Aggregation in Datalog Under Set Semantics

Abhijeet Mohapatra and Michael Genesereth

Stanford University
{abhijeet, genesereth}@stanford.edu

Abstract. We propose an extension of Datalog that supports aggregates under *set semantics* and is as expressive as the previous extensions of Datalog which use *bag semantics* to represent aggregate programs. We show that under set semantics, our extension has greater expressive power than the previous extensions of Datalog which support aggregates. By expressing aggregate Datalog queries under set semantics, we can study the equivalence of the aggregate queries without defining notions of equivalence under set, bag-set and bag semantics separately. Under set semantics, Datalog programs with aggregation can be evaluated efficiently since the different derivations of goal tuples are not tracked. In our proposed extension, complex aggregation predicates can be built modularly using simple predicates. This modularity can be potentially leveraged to rewrite aggregate queries using aggregate views where the aggregation functions used in the query and the views are different.

1 Introduction

The problem of representing and reasoning about Datalog programs with aggregates has received considerable interest in the database as well the logic community. Different extensions of Datalog [1–21] have been proposed to support aggregate rules. A common theme underlying the proposed extensions is that they operate under *bag semantics*. However, it is unclear as to why bag semantics is required, if at all, for representing aggregate programs and for computing their models. The evaluation of the Datalog programs under bag semantics presents the following problems. First, every derivation of a goal or a sub-goal is tracked. This is inefficient, since we cannot leverage efficient techniques such as pipelined semi-naive evaluation (PSN) [23] to evaluate the Datalog programs. Second, in order to decide the equivalence of Datalog queries with aggregation, we have to separately study the equivalence under the set, bag-set and bag semantics [5]. Previous work in [5] presents examples of aggregate queries which are equivalent under bag semantics but not equivalent under set semantics.

We propose an extension of Datalog that supports aggregates under *set semantics*. We show our extension is as expressive as the previous extensions of Datalog which use *bag semantics* to represent the aggregate programs. Additionally, we show that under set semantics, our extension has greater expressive power than the previous extensions of Datalog which support aggregates. The drawback of representing sets in Datalog is the cost of maintaining a set by removing duplicates. Removing duplicates from a set of n elements requires O($n \times \log n$) time. However, if we use sets in Datalog to evaluate aggregate programs, we can leverage efficient evaluation techniques such as [23].

Additionally, by expressing aggregate queries in Datalog under set semantics we can study the equivalence of the aggregate queries without having to study the equivalence under bag-set and bag semantics separately [5].

In our proposal (which is presented in Section 2), we introduce *Datalog sets* as first class citizens in Datalog. We support aggregation by defining predicates over the Datalog sets of the aggregated variables. By segregating the construction of the Datalog sets of aggregate variables and the specification of aggregation predicates (such as sum, count or average) over Datalog sets, we allow complex aggregation predicates to be defined modularly using simple predicates. For example, the aggregation predicate average(S, A) which computes the average A of the Datalog set S can be defined using the predicates sum(S, A) and count(S, A) using the rule:

$$\text{average}(S, A) \text{ :- } \text{sum}(S, A_1),\ \text{count}(S, A_2),\ A = A_1/A_2$$

We can potentially leverage the above modularity to rewrite the aggregate queries using the aggregate views where the aggregation functions used in the query and the views are different using the rewriting techniques such as [28].

Our contributions in this paper can be summarized as follows:

1. In Section 2 we propose an extension of Datalog to represent aggregate rules by constructing Datalog sets using the 'setof' operator.
2. In Section 3 we show that complex aggregation predicates can be modularly specified using simpler aggregation predicates. We also show how this modularity can be potentially leveraged using techniques such as [28] to rewrite the aggregate queries using the aggregate views.
3. In Section 4 we show that our extension has the following properties:
   a. Our extension of Datalog achieves the same expressive power under *set semantics* as the previous extensions of Datalog [1–7] which operate under *bag semantics* to represent the aggregates (Theorem 1).
   b. Our extension of Datalog has greater expressive power than the previous extensions of Datalog [1–7] under set semantics (Lemma 1).
   c. We lose expressive power in our language when we remove the don't care variables (which are defined in Section 2) from our 'setof' operator (Theorem 2).

We present the advantages of computing aggregates under set semantics in Section 5 and discuss the related work on extending Datalog with aggregates in Section 6.

2 Datalog Sets

In this section, we propose our extension of Datalog and show that it supports aggregation under set semantics. As a first step, we introduce the construction of sets in Datalog. Subsequently, we show how aggregation predicates such as count or sum can be defined over the sets of aggregated variables.

In our extension of Datalog, we introduce *Datalog sets* as first class citizens. We define a *Datalog set* to be a set of standard Datalog terms or tuples. Examples of Datalog sets are $\{1, 2, 3\}$, $\{(1, a), (2, b), (1, c)\}$ and $\{(X,\ Y) \mid \exists Z.\, p(Z, Y),\, r(X, Z, W)\}$.

We introduce the operators *memberOf* and '|' (concatenation operator) to manipulate Datalog sets. The memberOf operator has the signature: memberOf(X, S), where X is a Datalog term and S is a Datalog set. The expression memberOf(X, S) is true *iff* X belongs to the set S. The concatenation operator '|' is used to add new elements to a Datalog set. Suppose we have a predicate p which requires only one argument of the type Datalog set. Suppose that $p(\{1, 2, 3\})$ is true. We can add a new element (say 4) to the Datalog sets in p using the rule: $p(4 \mid S)$:- $p(S)$, $\neg$memberOf$(4, S)$. The left-hand side argument to the '|' operator is always a Datalog term or a tuple and the right-hand side argument is always a Datalog set. We can use the memberOf and '|' operators to compute aggregates (such as sum or count) over a Datalog set. Using Example 1 below, we show how we can compute the sum of the elements of a Datalog set.

Example 1. The following rules inductively define the rules to compute the sum of a Datalog set.

$$\begin{aligned} \text{sum}(\{\}, 0) \\ \text{sum}(X \mid S, A) \text{ :- } \text{sum}(S, A_1),\ A = A_1 + X \end{aligned}$$

The first rule specifies the base case of the induction i.e. the sum of an empty set is 0. The second rule defines the goal sum$(X \mid S,\ A)$ inductively using the sub-goal sum(S, A_1).

A Datalog set can either be empty $\{\}$ or it can be constructed using the *setof* operator which we define below.

Definition 1. *The signature of the setof operator is: setof(A, D, SG, S), where A is the set of the aggregated variables, D is the set of the don't care variables, SG is the sub-goal which is aggregated on, and S is a Datalog set over the set of variables A. The set of variables A are D are disjoint.*

The semantics of the setof operator is as follows. Suppose V is the set of the variables in the sub-goal SG. For every binding of the variables in the set $V - (A \cup D)$ in the sub-goal SG, the setof operator computes the Datalog set $S = \{A\ \cup\ D \mid SG\}$. In the Datalog set S, the values of the don't care variables (D) are ignored and only the values of the aggregated variables A can be manipulated. We demonstrate the use of the setof operator and the specification of the don't variables using the following example.

Example 2. Consider a relation schema $p(X, Y, Z)$ with the following relation instance:

$$\begin{aligned} &p(x_1, y_1, 1) \\ &p(x_1, y_2, 1) \\ &p(x_2, y_3, 3) \end{aligned}$$

Suppose the predicate count(S, A) computes the cardinality A of the Datalog set S. Suppose we evaluate the following query over the relation instance p.

$$q(X, A) \text{ :- } \text{setof}(\{Z\}, \{Y\}, p(X, Y, Z), S),\ \text{count}(S, A)$$

The evaluation of q results in the derivation of the tuples: $q(x_1, 2)$ and $q(x_2, 1)$. The setof expression first computes $S = \{(Z,\ Y) \mid p(X, Y, Z)\}$ for every binding of the variable X. For $X = x_1$, the set $S = \{(1, y_1), (1, y_2)\}$ and for $X = x_2$, the set $S = \{(3, y_3)\}$. For $X = x_1$, the cardinality of S is 2. Hence the tuple $q(x_1, 2)$ is derived.

Instead of count(S, A) if we use sum(S, A) in the above rule, we would obtain the tuples: $q(x_1, 2)$ and $q(x_2, 3)$. While the same set $S = \{(1, y_1), (1, y_2)\}$ is derived for $X = x_1$, the values y_1 and y_2 are not passed to the predicate 'sum' since the variable Y is specified as a don't care variable.

By specifying don't care variables in a setof expression, we can mimic the computation of bags under set semantics. For example, we effectively construct the bag $\{1, 1\}$ (in Example 2) by computing the set $\{(1, y_1), (1, y_2)\}$ and specifying Y as a don't care variable. We can also pass *existentially quantified* sub-goals to the setof operator. The existentially quantified variables, however, are disjoint from the don't care variables. We provide an example below.

Example 3. Consider a variation of the query q presented in Example 2.

$$q(X, A) \text{ :- setof}(\{Z\}, \{\}, \exists Y.\, p(X, Y, Z), S),\ \text{count}(S, A)$$

In the above rule, the variable Y is existentially quantified for the sub-goal p. Hence, the set $S = \{1\}$ is computed for the variable binding $X = x_1$, resulting in the derivation of the tuple $(x_1, 1)$ instead of $(x_1, 2)$ as in Example 2.

So far, we have demonstrated how we can specify the don't care variables and existentially quantified sub-goals in setof expressions. In the following example, we show that we can also specify variables that are existentially quantified *outside* the setof expression.

Example 4. Consider a relation schema $r(X, Y, Z, W, V)$ with the following relation instance:

$$\begin{gathered} r(x_1, y_1, z_1, 3, v_1) \\ r(x_1, y_2, z_2, 3, v_1) \\ r(x_1, y_3, z_2, 3, v_1) \\ r(x_2, y_1, z_2, 5, v_1) \\ r(x_2, y_1, z_2, 5, v_2) \end{gathered}$$

The query $q(X, A)$:- setof($\{W\}, \{Z\}, \exists Y.\, r(X, Y, Z, W, V), S$), sum($S, A$) when evaluated on the relation instance r, results in the derivation of the tuples: $q(x_1, 6)$ and $q(x_2, 5)$. Since the variable Y is existentially quantified, setof operator computes the set $S = \{(3, z_1), (3, z_2)\}$ for the variable binding $X = x_1$ and $V = v_1$. The variable Z is a don't care variable. Hence for $X = x_1$, the predicate 'sum' computes the sum 3 + 3 (= 6). For the variable binding $X = x_2$, the set $\{(5, z_2)\}$ is computed twice for $V = v_1$ and $V = v_2$. The variable V serves as an existentially quantified variable *outside* the setof expression. Although we compute the set S for every binding of the variables X and V, only the variable X is passed to the head of the rule. Hence, the tuple $q(x_2, 5)$ is derived only once.

We could have also expressed the Datalog rule for q by first computing an intermediate relation $q'(X, Y, A)$ where,

$$q'(X, Y, A) \text{ :- setof}(\{W\}, \{Z\}, \exists Y. \, r(X, Y, Z, W, V), S), \text{ sum}(S, A)$$

and then projecting out the variable Y. The variables which are projected out can be viewed as being existentially quantified outside the setof operator.

3 Aggregation over Datalog sets

In our extension of Datalog, we support aggregation by expressing predicates over the Datalog sets. In Example 4, we used sum(S, A) as an aggregation predicate. Since Datalog sets are first class citizens in our extension, we can pass them as variables to aggregation predicates such as sum and count. Hence in our extension, aggregate predicates (such as average) can be expressed modularly using different aggregate predicates (such as sum and count). Suppose we are given the predicates sum(S, A) and count(S, A), we can express average(S, A) as follows:

$$\text{average}(S, A) \text{ :- sum}(S, A_1), \text{ count}(S, A_2), \ A = A_1/A_2$$

In our extension of Datalog to support aggregates, we treat aggregates as predicates over the Datalog sets. This has two advantages. First, we can build complex aggregate predicates modularly using simple aggregate predicates. Second, we can leverage the rules that relate different aggregate predicates (such as average, sum and count) to rewrite aggregate queries using aggregate views where the aggregates in the query and views are different. Consider the following example where we have two sets of numbers, S_1 and S_2. Suppose we have two views $v_1(S_1, S_2, E)$ and $v_2(S_1, S_2, L)$ which treat the sets S_1 and S_2 as strings (after sorting the elements in each set) and compute the edit distance and the LCS between the two strings respectively. We provide an example below to show that a variant of the Inverse Method which is proposed in [28] can be leveraged to rewrite q using only the views v_1 and v_2.

Example 5. Consider two sets of numbers $S_1 = \{1, 2, 3\}$ and $S_2 = \{2, 3, 4\}$. The views $v_1(S_1, S_2, E)$ and $v_2(S_1, S_2, L)$ compute the edit distance and the LCS over the string representations of S_1 and S_2. Therefore, the derivations $v_1(\{1, 2, 3\}, \{2, 3, 4\}, 2)$ and $v_2(\{1, 2, 3\}, \{2, 3, 4\}, 2)$ are true. Suppose the query $q(J)$ computes the Jaccard distance between the sets S_1 and S_2. The rules for the views v_1, v_2 and the query q are given below.

$$\begin{aligned} v_1(S_1, S_2, E) &\text{ :- editDistance}(S_1, S_2, E) \\ v_2(S_1, S_2, L) &\text{ :- lcs}(S_1, S_2, L) \\ q(J) &\text{ :- jaccardSim}(S_1, S_2, J) \end{aligned}$$

Suppose we specify jaacardSim modularly using the predicates editDistance and lcs.

$$\text{jaccardSim}(S_1, S_2, J) \text{ :- editDistance}(S_1, S_2, E), \text{ lcs}(S_1, S_2, L), \ J = E/(E + L)$$

We can now rewrite the query $q(J)$ in the following steps:

1. We unfold (or expand) the definition of the query q to get the rule:

$$q(J) \text{ :- editDistance}(S_1, S_2, E),\ \text{lcs}(S_1, S_2, L),\ J = E/(E + L)$$

2. We invert the definitions of the views v_1 and v_2 using the techniques presented in [28].

$$\text{editDistance}(S_1, S_2, E) \text{ :- } v_1(S_1, S_2, E)$$
$$\text{lcs}(S_1, S_2, L) \text{ :- } v_2(S_1, S_2, L)$$

   When we add the facts $v_1(\{1, 2, 3\}, \{2, 3, 4\}, 2)$ and $v_2(\{1, 2, 3\}, \{2, 3, 4\}, 2)$ to the above set of inverted rules and the unfolded query q, we obtain the goal tuple $q(0.5)$ which is the Jaccard similarity of the sets S_1 and S_2.

4 Expressive power

In the previous section, we proposed our extension of Datalog to support aggregates by defining Datalog sets using the setof operator. In this section, we compare the expressive power of our extension with respect to the previous extensions of Datalog which support aggregates under bag semantics [1–7]. First, we formally prove (using Theorem 1) that the expressive power of our extension under set semantics is the same as the expressive power of previous extensions under bag semantics [1–7]. This shows that set semantics suffices to represent aggregate programs in Datalog. We then show (using Lemma 1) that under set semantics, the expressive power of our extension is greater than the previous extensions [1–7]. Additionally, we show (using Theorem 2) that we lose expressive power in our language when we adopt a weaker variant of the setof operator.

Theorem 1. *An aggregate rule which is written using the previous extensions of Datalog [1–7] can be equivalently represented in our extension.*

Proof. Let us consider the extension of Datalog proposed in [1] to support aggregates under *bag semantics*. The proposal in [1] represents the aggregate sub-goals by constructing sets[1] or bags using the 'group_by' operator which has the following signature:

$$\text{group\_by}(\langle\text{sub-goal}\rangle, \langle\text{groupingVars.}\rangle, \langle\text{headVar = agg(expression)}\rangle)$$

For example, consider a predicate p with the schema $p(X, Y, Z, W)$. The Datalog rule

$$q(X, Y, A) \text{ :- group\_by}(p(X, Y, Z, W),\ [X, Y],\ A = \text{sum}(W))$$

partitions the predicate p into equivalence classes based on different variable bindings of the grouping variables X and Y and for each equivalence class, computes the sum of the bag W as the head variable A. The extensions of Datalog presented in [2–5] represent aggregates by specifying the set of grouping variables and the aggregate functions. This is equivalent to the representation of aggregates that is proposed in [1]. The extensions

[1] Sets can be passed by specifying agg(distinct(expression)) in [1].

presented in [6, 7] represent aggregation in answer-set programs by applying aggregate functions over 'symbolic sets'. A symbolic set implicitly assigns the variables which are bound outside the definition of the symbolic set to be the grouping variables for the aggregate function. For example consider the following aggregate rule presented in [6]

$$ctrls(X, Y) \text{ :- } comp(X),\ comp(Y),\ \#\text{sum}\{S,\ Z\ :\ cs(X, Y, Z, S)\} > 50$$

The grouping variables for the aggregate expression $\#\text{sum}\{S,\ Z\ :\ cs(X, Y, Z, S)\}$ are X and Y which are bound outside the symbolic set. Hence the aggregate expressions over the symbolic sets [6, 7] can also be obtained using the representation scheme which is proposed in [1].

For the purpose of this proof we only consider the extension of Datalog which is presented in [1] and show that any 'group_by' expression has an equivalent representation using our proposed extension of Datalog. Consider a general group_by sub-goal:

$$\text{group\_by}(\phi(S_1, S_2, S_3),\ [S_1],\ A = agg(S_3))$$

In the above expression, ϕ is a sub-goal over *sets* of variables S_1, S_2 and S_3. The set S_1 is the set of the grouping variables, and the aggregate function agg is computed over the set of variables in S_3. The above group_by sub-goal is equivalent to:

$$\text{setof}(S_3, S_2, \phi(S_1, S_2, S_3), S),\ agg(S, A)$$

The setof operator first constructs a Datalog set S of the variable bindings of the set of variables $S_3 \cup S_2$ where the set S_2 represents the don't care variables. The bindings of the variable S_3 in S are passed to the aggregate predicate agg. Instead, if the group_by sub-goal computed the aggregate agg over *distinct* variable bindings of the variables in S_3, we would represent the group_by sub-goal as:

$$\text{setof}(S_3, \{\}, \exists S_2.\, \phi(S_1, S_2, S_3), S),\ agg(S, A)$$

The setof operator in the above expression first computes the Datalog set $S = \{S_3 \mid \exists S_2.\, \phi(S_1, S_2, S_3)\}$. Hence, every group_by sub-goal which can be represented under *bag semantics* can be equivalently represented by sub-goals in our extension of Datalog under *set semantics*. $\square$

Theorem 1 proves that set semantics suffice to represent aggregates in Datalog. We now prove that under set semantics the previous extensions of Datalog [1–7] cannot represent some aggregate rules which can be written in our extension.

Lemma 1. *Some aggregate sub-goals which can be written in our extension of Datalog cannot be equivalently represented using the 'group_by' sub-goals [1] under set semantics.*

Proof. For proving the above lemma, we use the relation instance $r(X, Y, Z, W, V)$ that is presented in Example 4. Consider the following aggregate rule.

$$q(X, A) \text{ :- } \text{setof}(\{W\}, \{Z\}, \exists Y.\, r(X, Y, Z, W, V), S),\ \text{sum}(S, A)$$

The query q when evaluated over the relation instance r yields the goal tuples: $q(x_1, 6)$ and $q(x_2, 5)$. We claim that the above aggregate rule *cannot* be represented using the group_by operator [1] under set semantics. The proof of the claim is quite obvious since the derivation of the goal tuple $q(x_1, 6)$ using the group_by operator requires the construction of the bag $\{3, 3\}$ which is not possible under set semantics. □

Lemma 1 shows that when we operate under set semantics, our extension of Datalog for representing aggregates is more expressive that the extensions presented in [1–7]. In the Example 2 we remarked that the don't care variables allow us to mimic the construction of bags under set semantics. We show that by dropping the don't care variables we lose expressive power in our language.

Theorem 2. *Every Datalog set which can be generated using the 'setof' operator cannot be generated using a variant of the 'setof' operator where the don't care variables are not specified.*

Proof. Consider the relation instance r and the query q presented previously in the Example 4 and the Lemma 1. Suppose we use a variant of the 'setof' operator in our extension of Datalog without specifying the don't care variables. Suppose we want to represent the query q over the relation instance r using the modified setof operator. Since we only need to specify the existentially quantified variables from the set of variables[2] $\{Y, Z, V\}$. There are $2^3 = 8$ possibilities, each of which corresponds to existentially quantifying a subset of $\{Y, Z, V\}$. Neither of these 8 possibilities result in the derivation of the goal tuple $q(x_1, 6)$. Hence, the query q cannot be represented using the modified setof operator where the don't care variables are not specified. □

5 Advantages of Aggregating under Set Semantics

Previously, in Sections 2 and 4, we proposed our extension of Datalog to support aggregates and analyzed its expressive power with respect to previous extensions of Datalog [1–7]. Our proposal has the following three distinct features.

a. We evaluate aggregates under set semantics.
b. We represent the aggregation in the body of a rule.
c. We separately specify the Datalog sets using the 'setof' operator and the aggregation predicates over the aggregation predicates.

In this section, we discuss the advantages of using the above design choices for supporting aggregates in Datalog.
Set semantics are sufficient: A common theme across all previous extensions of Datalog is to define aggregates under bag semantics. In our proposal, we evaluate the aggregates under set semantics without ever constructing bags in the intermediate steps. By specifying the don't care variables in our 'setof' operator we can mimic the construction of bags. Let us revisit the Example 4. For the variable binding $X = x_1$ and $V = v_1$ we were able to compute the sum $3 + 3$ by generating the Datalog sets of the

[2] The variable X cannot be existentially quantified since it is passed to the head of the rule.

W and Z variables and specifying the Z component of the Datalog set as don't care. Although, the Datalog set $S = \{(3, z_1), (3, z_2)\}$, only the W values are passed to the sum predicate. Previous approaches [1–21] which use bag semantics would generate the bag $\{3, 3\}$ either explicitly or implicitly in an intermediary step before computing the sum. In Section 4 we formally proved that the expressive power of our extension under set semantics is the same as that of the previous extensions of Datalog [1–7] to support aggregates under bag semantics. We also show that under set semantics, the expressive power of our extension is greater than the previous extensions of Datalog [1–7] to support aggregates. There are two immediate consequences of these results. First, we can restrict ourselves to set semantics only to evaluate Datalog programs with aggregation. Therefore we can leverage efficient evaluation strategies such as pipelined semi-naive evaluation [23] (for parallel environments), since the different derivations of goal tuples are not tracked. Second, this provides us a clean framework to study the equivalence of aggregation queries under set semantics only without defining notions of equivalence under bag and bag-set semantics separately [5].
Efficient evaluation of monotonic programs: Previous work in [2] discusses the construction of iterative minimal models for aggregate-monotonic programs under the assumption that the aggregated values are functionally dependent on the grouping variables. Aggregate-monotonic programs (which are defined in [1]) are the Datalog programs with aggregates where the derivation of a head tuple cannot be invalidated by the addition of the new base tuples. A necessary condition for a Datalog program to be aggregate-monotonic is that the aggregated value should not be passed to the head of a rule [2]. We borrow the 'Company control' example from [2] to give an example of an aggregate-monotonic program.

Example 6. Consider a relation $s(X, Y, N)$ where company X owns a proportion N ($0 \leq N \leq 1$) of all the shares in the company Y. A company X is said to control another company Y i.e. $c(X, Y)$ is true, if the sum of the shares it owns in Y together with the sum of the shares owned by the companies controlled by X is greater than half of the total number of the shares in Y. We use a relation $cv(X, Z, Y, N)$ to express the fact that company X controls N percent of the shares in Y via an intermediate company Z. The 'Company Controls' program [2] is given below:

$$\begin{aligned} cv(X, X, Y, N) &\text{ :- } s(X, Y, N) \\ cv(X, Z, Y, N) &\text{ :- } c(X, Z),\ s(Z, Y, N) \\ c(X, Y) &\text{ :- } N = \text{sum } M \ :\ cv(X, Z, Y, M),\ N > 0.5 \end{aligned}$$

The extensions of Datalog which support aggregates in the head of a rule [3–5] pass the aggregate value to the head of a rule. Hence the aggregate-monotonic programs cannot be represented in such extensions without destroying the monotonicity property. However, if we represent aggregates in the body of a rule, we can potentially leverage previous techniques (such as [2] and [23]) to evaluate aggregate-monotonic programs more efficiently than the standard semi-naive evaluation strategy.
Rewriting aggregate queries using aggregate views: The problem of rewriting queries using views has been well studied in the database community. A comprehensive survey of the techniques which are used to rewrite conjunctive queries using views is presented

in [24]. While the problem of rewriting conjunctive queries using views has received considerable attention [24–28], a small fraction of the work addresses the case where the language is extended with aggregation [4, 5, 29–31]. A common theme underlying the techniques proposed in [4, 5, 29–31] is that the aggregate functions in the query and the views are typically the same.

For example, it is easy to rewrite a query that computes the count of the elements in a set in terms of a view that compute the sum of the sum of the elements in a set. The facts such as the average of a set is a function of the sum and the count of the set and the count of a set can be obtained by summing the number 1 for every element in the set are common knowledge and can be hard-coded into the rewriting algorithms. However, consider Example 5 which is presented in Section 3. To rewrite the query q using only the views v_1 and v_2, we require prior knowledge of how the metrics Jaccard similarity, edit distance and LCS are related. We can leverage this knowledge by defining Jaccard similarity modularly using edit distance and LCS as in Example 5. Otherwise, rewriting the query q is impossible without explicitly computing the Jaccard similarity over the sets S_1 and S_2.

6 Related Work

Previous treatments of Datalog and the proposed extensions to compute queries with aggregation can be categorized under the following two tracks.

1. **Aggregation by indirectly manipulating sets or bags** ([1–5]): Mumick et al [1] provide the minimal model semantics for monotonic and magically stratified Datalog programs with aggregation. An aggregate rule (as defined by [1]) requires the use of a special operator *group_by* which applies aggregate functions on the bags derived from the input sub-goals. The syntax and the semantics of the group_by operator is discussed in detail in Section 4. The techniques proposed in [2] discuss the construction of iterative minimal models for aggregate-monotonic programs under the assumption that the aggregated values are functionally dependent on the grouping variables. Previous work [3] has studied the expressive power of the first-order logic when stratified, monotonic or unstratified aggregates are added to the language. The techniques proposed in [4, 5] extend the standard Datalog framework to incorporate aggregate functions in the head of a rule. The evaluation of an aggregate rule is carried out in two steps. In the first step, the aggregation query retrieves a multiset of the tuples from the instance. The tuples are then grouped into equivalence classes on the basis of the grouping variables and the aggregation function is subsequently applied to each equivalence class.
2. **Aggregation by explicitly constructing sets or bags** ([6–21]): The extensions proposed in [12–20] add constructors for explicitly creating sets and bags. An excellent overview of the four different Datalog extensions (LDL, COL, Hilog and RelationLog) which explicitly allow the definition of sets and bags is presented in [21]. Previous work in [6, 7] introduced the concept of 'symbolic sets' (e.g. $\{X \mid \exists Y.Z.\, p(X, Y), r(Y, Z)\}$). The aggregate atoms in [6, 7] are specified by applying the aggregate functions over the symbolic sets. The syntax of aggregation

over the symbolic sets is briefly discussed in Section 4. In order to evaluate programs with aggregate atoms, [6, 7] introduce the answer-set semantics for the disjunctive Datalog programs. The semantics proposed in [6, 7] is a natural extension of the Gelfond-Lifschitz transformation. A comprehensive survey on the deductive database systems with set-oriented evaluation strategies is presented in [22]. The techniques proposed in [8–11] provide the semantics of aggregation in Datalog programs where the sets and bags are created explicitly and where the variables ranging over the predicate names are quantified.

The previous extensions of Datalog [1–21] that support aggregation, operate under bag semantics. In contrast, we propose an extension of Datalog to support aggregates under set semantics. We have shown (in Section 4) that the aggregates rules which can be expressed using the previous extensions of Datalog [1–7] under bag semantics can also be equivalently represented using our extension of Datalog. This result has two implications. First, we can evaluate Datalog programs with aggregates more efficiently [23] under set semantics as compared to bag semantics since we do not need to keep track of the different derivations of a goal tuple. Second, we can study the equivalence of the aggregate queries under set semantics only without defining additional notions of equivalence under bag-set and bag semantics. Our proposed extension of Datalog has the following three distinct features.

a. We evaluate aggregates under set semantics.
b. We represent the aggregation in the body of a rule. This is in contrast with the extensions proposed in [3–5] that support aggregation in the head of a rule.
c. We separately specify the Datalog sets using the 'setof' operator and the aggregation predicates over the aggregation predicates in contrast to the techniques proposed in [1–5] which allow the set or bags to be explicitly constructed in the language.

7 Conclusion

In this paper, we have proposed an extension of Datalog which support aggregates under set semantics. We have established that our extension which uses set semantics is as expressive as the previous extensions of Datalog which use bag semantics to represent the aggregates. This result has two implications: (a) we can leverage efficient techniques such as [23] to evaluate aggregate programs (b) we can study the equivalence of aggregate queries under set semantics only without defining notions of equivalence under bag-set and bag semantics separately [5]. In our proposal we introduce Datalog sets as first class citizens. We support aggregation by defining predicates over the Datalog sets of aggregated variables. By separating the construction of Datalog sets and the specification of the aggregation predicate over a Datalog set we allow complex aggregation predicates to be specified modularly using simple predicates. We can potentially leverage this modularity to rewrite aggregate queries using aggregate views.

References

1. Mumick, I.S., Pirahesh, H., Ramakrishnan, R.: The magic of duplicates and aggregates. VLDB (1990)

2. Ross, K.A., Sagiv, Y.: Monotonic aggregation in deductive databases. PODS (1992)
3. Consens, M.P., Mendelzon, A.O.: Low complexity aggregation in graphlog and datalog. ICDT (1990)
4. Cohen, S., Nutt, W., Serebrenik, A.: Rewriting aggregate queries using views. PODS (1999)
5. Afrati, F., Chirkova, R.: Selecting and using views to compute aggregate queries. Journal Computer System Sciences (2011)
6. Alviano, M., Greco, G., Leone, N.: Dynamic magic sets for programs with monotone recursive aggregates. LPNMR (2011)
7. Faber, W., Pfeifer, G., Leone, N.: Semantics and complexity of recursive aggregates in answer set programming. Artificial Intelligence. (2011)
8. Abiteboul, S., Beeri, C.: The power of languages for the manipulation of complex values. The VLDB Journal (1995)
9. Tsur, S., Zaniolo, C.: Ldl: A logic-based data language. VLDB (1986)
10. Chen, W., Kifer, M., Warren, D.S.: Hilog: a foundation for higher-order logic programming. Journal of Logic Programming (1993)
11. Liu, M.: Relationlog: A typed extension to datalog with sets and tuples (extended abstract). Journal of Logic Programming (1995)
12. Bancilhon, F., Maier, D., Sagiv, Y., Ullman, J.D.: Magic sets and other strange ways to implement logic programs (extended abstract). PODS (1986)
13. Bancilhon, F., Ramakrishnan, R.: An amateur's introduction to recursive query processing strategies. SIGMOD (1986)
14. Beeri, C., Ramakrishnan, R.: On the power of magic. PODS (1987)
15. Ceri, S., Gottlob, G., Tanca, L.: Logic programming and databases. (1990)
16. Ioannidis, Y.E., Ramakrishnan, R.: Efficient transitive closure algorithms. VLDB (1988)
17. Jiang, B.: A suitable algorithm for computing partial transitive closures in databases. ICDE (1990)
18. Mumick, I.S., Finkelstein, S.J., Pirahesh, H., Ramakrishnan, R.: Magic conditions. ACM Transactions on Database Systems (TODS) (1996)
19. Sacca, D., Zaniolo, C.: Magic counting methods. SIGMOD (1987)
20. Ullman, J.D.: Principles of Database and Knowledge-Base Systems: Volume II. (1989)
21. Liu, M.: Overview of datalog extensions with tuples and sets. DDLP (1998)
22. Ramakrishnan, R., Ullman, J.D.: A survey of research in deductive database systems. Technical report (1995)
23. Loo, B.T., Condie, T., Garofalakis, M., Gay, D.E., Hellerstein, J.M., Maniatis, P., Ramakrishnan, R., Roscoe, T., Stoica, I.: Declarative networking: language, execution and optimization. SIGMOD (2006)
24. Halevy, A.Y.: Answering queries using views: A survey. The VLDB Journal (2001)
25. Afrati, F., Li, C., Ullman, J.D.: Generating efficient plans for queries using views. SIGMOD (2001)
26. Calvanese, D., De Giacomo, G., Lenzerini, M., Vardi, M.Y.: View-based query containment. PODS (2003)
27. Chirkova, R., Halevy, A.Y., Suciu, D.: A formal perspective on the view selection problem. The VLDB Journal (2002)
28. Duschka, O.M., Genesereth, M.R.: Answering recursive queries using views. PODS (1997)
29. Agrawal, S., Chaudhuri, S., Narasayya, V.R.: Automated selection of materialized views and indexes in sql databases. VLDB (2000)
30. Gupta, A., Harinarayan, V., Quass, D.: Aggregate-query processing in data warehousing environments. VLDB (1995)
31. Srivastava, D., Dar, S., Jagadish, H.V., Levy, A.Y.: Answering queries with aggregation using views. VLDB (1996)