and `mmax` implementation tracks the least (`mmin`) or greatest (`mmax`) 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, *B+AT* keys are the group-by arguments with the aggregate value stored in the leaf. *UHT* are given indexes with the group-by arguments as keys. For instance, `spaths` in Example 1 uses *B+AT* with keys (`X`, `Y`) and each `X`, `Y` is 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 store is merely being synchronized with the aggregate relation store. Therefore, we do not allocate the recursive predicate a store, and instead have it read from the monotonic aggregate store.

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 the final values are retrieved from the monotonic aggregate's TupleStore.

V. MMIN & MMAX PERFORMANCE ANALYSIS

All experiments on synthetic graphs were run on a machine with an i7-4770 CPU and 32GB memory running Ubuntu 14.04 LTS 64-bit. The experiments on real-life graphs were run on a machine with four AMD Opteron 6376 CPUs and 256GB memory running Ubuntu 12.04 LTS 64-bit. 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 experiments on systems written in Java were run using Java

[3]Since we scan the set, we use a B+Tree which stores the keys in order and can benefit *EMSN*.

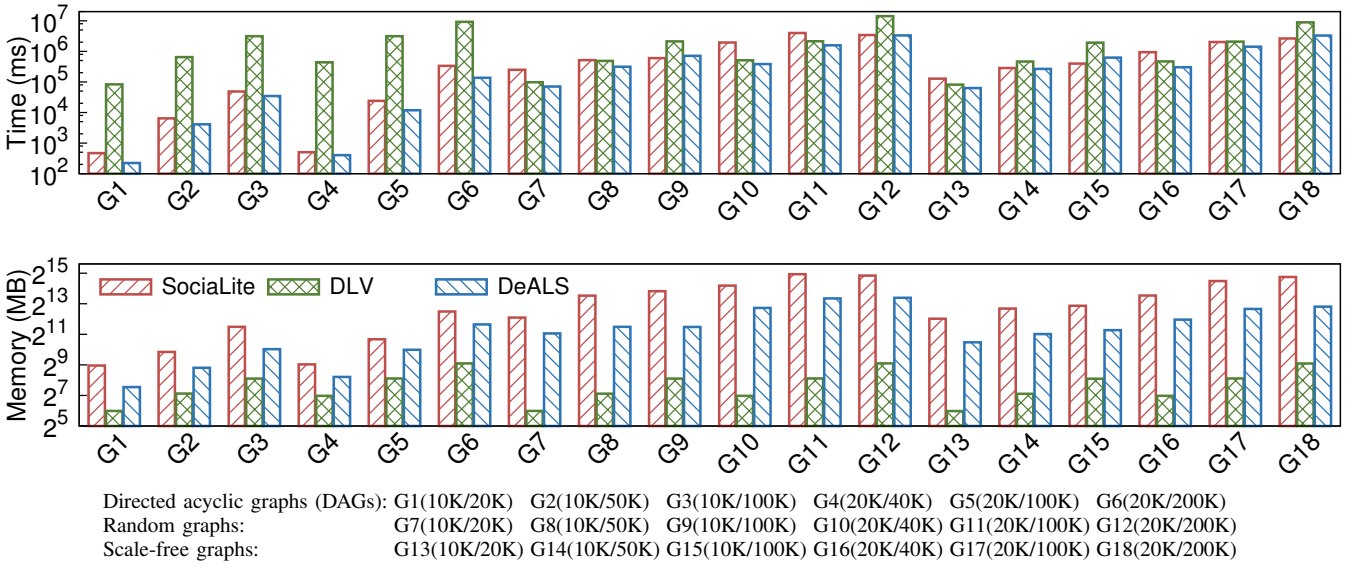


Fig. 10. Execution time and memory utilization of APSP on synthetic graphs

1.8.0 except for SociaLite (0.8.1) which did not support Java 1.8.0. 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 applied to the vertex labels and another random permutation is applied to the edges. The real-life graphs are not shuffled but we relabeled 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. *B+AT* 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 V-B, $DeALS$ used *EMSN* with *B+AT*.

A. Datalog Implementation Comparison

$DeALS$ is a sequential, interpreted, main memory Java implementation. We compare $DeALS$ execution time and memory utilization performance against three Datalog implementations supporting aggregates in recursion — 1) the DLV system[5], the state-of-the-art implementation of disjunctive logic programming; 2) the commercial LogicBlox system, which supports aggregates in recursion using *staged partial fixpoint* semantics[6]. From log files produced during execution of recursive queries with aggregates, we determined LogicBlox indeed uses an approach akin to *Semi-naive*, which uses only new facts found in the current iteration in derivations in the next iteration; 3) the SociaLite[7] graph query language, which is compiled into code-generated Java, efficiently evaluates single-source shortest paths (SSSP) queries using an approach with Dijkstra's algorithm-like efficiency [2] and supports a left-linear recursive APSP which it evaluates using *Semi-naive*. We used SociaLite 0.8.1 as it had the best sequential execution time performance of SociaLite versions available to us.

APSP on Synthetic Graphs. Fig. 10 shows the results of APSP on synthetic graphs with random integer edge costs between 1-100 executed with $DeALS$, SociaLite and DLV. We experimented with two versions of LogicBlox — 3.10.15 and 4.1.3. The former does not support aggregates in recursion and although we can express APSP in a stratified program, it only terminates on DAGs, and only G1(0.286s) and G2(18.328s) finish within 24 hours. The latter supports aggregates in recursion however, only G1(4.809s), G2(6.697m), G7(4.774h) and G13(3.977h) finish within 24 hours. We do not report LogicBlox results in Fig. 10 nor for the remaining experiments.

Among the 18 graphs described in Fig. 10, we found SociaLite has the fastest execution time on three graphs and $DeALS$ has the fastest execution time on the remaining 15 graphs. $DeALS$ is more than two times faster than SociaLite on sparse graphs where the average degree of each vertex is only two (e.g., G7, G10 and G16). This advantage decreases as the average degree increases from two to ten. The main reason for this change is due to the different designs used by $DeALS$ 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 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. However, this faster execution time comes

[4]GTgraph, http://www.cse.psu.edu/~madduri/software/GTgraph/.

[5]DLV with recursive aggregates support, http://www.dbai.tuwien.ac.at/proj/dlv/dlvRecAggr/.

[6]LogicBlox 4 migration guide, https://download.logicblox.com/wp-content/uploads/2014/05/LB-MigrationGuide-40.pdf.

[7]https://sites.google.com/site/socialitelang/

at the expense of higher memory utilization. SociaLite uses more than two times the memory as $DeALS$ on all 18 graphs. Although the C-based DLV has significantly lower memory utilization than both Java-based $DeALS$ and SociaLite, DLV is extremely slow compared with both $DeALS$ and SociaLite on DAGs. These results suggest that $DeALS$ achieves the best execution time versus memory utilization trade-off on sparse graphs among the three compared systems.

APSP on Real-life Graphs. Table I shows the results of APSP on three real-life graphs from the Stanford Large Network Dataset Collection[8]. The provided graphs do not have edge costs, therefore we assigned unit cost to each edge. The results are similar to that of synthetic graphs — $DeALS$ executes fastest while DLV has the lowest memory utilization. These results suggest that on real-life workloads the B+Tree-based design (low initialization cost) adopted by $DeALS$ is more favorable than the hash table-based design used by SociaLite.

TABLE I. EXECUTION TIME AND MEMORY UTILIZATION OF APSP ON REAL-LIFE GRAPHS

	HepTh			Gnutella			Slashdot		
	S1	S2	S3	S1	S2	S3	S1	S2	S3
Time(h)	0.98	17.72	0.39	13.31	4.69	0.49	>24.00	>24.00	2.72
Mem(GB)	12.76	0.99	7.19	41.57	0.48	23.46	>89.59	>1.06	64.70

S1, S2, S3 represent SociaLite, DLV and $DeALS$ respectively.
HepTh(28K/353K): High-energy physics theory citation network.
Gnutella(63K/148K): Gnutella peer-to-peer network.
Slashdot(82K/549K): Slashdot social network.

SSSP on Real-Life Graphs. Fig. 11 shows the results of SSSP on five real-life graphs from the USA road networks datasets[9]. For each graph, we evaluate SSSP on ten randomly selected vertices, and we report the min / (geometric) average / max execution time in the form of error bars. Since execution time captured for DLV includes time for loading the graph, evaluating the query and outputting the result, and query evaluation only accounts for a small percentage of overall time observed, timing for DLV is less informative for this experiment and thus we only report results for SociaLite and $DeALS$. SociaLite generates a Java program that evaluates the query using the Dijkstra's algorithm. The generated code achieves more than one order of magnitude speedup comparing to LogicBlox [2]. However, our interpreted $DeALS$ is faster than the code-generated SociaLite for SSSP on the road network graphs as shown in Fig. 11. This result is not surprising in the sense that *EMSN* optimizes *Semi-naive*, and the Bellman-Ford algorithm (equivalent to *Semi-naive*) usually yields comparable performance with the Dijkstra's algorithm on large sparse graphs.

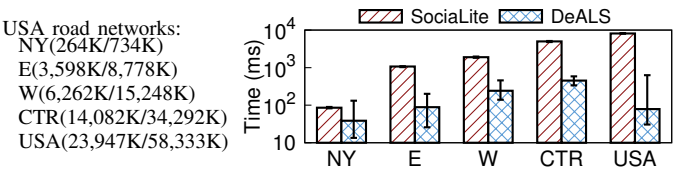


Fig. 11. Execution time of SSSP on Road Networks Datasets

[8]Stanford large network dataset, http://snap.stanford.edu/data/index.html.

[9]USA road networks, http://www.dis.uniroma1.it/challenge9/download.shtml.

B. $DeALS$ Storage Manager Evaluation

This experiment shows i) how *EMSN* performs relative to *MASN* and ii) how *B+AT* perform relative to *UHT*. We evaluated APSP on synthetic graphs G1, G2, G7, G8, G13 and G14 from Fig. 10. The (geometric) average execution time and memory utilization over the six graphs are displayed in Table II. Using *UHT* with B+Tree indexes, *EMSN* has a lower average execution time than *MASN* by 13%. A noticeable difference in performance is observed when using *B+AT* vs. *UHT* for *EMSN*. *B+AT* is 2.6 times faster than *UHT* and requires only approximately 25% of memory needed by *UHT*.

TABLE II. EVALUATION TYPE BY STORAGE CONFIGURATION

Evaluation Type	Storage Configuration	Time (s)	Memory (GB)
MASN	*UHT* w/ B+Tree index	77.743	4.087
EMSN	*UHT* w/ B+Tree index	68.927	4.131
EMSN	*B+AT*	26.450	1.048

C. Statistical Analysis of Evaluation Methods

To provide a characterization of the relative performance of *EMSN* compared to *Semi-naive*, we perform an analysis over sets of graphs using the statistical estimation technique from [28]. Statistics calculated[10] are 1) total number of facts derived by the recursive predicate (*derived facts*) and 2) total aggregate size of δS across all iterations (δ *facts*). These two statistics help to quantify the amount of computation the evaluation method must perform. For each statistic and each vertex number/edge probability combination, after each run of the program, the statistic is included in the average ($\bar{m}$) until a statistically significant number (30) of different graphs has been used AND $\bar{m}$ is within 5% error with 95% confidence[11].

We perform the analysis comparing APSP evaluated using *EMSN* to APSP evaluated using *Semi-naive*. We use randomly generated DAGs and random graphs with edge probability between 0.1 and 0.9 (increments of 0.1) and random integer edge cost between 1-50. *EMSN* and *Semi-naive* use the same sequence of graphs. On DAGs, *Semi-naive* requires 3-11% more *derived* and δ *facts* and on random graphs requires 13-18% more *derived* and δ *facts* than *EMSN*, respectively.

VI. MCOUNT AND MSUM MONOTONIC AGGREGATES

With efficient support for monotonic count and sum aggregates, $DeALS$ supports many exciting applications.

An `mcount` or `msum` monotonic aggregate rule has the form:

$$\texttt{p}(\texttt{K}_1, \ldots, \texttt{K}_\texttt{m}, \texttt{aggr}\langle(\texttt{T}, \texttt{P}_\texttt{T})\rangle) \leftarrow \texttt{Rule Body}.$$

In the rule head, $\texttt{K}_1, \ldots, \texttt{K}_\texttt{m}$ are the zero or more *group-by arguments* ($\overline{\texttt{K}}$), $\texttt{aggr} \in \{\texttt{mcount}, \texttt{msum}\}$ is the *monotonic aggregate*, and $(\texttt{T}, \texttt{P}_\texttt{T})$ is the *aggregate term* pair passed from the body where T is a variable and $\texttt{P}_\texttt{T}$ is a constant or a variable indicating the partial count/sum contributed by T.

[10]For these statistics, we assume a normal distribution ($N(\mu, \sigma^2)$, with mean μ and variance σ^2).

[11]As in [28], $\bar{m}$ is accepted when $\epsilon < (0.05 * \bar{m})$, where $\epsilon = (1.96 * \sigma)/\sqrt{k}$. σ 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.

As with mmin and mmax, the mcount and msum aggregates are monotonic w.r.t. set-containment and can be used in recursive rules. When viewed as a sequence, the values produced by mcount and msum are monotonic. The mcount and msum aggregate functions 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 J. Note, J maps to T and N_J maps to $\mathtt{P_T}$ in the definition of the aggregate term above. Now, given G, for each element $g \in G$, if $N_J > N_{J_{prev}}$, where $N_{J_{prev}}$ is the memorized previous count (sum) for J or 0 if no previous count (sum) for J exists, mcount (msum) computes the count (sum) for G by summing the maximum partial count (sum) N_J for all J. Since only the maximum N_J for each J is summed, no double counting (summing) occurs. Lastly, msum only computes with positive numbers, thereby ensuring its monotonicity.

A. 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 mcount to explain monotonic counting in $DeAL$.

Example 2: Counting Paths in a DAG

r1. $\mathtt{cpaths(X,Y,mcount\langle(X,1)\rangle)} \leftarrow \mathtt{edge(X,Y)}.$
r2. $\mathtt{cpaths(X,Y,mcount\langle(Z,C)\rangle)} \leftarrow \mathtt{cpaths(X,Z,C)}, \mathtt{edge(Z,Y)}.$
r3. $\mathtt{countpaths(X,Y,max\langle C\rangle)} \leftarrow \mathtt{cpaths(X,Y,C)}.$

In Example 2, *r1* counts each edge as one path between its vertices. In *r2*, any edge(Z, Y) that extends from a computed path count cpath(X, Z, C) establishes there are C distinct paths from X to Y through Z. The $\mathtt{mcount}\langle(\mathtt{Z},\mathtt{C})\rangle$ 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 cpaths is desired. As explained in Section IV-D, *r3* does not have to be evaluated.

Counting Paths By Example. Next, we walk through an evaluation of Counting Paths in Example 2 using *EMSN* to further explain mcount; this explanation is easily generalizable to msum. The edge facts in Fig. 12 are the example dataset.

First, *r1* in Example 2 is evaluated and results in the six cpaths derivations as shown in the Fig. 13. Each cpaths fact has a count of 1 indicating one path between each pair of vertices connected by edge facts. Displayed in the right column of Fig. 13 is the memorized partial count (recall (J, N_J)) for each group. For example, in the first derivation, J=a, N_J=1, $(\mathtt{a}, 1)$ is memorized for group $(\mathtt{a}, \mathtt{b})$.

Facts
edge(a, b).
edge(a, c).
edge(a, d).
edge(b, c).
edge(b, d).
edge(c, d).

Fig. 12. Facts

r1 Successful Derivations	Partial Count
cpaths(a, b, 1) ← edge(a, b).	(a, 1) for (a, b)
cpaths(a, c, 1) ← edge(a, c).	(a, 1) for (a, c)
cpaths(a, d, 1) ← edge(a, d).	(a, 1) for (a, d)*
cpaths(b, c, 1) ← edge(b, c).	(b, 1) for (b, c)
cpaths(b, d, 1) ← edge(b, d).	(b, 1) for (b, d)
cpaths(c, d, 1) ← edge(c, d).	(c, 1) for (c, d)

Fig. 13. Derivations of Example 2, *r1* evaluation

r2 Successful Derivations	Partial Count
cpaths(a, c, 2) ← cpaths(a, b, 1), edge(b, c).	(b, 1) for (a, c)
cpaths(a, d, 2) ← cpaths(a, b, 1), edge(b, d).	(b, 1) for (a, d)*
cpaths(a, d, 4) ← cpaths(a, c, 2), edge(c, d).	(c, 2) for (a, d)*
cpaths(b, d, 2) ← cpaths(b, c, 1), edge(c, d).	(c, 1) for (b, d)

Fig. 14. Derivations of Example 2, *r2* - Iteration 1

EMSN evaluates the recursive *r2* rule from Example 2 using the cpaths derived by *r1*. Fig. 14 shows the successful derivations performed by *r2*. As each cpaths fact is derived, it replaces the previous fact for the group (i.e. cpaths(a, c, 2) replaces cpaths(a, c, 1)). Note the derivation of cpaths(a, d, 2) from joining cpaths(a, b, 1) and 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 Fig. 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 cpaths(a, d, 2). Next, we observe the benefits of using *EMSN* with the derivation of cpaths(a, d, 4). Since cpaths(a, c, 2) existed even though it was derived this iteration, it was used and successfully joined with edge(c, d). Then, the partial counts for (a, d), which are (a, 1), (b, 1), and (c, 2), are summed to produce cpaths(a, d, 4). Finally, with no new facts produced after those in Fig. 14, a fixpoint is reached, and since there is no need to evaluate *r3* (Section IV-D) we have our result.

VII. MCOUNT AND MSUM IMPLEMENTATION

In this section, we present implementation details for the mcount and msum aggregates. We use definitions from Section VI (e.g., G). Note, G is a single group produced from the implicit *group-by* for a distinct assignment of $\overline{\mathtt{K}}$, the zero or more *group-by* arguments. We will also refer to the TupleStore descriptions from Section IV-B. Lastly, although we use mcount to present our efficient count/sum technique, this discussion is generalizable to msum.

For mcount and msum, we use an approach based on delta-maintenance (Δ-*Maintenance*) techniques. Recalling our explanation for mcount in Section VI, given a new partial count $N_J > N_{J_{prev}}$, mcount will sum all maximum partial counts to compute the new total count for G. However, rather than recompute the total count, we can instead use Δ-*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 greater N_J. This requires memorizing both N for G and N_J for all J.

TABLE III. STORAGE DESIGN SCHEMAS

Name	**Schema**	**Indexes**
Double	$(\overline{\mathtt{K}}, \mathtt{N}) \mid (\overline{\mathtt{K}}, \mathtt{T}, \mathtt{P_T})$	$\overline{\mathtt{K}} \mid (\overline{\mathtt{K}}, \mathtt{T})$
List	$(\overline{\mathtt{K}}, \mathtt{N}, List[(\mathtt{T}, \mathtt{P_T})])$	$\overline{\mathtt{K}}$
B+Tree	$(\overline{\mathtt{K}}, \mathtt{N}, B{+}Tree[(\mathtt{T}, \mathtt{P_T})])$	$\overline{\mathtt{K}}$
Hashtable	$(\overline{\mathtt{K}}, \mathtt{N}, Hashtable[(\mathtt{T}, \mathtt{P_T})])$	$\overline{\mathtt{K}}$

A. Storage Designs

Table III displays storage designs we investigated for `mcount` and `msum`. Here we use `N` to indicate the current count/sum for the group-by arguments ($\overline{\mathtt{K}}$). As in Section VI, each `T` contributes a partial count/sum $\mathtt{P_T}$ towards a distinct assignment of $\overline{\mathtt{K}}$ (group).

Double uses two relations, one relation ($\overline{\mathtt{K}}, \mathtt{N}$) indexed on $\overline{\mathtt{K}}$ to store tuples containing the group's total aggregate value and a second relation ($\overline{\mathtt{K}}, \mathtt{T}, \mathtt{P_T}$) indexed on ($\overline{\mathtt{K}}, \mathtt{T}$) to store the partial count $\mathtt{P_T}$ for each distinct assignment of ($\overline{\mathtt{K}}, \mathtt{T}$). Early testing showed *Double* using *UHT* without Δ-*Maintenance* taking 2-5 times longer to execute than with Δ-*Maintenance*.

We investigated designs using *KeyValue* type columns as a more efficient way of managing ($\mathtt{T}, \mathtt{P_T}$) pairs. We developed three single relation designs ($\overline{\mathtt{K}}, \mathtt{N}$, $KeyValue[(\mathtt{T}, \mathtt{P_T})]$), where `N` is the total count for $\overline{\mathtt{K}}$ and $KeyValue[(\mathtt{T}, \mathtt{P_T})]$ is a reference to the tuple's own *KeyValue*-type data structure. The relation is indexed on $\overline{\mathtt{K}}$ and each group has a single tuple. The *KeyValue*-types each represent a different retrieval time complexity; a *List* ($O(n)$) type, a *B+Tree* ($O(log(n))$) 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 (`T`) in internal and leaf nodes and non-key attributes ($\mathtt{P_T}$) in leaf nodes, and uses linear search. Lastly, *List* stores ($\mathtt{T}, \mathtt{P_T}$) pairs ordered by `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 Δ-*Maintenance*.

For the designs shown in Table III, *DeALS* supports *List*, *HashTable* and *B+Tree* with *B+AT* and all designs with *UHT* indexed as shown. For example, using *r1*, *r2* from Example 2, with *B+Tree*, the *B+AT* would have `X, Y` as keys and each entry in a leaf would have the current total count `N` and a reference to a B+Tree *KeyValue*-type to store ($\mathtt{T}, \mathtt{P_T}$) pairs. This design is essentially a B+Tree of B+Trees.

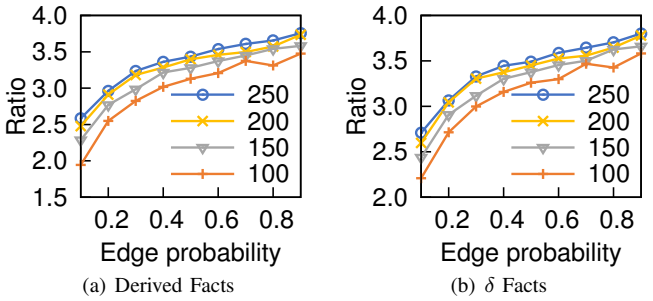
VIII. MCOUNT AND MSUM PERFORMANCE ANALYSIS

Configuration. B+Tree TupleStores and indexes, *B+AT* and the *B+Tree KeyValue* design were configured with 256 bytes allocated for keys in each node (internal and leaf). The *Hashtable KeyValue* design used a directory and segment size of 256, 16 initial buckets and load factor of 10.

A. Statistical Analysis of Evaluation Methods

We also perform the statistical analysis described in Section V-C comparing the evaluation of Counting Paths (Example 2) using *EMSN* to Counting Paths using *Semi-naive*. The experiment uses 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* use the same sequence of graphs.

Fig. 15(a) and Fig. 15(b) show the results of the analysis. Each point on a line represents the ratio of *Semi-naive* to *EMSN* for number of *derived facts* (Fig. 15(a)) or number of δ *facts* (Fig. 15(b)) for the size of the graph indicated by the line and edge probability indicated by the x-axis. For example, in Fig. 15(b), *Semi-naive* produces more than three times as many δ facts as *EMSN* for graphs of 200 and 250 vertices starting at



(a) Derived Facts (b) δ Facts

Fig. 15. Ratio *Semi-naive*/*EMSN* Derivations - Counting Paths

0.2 (20%) edge probability. Fig. 15(a) and Fig. 15(b) show that for the test graphs, Counting Paths using *Semi-naive* derives 1.94 to 3.48 times more facts than *EMSN*. As edge probability increases, so does the ratio between *Semi-naive* and *EMSN* because the higher edge probability allows *EMSN* to derive and use facts earlier, which prunes the search space faster.

B. Storage Design Evaluation

This experiment tests how each of the storage designs presented in Section VII-A performed on DAGs. Fig. 16 shows the (geometric) average execution time and memory utilization, along with minimum and maximum values, on 45 random 250-vertex DAGs (5 graphs for each edge probability from 0.1 to 0.9) for each design. In Fig. 16 results are shown in left-to-right order from worst to best average execution time performance.

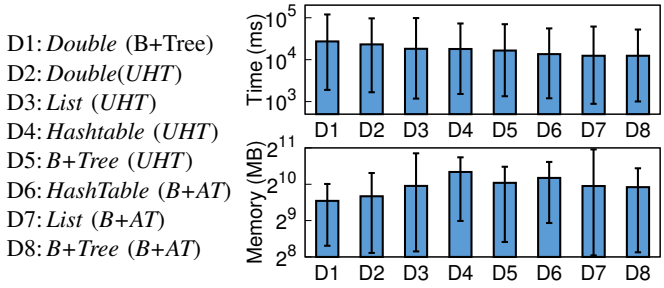


Fig. 16. `mcount` and `msum` Storage Design Performance

Recall the TupleStore descriptions from Section IV-B and storage designs from Section VII-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 ($\overline{\mathtt{K}}, \mathtt{T}, \mathtt{P_T}$) relation[12]. D3-D5 and D6-D8 are *KeyValue* designs executed using *UHT* and *B+AT*, respectively. In Fig. 16 we see D6-D8, the three *B+AT KeyValue* designs, have the best execution time performance. D7 and D8 have the lowest average execution time performance with *B+Tree* having better maximum (51s vs. 62s) execution time. Compared with D7, D8 has slightly better memory average utilization (969MB vs. 989MB) but lower maximum memory utilization (1.4GB vs. 2GB). Note, D1 and D2 have lowest average memory utilization but their average execution times are nearly twice that of D7 and D8. Of the designs, D8, the *B+Tree* design using *B+AT*, best balances good average execution time performance with good average memory utilization.

[12]In *Double*, the ($\overline{\mathtt{K}}, \mathtt{T}, \mathtt{P_T}$) relation will contain many times more tuples than the ($\overline{\mathtt{K}}, \mathtt{N}$) relation (still *UHT*), so we focused on optimizing the larger.

C. Discussion

Finding other systems to perform an experimental comparison with mcount and msum proved challenging. Support in Datalog implementations 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 execute the Counting Paths program but experienced similar slow performance as with APSP (Section V-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 using DLV, but again the results were different from ground truth.

IX. Additional $DeAL$ Programs

This section includes additional programs to show $DeAL$'s expressiveness and support for a variety of applications. More examples are found in [29] and on the $DeALS$ website[13].

Example 3: How many days until delivery?

$$
\begin{aligned}
&r1.\ \texttt{delivery}(\texttt{Part}, \texttt{mmax}\langle\texttt{Days}\rangle) &&\leftarrow \texttt{basic}(\texttt{Part}, \texttt{Days}, \_).\\
&r2.\ \texttt{delivery}(\texttt{Part}, \texttt{mmax}\langle\texttt{Days}\rangle) &&\leftarrow \texttt{assb}(\texttt{Part}, \texttt{Sub}, \_),\\
&&&\quad\ \texttt{delivery}(\texttt{Sub}, \texttt{Days}).\\
&r3.\ \texttt{actualDays}(\texttt{Part}, \texttt{max}\langle\texttt{Days}\rangle) &&\leftarrow \texttt{delivery}(\texttt{Part}, \texttt{Days}).
\end{aligned}
$$

Example 4: What is the maximum cost of a part?

$$
\begin{aligned}
&r1.\ \texttt{cost}(\texttt{Part}, \texttt{msum}\langle(\texttt{Part}, \texttt{Cost})\rangle) &&\leftarrow \texttt{basic}(\texttt{Part}, \_, \texttt{Cost}).\\
&r2.\ \texttt{cost}(\texttt{Part}, \texttt{msum}\langle(\texttt{Sub}, \texttt{Cost})\rangle) &&\leftarrow \texttt{assb}(\texttt{Part}, \texttt{Sub}, \texttt{Num}),\\
&&&\quad\ \texttt{cost}(\texttt{Sub}, \texttt{Scost}),\\
&&&\quad\ \texttt{Cost} = \texttt{Scost} * \texttt{Num}.\\
&r3.\ \texttt{totalCost}(\texttt{Part}, \texttt{max}\langle\texttt{Cost}\rangle) &&\leftarrow \texttt{cost}(\texttt{Part}, \texttt{Cost}).
\end{aligned}
$$

Example 3 and 4 are the Bill of Materials (BOM) program for finding the days required to deliver a part and the 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

$$
\begin{aligned}
&r1.\ \texttt{calcV}(\texttt{0}, \texttt{X}, \texttt{mmax}\langle\texttt{L}\rangle) &&\leftarrow \texttt{s}(\texttt{0}, \texttt{EX}),\ \texttt{p}(\texttt{X}, \texttt{EX}, \texttt{L1}),\ \texttt{pi}(\texttt{X}, \texttt{L2}),\\
&&&\quad\ \texttt{L} = \texttt{L1} * \texttt{L2}.\\
&r2.\ \texttt{calcV}(\texttt{T}, \texttt{Y}, \texttt{mmax}\langle\texttt{L}\rangle) &&\leftarrow \texttt{s}(\texttt{T}, \texttt{EY}),\ \texttt{p}(\texttt{Y}, \texttt{EY}, \texttt{L1}),\ \texttt{T1} = \texttt{T} - \texttt{1},\\
&&&\quad\ \texttt{t}(\texttt{X}, \texttt{Y}, \texttt{L2}),\ \texttt{calcV}(\texttt{T1}, \texttt{X}, \texttt{L3}),\\
&&&\quad\ \texttt{L} = \texttt{L1} * \texttt{L2} * \texttt{L3}.\\
&r3.\ \texttt{viterbi}(\texttt{T}, \texttt{Y}, \texttt{max}\langle\texttt{L}\rangle) &&\leftarrow \texttt{calcV}(\texttt{T}, \texttt{Y}, \texttt{L}).
\end{aligned}
$$

Example 5 is the Viterbi algorithm for hidden Markov models. Four base predicates are used — t denotes the transition probability L2 from state X to Y; s denotes the observed sequence of length L+1; pi denotes the likelihood L2 that X is the initial state; p denotes the likelihood L1 that state X (Y) emitted EX (EY). *r1* finds the most likely initial observation for each X. *r2* finds the most likely transition for observation T for each Y. *r3* finds the max likelihood for T, Y.

Example 6: Max Probability Path

$$
\begin{aligned}
&r1.\ \texttt{reach}(\texttt{X}, \texttt{Y}, \texttt{mmax}\langle\texttt{P}\rangle) &&\leftarrow \texttt{net}(\texttt{X}, \texttt{Y}, \texttt{P}).\\
&r2.\ \texttt{reach}(\texttt{X}, \texttt{Y}, \texttt{mmax}\langle\texttt{P}\rangle) &&\leftarrow \texttt{reach}(\texttt{X}, \texttt{Z}, \texttt{P1}),\ \texttt{reach}(\texttt{Z}, \texttt{Y}, \texttt{P2}),\\
&&&\quad\ \texttt{P} = \texttt{P1} * \texttt{P2}.\\
&r3.\ \texttt{maxP}(\texttt{X}, \texttt{Y}, \texttt{max}\langle\texttt{P}\rangle) &&\leftarrow \texttt{reach}(\texttt{X}, \texttt{Y}, \texttt{P}).
\end{aligned}
$$

[13] $DeALS$ website, http://wis.cs.ucla.edu/deals.

Example 6 is the non-linear program for computing the max probability path between two nodes in a network. The net predicate denotes the probability P of reaching Y from X.

X. 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 Datalog^{FS} papers [17], [18], inasmuch as properties such as least fixpoint and stable models will follow from it.

We start with the example inspired by [9] for determining who will come to a party. In this program, some people will come to the party for sure, whereas others only join when at least three of their friends are coming. Example 7 is the $DeAL$ version of this program. The idea is that with cntComing each person watches the number of their friends that are coming grow, and once that number reaches three, the person will then come to the party too. To count the number of friends, rather than the final count used in [9], we can use the mcount continuous count aggregate that enumerates all the integers until the actual maximum, i.e. it returns I, the actual maximum, representing the integer interval $[1, I]$.

Example 7: Who will come to the party?

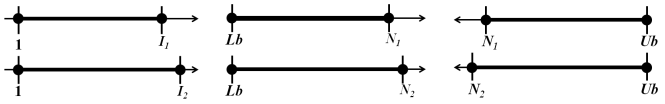
$$
\begin{aligned}
&r1.\ \texttt{coming}(\texttt{X}) &&\leftarrow \texttt{sure}(\texttt{X}).\\
&r2.\ \texttt{coming}(\texttt{X}) &&\leftarrow \texttt{cntComing}(\texttt{X}, \texttt{N}),\ \texttt{N} \geq \texttt{3}.\\
&r3.\ \texttt{cntComing}(\texttt{Y}, \texttt{mcount}\langle(\texttt{X}, \texttt{1})\rangle) &&\leftarrow \texttt{friend}(\texttt{Y}, \texttt{X}),\ \texttt{coming}(\texttt{X}).
\end{aligned}
$$

Here the use of mcount over count is justified on the grounds of performance, since it is inefficient to count all friends of people if only three are required. More importantly though, 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 techiques such as magic sets, perfect models and stable models can be immediately generalized to programs with mcount.

A. $DeAL$ Interval Semantics

The lessons learned with mcount tell 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 mmax and 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 used in Section II under general conditions that hold for all our examples. Due to space limitations the discussion is kept at an informal level: formal proofs are given in [17], [18].

Before we introduce $DeAL$'s interval semantics, consider the following example of interval semantics for counting, based on the Datalog^{FS} interval semantics, depicted in Fig. 17(a). If *r2* in Example 2 produced cpaths(a, b, 3) and then cpaths(a, b, 4), we cannot sum the aggregate values to get a new count for group (a, b). Instead, with the counts for cpaths(a, b, 3) and cpaths(a, b, 4) represented by $[1, 3]$



(a) Counting (b) +/- Rational Numbers (c) Minimum

Fig. 17. $DeAL$ Interval Semantics

and $[1, 4]$, respectively, $[1, 3] \cup [1, 4] = max(3, 4) = 4$. Thus $\texttt{cpaths}(\texttt{a}, \texttt{b}, 4)$ represents $(\texttt{a}, \texttt{b})$'s count.

+/- Rational Numbers. Computations on numbers in mantissa * exponent representation is *tantamount to integer arithmetic* on their numerators over a fixed large denominator. For instance, for floating point representations, the smallest value of exponent supported could be -95, whereby every number can be viewed as the integer numerator of a rational number with denominator 10^{95}. However, because of the limited length of the mantissa, this numerator will often be rounded-off — a monotonic operation that preserves the existence of a least-fixpoint semantics for our programs [17]. Thus floating-point-type representation in programming languages can be used without violating monotonicity [17], [18].

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

Minimum. Now, say 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 Ub, i.e. the interval $[N, Ub]$, as the number N. Observe that a number smaller than N is represented by an interval that contains the interval $[N, Ub]$. As before, if we take a union of two or more such representations, the result is the *largest interval*. Using Fig. 17(c) as an example, with N_1 and N_2 represented by $[N_1, Ub]$ and $[N_2, Ub]$, respectively, then $[N_1, Ub] \cup [N_2, Ub]$ represents the smaller of the two, i.e., $min(N_1, N_2) = N_2$.

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

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*[14]. 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 - 6 which are expressed using a stratified `max` (`min`) aggregate to select the max (min) value produced by the recursive rules when a fixpoint is reached. Examining the intermediate results of Class 1 programs: at each step of the fixpoint iteration, we have (i) the max (min) value and (ii) values less than the max value (greater than the min value). However, we do not need (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, rules are satisfied if and only if they are satisfied for the max (min) values. Example 7 is a Class 2 normal program.

C. Normal Program Evaluation

Recall the algorithm for *MASN* in Fig. 6. Let us call L the set produced by $T_E(M)$ or $T_R(\delta S)$ and F the set produced from applying $getLast()$ to L. For *Semi-naive*, δS will be L, whereas for *MASN*, δS will be F. Let $W = L - F$. $W = \varnothing$ when, for facts of monotonic aggregate predicates, each group with facts derived during the iteration has only one fact. For an iteration, if $W = \varnothing$, *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, and thus *MASN* does not need to derive facts with these either.

Theorem 10.1: 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 *EMSN* in Fig. 8. Assume we are evaluating a *normal* program with *Semi-naive*. Let us call K_{SN} the set of facts used in derivations ($\cup$ of all δS) 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 a derivation. Now, let $C = K_{SN} - K_{EMSN}$. If $C = \varnothing$, *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, facts in C are not going to lead to final answers. For Class 2 *normal* programs, any condition satisfied (in a rule body) by a fact in

[14]The compiler can easily check if a program is normal when the program contains only arithmetic and simple functions (e.g. addition, multiplication).

C would have also been satisfied by the value that was instead used. *EMSN* does not need to evaluate C. We have:

Theorem 10.2: A *normal* program with monotonic aggregates evaluated using *EMSN* will produce the same result as that program evaluated using *Semi-naive*.

XI. Additional Related Work

We have previously discussed the contributions of many works on supporting aggregation in recursion including [2], [9], [23]. For extrema aggregates, [25] proposes rewriting programs by pushing the aggregate into the recursion and *Greedy Fixpoint* evaluation to select the next min/max value to execute. Another approach proposes using *aggregate selections* to identify *irrelevant* facts that are to be discarded by an extended version of *Semi-naive* evaluation [26]. Nondeterministic constructs and stable models have been proposed to define monotone aggregates [27]. The DRed [30] algorithm incrementally maintains recursive views that can include negation and aggregation. The Bloom^L distributed programming language [16] uses logical monotonicity via built-in and user-defined lattice types to support eventual consistency.

XII. Conclusion and Future Work

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 [31]. 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. Lines of future work include i) supporting advanced KDD algorithms in $DeALS$, ii) parallel/distributed extensions of techniques from this paper and iii) extending $DeALS$ with highly-parallel evaluation techniques for recursive queries [32].

XIII. Acknowledgment

We thank Tyson Condie, Kai Zeng, Matteo Interlandi and Neng Lu for their discussions and suggestions on versions of this work. We thank the reviewers for their thoughtful comments. This work was supported by NSF awards IIS 1218471 and IIS 1302698.

References

[1] B. T. Loo, T. Condie, M. N. Garofalakis, D. E. Gay, J. M. Hellerstein, P. Maniatis, R. Ramakrishnan, T. Roscoe, and I. Stoica, "Declarative networking," *Commun. ACM*, vol. 52, no. 11, pp. 87–95, 2009.

[2] J. Seo, S. Guo, and M. S. Lam, "Socialite: Datalog extensions for efficient social network analysis," in *ICDE*, 2013, pp. 278–289.

[3] E. F. Codd, "Relational completeness of data base sublanguages," *In: R. Rustin (ed.): Database Systems: 65-98, Prentice Hall and IBM Research Report RJ 987, San Jose, California*, 1972.

[4] S. Greco and C. Zaniolo, "Greedy algorithms in datalog," *TPLP*, vol. 1, no. 4, pp. 381–407, 2001.

[5] P. G. Kolaitis, "The expressive power of stratified logic programs," *Info. and Computation*, pp. 50–66, 1991.

[6] M. P. Consens and A. O. Mendelzon, "Low complexity aggregation in graphlog and datalog," in *ICDT*, 1990, pp. 379–394.

[7] I. S. Mumick, H. Pirahesh, and R. Ramakrishnan, "The magic of duplicates and aggregates," in *VLDB*, 1990, pp. 264–277.

[8] I. S. Mumick and O. Shmueli, "How expressive is stratified aggregation?" *Annals of Mathematics and AI*, pp. 407–435, 1995.

[9] K. A. Ross and Y. Sagiv, "Monotonic aggregation in deductive databases," in *PODS*, 1992, pp. 114–126.

[10] C. Zaniolo, S. Ceri, C. Faloutsos, R. T. Snodgrass, V. S. Subrahmanian, and R. Zicari, *Advanced Database Systems*. Morgan Kaufmann, 1997.

[11] C. Zaniolo, N. Arni, and K. Ong, "Negation and aggregates in recursive rules: the ldl++ approach," in *DOOD*, 1993, pp. 204–221.

[12] G. Lausen, B. Ludäscher, and W. May, "On active deductive databases: The statelog approach," in *Transactions and Change in Logic Databases*, 1998, pp. 69–106.

[13] A. V. Gelder, "The well-founded semantics of aggregation," in *PODS*, 1992, pp. 127–138.

[14] W. Faber, G. Pfeifer, and N. Leone, "Semantics and complexity of recursive aggregates in answer set programming," *Artif. Intell.*, vol. 175, no. 1, pp. 278–298, 2011.

[15] A. Van Gelder, "Foundations of aggregation in deductive databases," in *DOOD*, 1993, pp. 13–34.

[16] N. Conway, W. R. Marczak, P. Alvaro, J. M. Hellerstein, and D. Maier, "Logic and lattices for distributed programming," in *SoCC*, 2012.

[17] M. Mazuran, E. Serra, and C. Zaniolo, "A declarative extension of horn clauses, and its significance for datalog and its applications," *TPLP*, vol. 13, no. 4-5, pp. 609–623, 2013.

[18] ——, "Extending the power of datalog recursion," *VLDB J.*, vol. 22, no. 4, pp. 471–493, 2013.

[19] D. Chimenti, R. Gamboa, R. Krishnamurthy, S. A. Naqvi, S. Tsur, and C. Zaniolo, "The ldl system prototype," *IEEE Trans. Knowl. Data Eng.*, vol. 2, no. 1, pp. 76–90, 1990.

[20] F. Arni, K. Ong, S. Tsur, H. Wang, and C. Zaniolo, "The deductive database system ldl++," *TPLP*, vol. 3, no. 1, pp. 61–94, 2003.

[21] J. M. Hellerstein, "The declarative imperative: experiences and conjectures in distributed logic," *SIGMOD Record*, vol. 39, no. 1, pp. 5–19, 2010.

[22] F. Giannotti, G. Manco, and F. Turini, "Specifying mining algorithms with iterative user-defined aggregates," *IEEE Trans. Knowl. Data Eng.*, vol. 16, no. 10, pp. 1232–1246, 2004.

[23] W. Faber, G. Pfeifer, N. Leone, T. Dell'Armi, and G. Ielpa, "Design and implementation of aggregate functions in the dlv system," *TPLP*, vol. 8, no. 5-6, pp. 545–580, 2008.

[24] T. J. Green, M. Aref, and G. Karvounarakis, "Logicblox, platform and language: A tutorial," in *Datalog 2.0*, 2012, pp. 1–8.

[25] S. Ganguly, S. Greco, and C. Zaniolo, "Minimum and maximum predicates in logic programming," in *PODS*, 1991, pp. 154–163.

[26] S. Sudarshan and R. Ramakrishnan, "Aggregation and relevance in deductive databases," in *VLDB*, 1991, pp. 501–511.

[27] C. Zaniolo and H. Wang, "Logic-based user-defined aggregates for the next generation of database systems," in *The Logic Programming Paradigm*, K. R. Apt, V. Marek, M. Truszczynski, and D. S. Warren, Eds. Springer Verlag, 1999, pp. 401–426.

[28] S. Ganguly, R. Krishnamurthy, and A. Silberschatz, "An analysis technique for transitive closure algorithms: A statistical approach," in *ICDE*, 1991, pp. 728–735.

[29] A. Shkapsky, K. Zeng, and C. Zaniolo, "Graph queries in a next-generation datalog system," *PVLDB*, vol. 6, no. 12, pp. 1258–1261, 2013.

[30] A. Gupta, I. S. Mumick, and V. S. Subrahmanian, "Maintaining views incrementally," in *SIGMOD*, 1993, pp. 157–166.

[31] P. Alvaro, N. Conway, J. Hellerstein, and W. R. Marczak, "Consistency analysis in bloom: a calm and collected approach," in *CIDR*, 2011, pp. 249–260.

[32] M. Yang and C. Zaniolo, "Main memory evaluation of recursive queries on multicore machines," in *IEEE Big Data*, 2014, pp. 251–260.