Incremental Maintenance of Aggregate Views

Abhijeet Mohapatra and Michael Genesereth

Stanford University
{abhijeet, genesereth}@stanford.edu

Abstract. We propose an algorithm called *CReaM* to incrementally maintain materialized aggregate views with user-defined aggregates in response to changes to the database tables from which the view is derived. CReaM is *optimal* and guarantees the self-maintainability of aggregate views that are defined over a single database table. For aggregate views that are defined over multiple database tables and do not contain all of the non-aggregated attributes in the database tables, CReaM speeds up the time taken to update a view as compared to prior view maintenance techniques. The speed up in the time taken to update a materialized view with n tuples is either $\frac{n}{\log n}$ or $\log n$ depending on whether the physical design of the underlying database is optimized or not. For other types of aggregate views, CReaM updates the view in no more time than that is required by prior view maintenance techniques to update the view.

1 Introduction

In data management systems, views are derived relations that are computed over database tables (which are also known as extensional database relations or *edbs*). Views are materialized in a database to support efficient querying of the data. A materialized view becomes out-of-date when the underlying edb relations from which the view is derived are changed. In such cases, the materialized view is either recomputed from the edb relations or the changes in the edb relations are *incrementally* propagated to the view to ensure the correctness of the answers to queries against the view. Prior work presented in [5, 20, 26] shows that incrementally maintaining a materialized view can be significantly faster than recomputing the view from the edb relations especially if the size of the view is large compared to the size of the changes.

Several techniques [1–6, 10, 14–16, 18, 20–22, 25–27, 29–31, 33] have been proposed to incrementally maintain views in response to changes to the edb relations. However, only a small fraction of the prior work on incremental view maintenance [10, 14, 15, 20, 23, 26, 27] addresses the maintenance of views that contain aggregates such as sum and count. The techniques proposed in [10, 14, 20, 23] incrementally maintain views that have only one aggregation operator. Furthermore, the incremental maintenance algorithms presented in [10, 14, 15, 20, 23, 26, 27] support only a fixed set of built-in aggregate operators (min, max, sum, and count).

In contrast, we present an algorithm to incrementally maintain views with *multiple* aggregates each of which could be *user-defined*. As our underlying query language, we extend Datalog using tuples and sets, and express aggregates as predicates over sets. In section 2, we discuss the specification of user-defined aggregates in our language.

In section 3, we present differential rules to correctly characterize the changes in edb relations to aggregate views. In section 4, we present an algorithm *CReaM* to *optimize* the maintenance of a special class of materialized aggregate views that do not contain all of the non-aggregated attributes in the underlying edb relations. In section 5, we establish the optimality of CReaM and the theoretical results on the performance of CReaM which are summarized in table 1. In addition, we show that by materializing auxiliary views, CReaM guarantees the *self-maintainability* [17] of aggregate views that are defined over a single edb relation. This property is desirable when access to the edb relations is restricted or when the edb relations are hypothetical such as in a LAV integration scenario [32]. In section 6, we compare our work to prior work on incremental maintenance of aggregate views.

Before we discuss our proposed solution, we illustrate the problem of incrementally maintaining aggregate views using a running example. We use our running example, which is based on the Star Wars universe, in examples throughout the paper.

Speed up in the time taken to update a materialized aggregate view with n tuples as compared to prior techniques		
	Physical design of the database	
	Non-optimized	Optimized
Aggregate view over single edb relation	≥ 1	≥ 1
Aggregate view over multiple edb relations (single update)	$\log n$	$\frac{n}{\log n}$
Aggregate view over multiple edb relations ($k > 1$ updates)	$\log n$	$\frac{n}{k}$

Table 1. Performance summary of the CReaM algorithm

Running Example: Suppose that there are tournaments in the Star Wars universe on different planetary systems. The tournament results are recorded in an edb relation, say tournament(V, D, L). A tuple $(V, D, L) \in$ tournament *iff* V has defeated D on the planet L. For instance, if Yoda has defeated Emperor Palpatine at Dagobah then the tuple (yoda, palpatine, dagobah) is in the extension of the edb relation tournament. We use the extension of tournament that is presented in table 2 in examples throughout the paper.

tournament		
Victor	**Defeated**	**Location**
yoda	vader	dagobah
yoda	palpatine	dagobah
vader	yoda	tatooine
yoda	palpatine	tatooine

victories	
Victor	**Wins**
yoda	2
vader	1
yoda	1

Table 2. Extension of the edb relation *tournament* and the view *victories* in the Star Wars Universe

We define and materialize a view, say victories(V, W) to record the number of victories W achieved by a character V on a planet. For instance, Yoda has two victories in Dagobah and one victory in Tatooine. Therefore the tuples (yoda, 2) and (yoda, 1) $\in$ victories. The extension of victories that corresponds to the extension of the edb relation tournament is presented in table 2.

Suppose that a new tournament match is played in Tatooine and that Darth Vader defeats Emperor Palpatine in this match. The new tournament match at Tatooine causes an *insert* to the tournament relation. In response to the insert to the tournament relation, the tuple (vader, 1) $\in$ victories must be updated to (vader, 2) to ensure the correctness of the answers to queries against the view. Now, suppose that the previous tournament match between Yoda and Palpatine at Tatooine is invalidated. In this case, the tuple (yoda, palpatine, tatooine) is *deleted* from the tournament relation. In response to this deletion, the tuple (yoda, 1) must be deleted from the materialized view victories.

2 Preliminaries

We support aggregation in Datalog under set semantics by introducing tuples and *sets* as first-class citizens. A tuple is an ordered sequence of Datalog constants or sets. A set is either empty or contains Datalog constants or tuples. For example, the tuple (yoda, vader, dagobah) and the sets $\{\}$, $\{$(yoda, vader, dagobah)$\}$ and $\{$(yoda, $\{1, 2\}$)$\}$ are legal in our language. We introduce the *setof* operator in our language to represent sets as follows.

Definition 1. *Suppose $\phi(\bar{X}, \bar{Y})$ is a conjunction of subgoals. The setof subgoal setof($\bar{Y}$, $\phi(\bar{X}, \bar{Y})$, S) represents the set $S = \{\bar{Y} \mid \phi(\bar{X}, \bar{Y})\}$ for every binding of values in $\bar{X}$.*

We illustrate the construction of sets in our language using the following example.

Example 1. Consider the running example that we presented in section 1. Suppose we would like to compute the set of characters who were defeated by Yoda at Dagobah. In our language, we compute the desired set using the following query.

$$q_1(S) \text{ :- setof}(D, \text{tournament}(\text{yoda}, D, \text{dagobah}), S)$$

In query q_1, the set $S = \{D \mid \text{tournament}(\text{yoda}, D, \text{dagobah})\}$. The evaluation of the query q_1 on the extension of tournament that is presented in table 2 results in the answer tuple $q_1(\{\text{vader, palpatine}\})$.

Construction of multisets: In example 1, we computed the set of characters who were defeated by Yoda at Dagobah. In addition to generating sets, we could leverage the setof operator to effectively aggregate *multisets* as illustrated in the following example.

Example 2. Suppose we would like to count the total number of victories achieved by Yoda. Since Yoda defeats Emperor Palpatine at multiple locations, we would have to compute the cardinality of the multiset of people who were defeated by Yoda i.e. $\{$palpatine, vader, palpatine$\}$ to compute Yoda's total number of victories. Consider a query q_2 which is defined as follows.

$$q_2(C) \text{ :- setof}((D, L), \text{tournament}(\text{yoda}, D, L), S), \text{ count}(S, C)$$

We assume that the predicate count(X, Y) computes the cardinality Y of the set X. We discuss the representation of user-defined aggregates in our language shortly. In query q_2, the set $S = \{(D, L) \mid \text{tournament}(\text{yoda}, D, L)\}$. The evaluation of q_2 on the extension of tournament relation (table 2) computes the cardinality of the set $\{$(palpatine,

dagobah), (vader, dagobah), (palpatine, tatooine)} thus mimicking the computation of the cardinality of the multiset {palpatine, vader, palpatine}. As a result, the answer tuple $q_2(3)$ is generated.

We use the '|' operator in our language to represent the decompositions of a set. We represent the decomposition of a set S into an element $X \in S$ and the subset $S_1 = S \setminus \{X\}$ as $\{X \mid S_1\}$. For example, $\{3 \mid \{1, 2\}\}$ represents the decomposition of the set of numbers $\{1, 2, 3\}$ into 3 and the subset $\{1, 2\}$. We define the predicate *member* in our language to check the membership of an element in a set. The member predicate has the signature member(X, S), where X is a Datalog constant or a tuple and S is a set. If $X \in S$ then member(X, S) is true, otherwise it is false. The member predicate can be defined in our language using the decomposition operator '|' operator as follows.

$$\text{member}(X, \{X \mid Y\})$$
$$\text{member}(Z, \{X \mid Y\}) \text{ :- member}(Z, Y)$$

In addition, we use $\cup$ and $\setminus$ operators in our language to represent set-union and set-difference respectively. We note that we can define $\cup$ and $\setminus$ operators in our language using the member predicate and the decomposition operator '|' although we do not define them as such in this paper.

Aggregation over sets: In our language, user-defined aggregates are defined as predicates over sets. A user-defined aggregate could either be defined (a) in a stand-alone manner using the member predicate, the decomposition operator '|', the set-union $\cup$ and the set-difference $\setminus$ operators, and the arithmetic operators or (b) as a view over other aggregates. For instance, we can compute the cardinality of a set in our language by inductively defining an aggregate, say count(X, C), as follows.

$$\text{count}(\{\}, 0)$$
$$\text{count}(\{X \mid Y\}, C) \text{ :- count}(Y, C_1),\ C = C_1 + 1$$

The first rule specifies the base case of the induction i.e. the cardinality of an empty set is 0. The second rule decomposes a set S into an element X and the subset Y and computes the cardinality of S by leveraging the cardinality of Y. In addition, we can define an aggregate such as *average* modularly by leveraging the definitions of the aggregates count(X, C) and sum(X, S) as follows.

$$\text{average}(X, A) \text{ :- sum}(X, S),\ \text{count}(X, C),\ A = \frac{S}{C}$$

3 Maintenance of Aggregate Views

In the previous section, we discussed the specification of aggregates as predicates over sets in our language. Consider the running example that we presented in section 1. Suppose we would like to query the number of victories V achieved by a character W on a planet. We represent this query in our language as follows.

$$q(W, V) \text{ :- setof}(D, \text{tournament}(W, D, L), S),\ \text{count}(S, V)$$

For every binding of the variables W and L in the query q, V = cardinality of $\{D \mid \text{tournament}(W, D, L)\}$. In the Star Wars universe, this is equivalent to computing the number of victories V achieved by a character W on a planet. Since the answers to queries are computed under set semantics, the distinct numbers of victories are generated by the query q. To efficiently compute the answer to the query q, we can leverage the materialized view victories from our running example (in section 1). The materialized view victories(W, V) is defined as follows.

$$\text{victories}(W, V) \text{ :- setof}(D, \text{tournament}(W, D, L), S), \text{ count}(S, V)$$

When changes are made to the edb relation tournament, we must maintain the materialized view victories to ensure the correctness of answers to the query q. In this section, we discuss the maintenance of materialized aggregate views in response to changes to the underlying edb relations. As a first step, we leverage differential relational calculus to incrementally propagate the changes in the edb relations to the materialized views through *differential rules*. Subsequently in section 4, we propose an algorithm called CReaM that leverages differential rules to optimally maintain materialized aggregate views.

Differential Rules: In differential relational calculus, a database is represented as a set of edb relations and views $r_1, r_2, \ldots, r_k$ with arities $d_1, d_2, \ldots, d_k$. Each relation r_i is a set of d_i-tuples [9]. The changes to a relation r_i in the database consist of insertions of new tuples and deletions of existing tuples. The new state of a relation r_i after applying a change is represented as r_i'. An update to an existing tuple can be modeled as a deletion followed by an insertion. The insertion of new tuples into a relation r_i and the deletion of existing tuples from r_i are represented as the differential relations r_i^+ and r_i^- respectively. Prior work in [22] presents a set of *differential rules* to compute the differentials (v^+ or v^-) of a non-aggregate view v. We extend the framework that is presented in [22] to compute the differentials of aggregate views.

Consider a view v in our language that is defined over the formula $\phi(\bar{X}, \bar{Y}, \bar{Z})$ using the aggregation predicate *agg* as follows.

$$v(\bar{X}, A) \text{ :- setof}(\bar{Y},\ \phi(\bar{X}, \bar{Y}, \bar{Z}),\ W),\ \text{agg}(W, A)$$

Since we define aggregates as predicates over sets in our language, we can rewrite the definition of the view v using the following two rules, one of which contains setof subgoals while the other does not.

$$v(\bar{X}, A) \text{ :- } u(\bar{X}, W),\ \text{agg}(W, A)$$
$$u(\bar{X}, W) \text{ :- setof}(\bar{Y},\ \phi(\bar{X}, \bar{Y}, \bar{Z}),\ W)$$

We note that if the definition of v contains k setof subgoals instead of one, we can rewrite the definition of v using $k + 1$ rules where only k of the rules contain setof subgoals. The differential rules in [22] already compute the differentials of views that do not contain sets. Hence, we focus on computing the differentials of views whose definitions contain setof subgoals.

There are two possible ways in which a view v can be defined in our language using the setof operator over the formula $\phi(\bar{X}, \bar{Y}, \bar{Z})$.

1. The view v is defined as $v(\bar{X}, \bar{Z}, W)$:- setof($\bar{Y}$, $\phi(\bar{X}, \bar{Y}, \bar{Z})$, W). In this case, all of the variables of ϕ that are bound outside the setof subgoal are *passed* to the view v.
2. The view v is defined as $v(\bar{X}, W)$:- setof($\bar{Y}$, $\phi(\bar{X}, \bar{Y}, \bar{Z})$, W). In this case, all of the variables of ϕ that are bound outside the setof subgoal are *not passed* to the view v.

We consider the above two cases separately and present differential rules to compute the differentials of the view v in each case.

Case 1: Suppose that a view v is defined over a conjunction of subgoals $\phi(\bar{X}, \bar{Y})$ as follows.

$$v(\bar{X}, W) \text{ :- setof}(\bar{Y},\ \phi(\bar{X}, \bar{Y}),\ W)$$

Suppose we define a view u over $v(\bar{X}, W)$ as $u(\bar{X})$:- $v(\bar{X}, W)$. In this case, the following differential rules correctly compute the differential relations $v^+(\bar{X}, W)$ and $v^-(\bar{X}, W)$ in response to the changes to $\phi(\bar{X}, \bar{Y})$.

$$v^+(\bar{X}, W) \text{ :- setof}(\bar{Y},\ \phi^+(\bar{X}, \bar{Y}),\ W),\ \neg u(\bar{X}) \qquad (\Delta_1)$$

$$v^+(\bar{X}, W \cup W') \text{ :- setof}(\bar{Y},\ \phi^+(\bar{X}, \bar{Y}),\ W),\ v(\bar{X}, W') \qquad (\Delta_2)$$

$$v^+(\bar{X}, W' \setminus W) \text{ :- setof}(\bar{Y},\ \phi^-(\bar{X}, \bar{Y}),\ W),\ v(\bar{X}, W') \qquad (\Delta_3)$$

$$v^-(\bar{X}, W) \text{ :- setof}(\bar{Y},\ \phi^-(\bar{X}, \bar{Y}),\ W),\ v(\bar{X}, W) \qquad (\Delta_4)$$

$$v^-(\bar{X}, W) \text{ :- setof}(\bar{Y},\ \phi^+(\bar{X}, \bar{Y}),\ \_),\ v(\bar{X}, W) \qquad (\Delta_5)$$

$$v^-(\bar{X}, W) \text{ :- setof}(\bar{Y},\ \phi^-(\bar{X}, \bar{Y}),\ \_),\ v(\bar{X}, W) \qquad (\Delta_6)$$

In the above differential rules, '_' represents *don't care* variables. We prove the correctness of the differential rules Δ_1– Δ_6 in the following theorem.

Theorem 1. *The differential rules Δ_1– Δ_6 correctly maintain a view containing a setof subgoal where all the variables that are bound outside the setof are passed to the head.*

Proof. Consider a view v that is defined over a conjunction of subgoals $\phi(\bar{X}, \bar{Y})$ as follows.

$$v(\bar{X}, W) \text{ :- setof}(\bar{Y},\ \phi(\bar{X}, \bar{Y}),\ W)$$

In the definition of v, the variables that are bound outside the setof subgoal i.e. $\bar{X}$ are passed to the head. Hence the view v contains exactly one tuple $(\bar{X}, W)$ for every distinct value of $\bar{X}$. Suppose a tuple, say $\phi(\bar{x}, \bar{y})$, is **inserted**. Either the view v does not contain any tuple $v(\bar{X}, W)$ where $\bar{X} = \bar{x}$ or v contains a tuple $v(\bar{x}, w)$. For correctly maintaining the view v, the tuple $(\bar{x}, \{\bar{y}\})$ is inserted into the view v in the former case. The differential rule Δ_1 handles this case. In the latter case, the tuple $v(\bar{x}, w)$ is updated to $v(\bar{x}, w \cup \{\bar{y}\})$ to correctly update the view v. The differential rules Δ_2 and Δ_5 capture this update.

Suppose a tuple, say $\phi(\bar{x}, \bar{y})$, is **deleted**. Either v contains the tuple $v(\bar{x}, \{\bar{y}\})$ or v contains the tuple $v(\bar{x}, W)$ where $\{\bar{y}\} \subset W$. For correctly maintaining the view v, the tuple $(\bar{x}, \{\bar{y}\})$ is deleted from the view v in the former case. The differential rule Δ_4 captures this deletion. In the latter case, the tuple $v(\bar{x}, w)$ is updated to $v(\bar{x}, w \setminus \{\bar{y}\})$ to correctly update the view v. The differential rules Δ_3 and Δ_6 capture this update.

Case 2: Now, suppose that a view v is defined over a conjunction of subgoals $\phi(\bar{X}, \bar{Y}, \bar{Z})$ as follows.

$$v(\bar{X}, W) \text{ :- setof}(\bar{Y},\ \phi(\bar{X}, \bar{Y}, \bar{Z}),\ W)$$

In addition, suppose that we define a view u over $v(\bar{X}, W)$ as $u(\bar{X})$:- $v(\bar{X}, W)$. In the definition of v, the set W is computed as $\{\bar{Y} \mid \phi(\bar{X}, \bar{Y}, \bar{Z})\}$ for every binding of $\bar{X}$ and $\bar{Z}$. Since $\bar{Z}$ is not passed to the view v, a view tuple, say $v(x, w)$, potentially has multiple derivations. In this case, we compute the differentials $v^+(\bar{X}, W)$ and $v^-(\bar{X}, W)$ as follows.

$$v^+(\bar{X}, W) \text{ :- setof}(\bar{Y},\ \phi^+(\bar{X}, \bar{Y}, \bar{Z}),\ W),\ \neg u(\bar{X}) \tag{Γ_1}$$

$$\begin{aligned} v^+(\bar{X}, W \cup W') \text{ :- setof}(\bar{Y},\ \phi^+(\bar{X}, \bar{Y}, \bar{Z}),\ W),\ \text{setof}(\bar{Y}',\ \phi(\bar{X}, \bar{Y}', \bar{Z}),\ W'),\\ \neg v(\bar{X}, W \cup W') \end{aligned} \tag{Γ_2}$$

$$\begin{aligned} v^+(\bar{X}, W' \setminus W) \text{ :- setof}(\bar{Y},\ \phi^-(\bar{X}, \bar{Y}, \bar{Z}),\ W),\ \text{setof}(\bar{Y}',\ \phi(\bar{X}, \bar{Y}', \bar{Z}),\ W'),\\ \neg v(X, W' \setminus W) \end{aligned} \tag{Γ_3}$$

$$\begin{aligned} v^-(\bar{X}, W) \text{ :- setof}(\bar{Y},\ \phi^-(\bar{X}, \bar{Y}, \bar{Z}),\ W),\ v(\bar{X}, W),\\ \neg\text{setof}(\bar{Y}',\ \phi'(\bar{X}, \bar{Y}', \bar{Z}),\ W) \end{aligned} \tag{Γ_4}$$

$$\begin{aligned} v^-(\bar{X}, W) \text{ :- setof}(\bar{Y},\ \phi^+(\bar{X}, \bar{Y}, \bar{Z}),\ \_),\ \text{setof}(\bar{Y}',\ \phi(\bar{X}, \bar{Y}', \bar{Z}),\ W),\\ \neg\text{setof}(\bar{Y}'',\ \phi'(\bar{X}, \bar{Y}'', \bar{Z}),\ W) \end{aligned} \tag{Γ_5}$$

$$\begin{aligned} v^-(\bar{X}, W) \text{ :- setof}(\bar{Y},\ \phi^-(\bar{X}, \bar{Y}, \bar{Z}),\ \_),\ \text{setof}(\bar{Y}',\ \phi(\bar{X}, \bar{Y}', \bar{Z}),\ W),\\ \neg\text{setof}(\bar{Y}'',\ \phi'(\bar{X}, \bar{Y}'', \bar{Z}),\ W) \end{aligned} \tag{Γ_6}$$

We prove the correctness of the differential rules Γ_1– Γ_6 in the following theorem.

Theorem 2. *The differential rules Γ_1– Γ_6 correctly maintain a view containing a setof subgoal where all the variables that are bound outside the setof are not passed to the head.*

The proof of theorem 2 is similar to the proof of theorem 1, except that before deleting a tuple from the view we check for alternate derivations of the tuple in the updated subgoals. In addition, a tuple is inserted into the view only if the view does not contain an alternate derivation of the tuple to be inserted.

4 Efficient Incremental Maintenance

In the previous section, we extended the differential rules that are presented in [22] to incrementally compute the differentials of views containing setof subgoals. In this section, we leverage the differential rules (from section 3) to *optimally* maintain views containing setof subgoals. As a first step, we present an example where the differential rules are leveraged to incrementally maintain views containing setof subgoals.

Example 3. Consider a materialized view *dominates* which is defined over the tournament relation from our running example (in section 1) as follows.

$$\text{dominates}(V, W) \text{ :- setof}(D, \text{tournament}(V, D, L), W)$$

The extension of the view dominates that corresponds to the extension of tournament (in table 2) is presented below.

dominates	
Victor	**Defeated**
yoda	{palpatine, vader}
vader	{yoda}
yoda	{palpatine}

Suppose that Yoda defeats Darth Vader at Tatooine in a new tournament match. This match results in the insertion of the tuple (yoda, vader, tatooine) into the tournament relation i.e. (yoda, vader, tatooine) $\in$ tournament$^+$. Since the non-aggregated variable L in tournament is not passed to the view dominates, we apply the differential rules Γ_1– Γ_6 to incrementally compute the differentials of the view dominates. By applying the differential rules Γ_2 and Γ_5 on the differential tournament$^+$ and the relations tournament and dominates, we derive the differentials dominates$^-$(yoda, {palpatine}) and dominates$^+$(yoda, {palpatine, vader}). The computed differentials correspond to updating the tuple (yoda, {palpatine}) $\in$ dominates to the tuple (yoda, {palpatine, vader}).

We note that in example 3, the differential rules Γ_2 and Γ_5 access tournament's extension in addition to the differential tournament$^+$ to maintain the view dominates. A tuple $(V, W) \in$ dominates could potentially have multiple derivations in tournament because V could defeat the *same* set of characters W at multiple planetary systems. Hence, additional accesses to the extensions of edb relations are required to maintain materialized views using differential rules.

Alternatively, we could maintain the count of the different derivations of tuples to optimize the maintenance of aggregate views. Prior techniques [10, 14, 15, 20, 23] have leveraged this idea to optimize the maintenance of views where the tuples in the view have multiple derivations in the edb relations. Suppose that in example 3, we maintain the count of the different derivations of a tuple in a manner similar to the *counting algorithm* that is presented in [14].

dominates		
Victor	**Defeated**	**Number of Derivations**
yoda	{palpatine, vader}	1
vader	{yoda}	1
yoda	{palpatine}	1

Now suppose that we delete a tuple, say (yoda, vader, dagobah) from tournament's extension. In this case, we decrease the count of the tuple (yoda, {palpatine, vader}) $\in$ dominates from 1 to 0 (thereby deleting it from the view) and increase the count of the tuple (yoda, {palpatine}) from 1 to 2.

dominates		
Victor	**Defeated**	**Number of Derivations**
vader	{yoda}	1
yoda	{palpatine}	2

When the tuple (yoda, vader, dagobah) is deleted from tournament's extension, we do not have to access tournament's extension to incrementally maintain the materialized view dominates. However, consider a scenario where we delete the tuple (yoda, palpatine, dagobah) instead of the tuple (yoda, vader, palpatine) from tournament's extension. In this scenario, unless we access tournament's extension, we *cannot* correctly update the materialized view dominates because we do not have sufficient information to determine whether the existing tuple (yoda, {palpatine}) $\in$ dominates is to be deleted or the tuple (yoda, {palpatine, vader}) $\in$ dominates is to be updated.

Incremental Maintenance using CReaM[1]: Consider the materialized view dominates that we presented in example 3. Suppose we rewrite the definition of dominates using an auxiliary view v_a as follows.

$$\text{dominates}(V, W) \text{ :- } v_a(V, L, W)$$
$$v_a(V, L, W) \text{ :- setof}(D, \text{tournament}(V, D, L), W)$$

In addition, suppose that we materialize the auxiliary view v_a and maintain the counts of the derivations of a tuple in the view dominates. The extension of the auxiliary view v_a is presented below.

v_a		
Victor	**Location**	**Defeated**
yoda	dagobah	{palpatine, vader}
vader	tatooine	{yoda}
yoda	tatooine	{palpatine}

Now, suppose that we delete the tuple (yoda, palpatine, dagobah) from the extension of tournament. Since all of the non-aggregated variables of tournament are passed to the auxiliary view v_a, we can incrementally maintain v_a using the differential rules $\Delta_1 - \Delta_6$ (from section 3). We note that $\Delta_1 - \Delta_6$ only access the extension of a view and the differentials of the edb relations over which the view is defined. Thus, we are able to compute the differentials v_a^-(yoda, dagobah, {palpatine, vader}) and v_a^+(yoda, dagobah, {vader}) without accessing the extension of tournament.

Since the modified definition of the view dominates *does not* contain setof subgoals, we use the counting algorithm [14] to incrementally maintain the count of the tuple derivations in the view dominates in a subsequent step. The updated extension of the view dominates is presented below.

dominates		
Victor	**Defeated**	**Number of Derivations**
vader	{yoda}	1
yoda	{palpatine}	2

[1] The algorithm has been named CReaM because it **C**ounts the tuple derivations in a view, **Re**writes the view using auxiliary views and **M**aintains the auxiliary views.

We now propose an algorithm called CReaM to incrementally maintain views containing setof subgoals. The CReaM algorithm is presented in figure 1. In step 1 of the algorithm, the supplied view is rewritten using an auxiliary view which is materialized in a subsequent step. In step 3 of the algorithm, count of the tuple derivations in the supplied view is maintained. The *incremental maintenance* of the view v is carried out in step 4 of the algorithm by computing the differentials of the auxiliary view which was created in step 1 of the algorithm.

Fig. 1. Algorithm to *optimally* maintain views containing setof subgoals

CReaM Algorithm

Input: 1. Materialized view $v(\bar{X}, W)$ defined as:
$v(\bar{X}, W)$:- setof$(\bar{Y}, \phi(\bar{X}, \bar{Y}, \bar{Z}), W)$,
2. Differentials $\phi^+(\bar{X}, \bar{Y}, \bar{Z})$ and $\phi^-(\bar{X}, \bar{Y}, \bar{Z})$

Step 1: **Rewrite** the view v using an auxiliary view v_a which contains all of the non-aggregated variables
$v(\bar{X}, W)$:- $v_a(\bar{X}, \bar{Z}, W)$
$v_a(\bar{X}, \bar{Z}, W)$:- setof$(\bar{Y}, \phi(\bar{X}, \bar{Y}, \bar{Z}), W)$

Step 2: **Materialize** the auxiliary view v_a

Step 3: Maintain the **count** of the tuple derivations in the view v

Step 4: Apply the differential rules Δ_1- Δ_6 over $\phi^+(\bar{X}, \bar{Y}, \bar{Z})$ and $\phi^-(\bar{X}, \bar{Y}, \bar{Z})$ to compute $v_a^+(\bar{X}, \bar{Z}, W)$ and $v_a^-(\bar{X}, \bar{Z}, W)$
Use $v_a^+(\bar{X}, \bar{Z}, W)$ and $v_a^-(\bar{X}, \bar{Z}, W)$ to incrementally update the counts of v's tuples using [9]

We note that the CReaM algorithm incrementally maintains a view whose definition contains a single setof subgoal. However, when the supplied view definition contains multiple setof subgoals and aggregate predicates, we can incrementally maintain the view using CReaM as follows. Suppose a materialized view v contains k setof subgoals $\{s_i\}$ and m aggregate predicates $\{a_i\}$. First, we rewrite the definition of v using k auxiliary predicates, say $\{t_i\}$ where each t_i is defined as t_i :- s_i. Next, we maintain the counts of the tuple derivations in v and incrementally compute the differentials of t_i by applying CReaM to the extensions of the auxiliary predicates $\{t_i\}$ and the differentials of the edb relations. Since the modified definition of v does not contain setof subgoals, we use the counting algorithm that is presented in [14] to incrementally maintain the materialized view v.

In the following theorem, we establish that CReaM correctly maintains views that contain setof subgoals.

Theorem 3. *CReaM correctly maintains a materialized view containing setof subgoals.*

Proof. Consider a view v in our language which is defined using k setof subgoals $s_1, s_2, \ldots, s_k$ as v :- $s_1, s_2, \ldots, s_k$. Suppose that we introduce k auxiliary views v_{a_1}, v_{a_2}, $\ldots$, v_{a_k} where each v_{a_i} is defined as v_{a_i} :- s_i. In the definition of the auxiliary view v_{a_i}, all of the variables that are bound outside the setof subgoal s_i are passed to the view. By replacing the setof subgoals using the auxiliary views, we can rewrite the definition of the view v as v :- $v_{a_1}, v_{a_2}, \ldots, v_{a_k}$. Since the modified definition of the view v does not contain setof subgoals, we can correctly maintain it by applying the

counting algorithm [14]. In addition, we can leverage the rules Δ_1– Δ_6 to correctly compute the differentials of the auxiliary views $\{v_{a_i}\}$ by theorem 1.

In the next section, we discuss the performance of CReaM and prove that it optimally maintains materialized views containing setof subogals.

5 Performance of CReaM

Previous aggregate view maintenance algorithms [10, 15, 14, 20, 23, 26, 27] do not materialize and maintain additional views. Instead, the algorithms leverage differential relational algebra [10, 15, 14, 26, 27] or maintain the count of tuple derivations [10, 15, 14, 20, 23] to efficiently maintain aggregate views. In this section, we show that by rewriting, and maintaining additional auxiliary views, CReaM speeds up the time taken to incrementally maintain a view containing setof subgoals in comparision to previous view maintenance algorithms. As our underlying cost model, we assume that the time taken by an algorithm to incrementally maintain a view is proportional to the number of tuple accesses that are required by the algorithm to maintain the view. As a first step, we discuss the performance of CReaM when a single tuple is changed in the underlying edb relations. We then discuss the performance of CReaM with respect to multiple tuple updates.

Consider a view v which is defined as $v(\bar{X}, W)$:- setof$(\bar{Y}, \phi(\bar{X}, \bar{Y}, \bar{Z}), W)$. Suppose the extensions of v and ϕ consist of n_v and n_ϕ tuples respectively. The extension of the view v and the differential ϕ^+ and ϕ^- are provided as inputs to the CReaM algorithm (see figure 1). In steps 1 and 2, CReaM rewrites definition of v using an auxiliary view v_a and materializes v_a. Suppose that the number of tuples in v_a is n_{v_a}. Then, $n_v \leq n_{v_a} \leq n_\phi$. In step 3, CReaM maintains the count of the tuple derivations in v. In our analysis of the time required to maintain a view using CReaM, we ignore the time taken by steps 1, 2, and 3 because v_a can be materialized and the derivations of the tuples in v can be counted before changing ϕ.

In step 4, CReaM maintains v_a using the differential rules Δ_1– Δ_6. The time required to update v_a using Δ_1– Δ_6 is equal to the time required to compute the join of the view v_a and the differentials ϕ^+ and ϕ^-. When the underlying database is optimized i.e. the materialized view v_a is indexed on the attributes $\bar{X}$ and $\bar{Z}$, the time required to compute the join is O$(\log n_{v_a})$, otherwise it is O(n_{v_a}). In a subsequent step, CReaM leverages the differentials v_a^+ and v_a^- that were previously computed using Δ_1– Δ_6 to update the extension of the view view v using the counting algorithm [14]. Since the view v is a projection of the the view v_a, the time required to incrementally maintain v in response to the differentials v_a^+ and v_a^- is either O$(\log n_v)$ or O(n_v) depending on whether the attribute $\bar{X}$ in the view v is indexed or not. Therefore, the time taken by CReaM to update a view v with n_v tuples in response to a single tuple update in the underlying edb relations is either O$(\log n_{v_a})$ or O(n_{v_a}) depending on whether the physical design of the database is optimized or not.

Suppose that we did not materialize the auxiliary view v_a. In this case, we cannot update v without accessing the extension of ϕ. Suppose that the differential of ϕ consists of a single tuple $\phi(\bar{x}, \bar{y}, \bar{z})$. We need to recompute the set $S_{\bar{x},\bar{z}} = \{\bar{Y} \mid \phi(\bar{x}, \bar{Y}, \bar{z})\}$ to

incrementally update v. We analyze the time required to update v under two possible scenarios depending on whether ϕ consists of a single edb relation or ϕ is a conjunction of edb relations. In the first case, the time required to incrementally compute $S_{\bar{x},\bar{z}}$ is either O($\log n_\phi$) or O(n_ϕ) depending on whether an index exists on the attributes $\bar{X}$ and $\bar{Z}$ in ϕ or not. However, in the second case, when ϕ is a conjunction of edb relations, we need to recompute ϕ and update it before recomputing the set $S_{\bar{x},\bar{z}}$. In this case, the cost of recomputing and updating ϕ is dominated by the cost of computing the join of the edb relations which is either O(n_ϕ) or O($n \times \log n_\phi$) depending on whether the physical design of the database is optimized or not.

When there are multiple (say k) changes to ϕ, the time required by CReaM to incrementally update the view $v(\bar{X}, W)$:- setof($\bar{Y}, \phi(\bar{X}, \bar{Y}, \bar{Z}), W$) with n_v tuples is either O($k \times \log n_v$) or O($k \times n_v$) depending on whether the views v_a and v are indexed or not. However, when v_a is not materialized and ϕ is a conjunction of multiple edb relations, we have to recompute the sets over ϕ. This requires O($n \times \log n$) time. Therefore, the speed up in the time to update v using CReaM in comparison to previous view maintenance algorithms is by a factor of $\frac{n}{k}$ when v_a and v are indexed.

We note that if the view v is defined as $v(\bar{X}, \bar{Z}, W)$:- setof($\bar{Y}, \phi(\bar{X}, \bar{Y}, \bar{Z}), W$), we can update v without accessing or computing the extension of ϕ. In this case, the time taken to update v is the same as is required by CReaM. The summary of CReaM's performance is presented in table 1.

Next, we show that when the supplied materialized view and the auxiliary views that are materialized by CReaM are indexed, the time taken by CReaM to incrementally maintain a view is *optimal*.

Theorem 4. *CReaM optimally maintains a materialized view containing a setof subgoal in response to changes in edb relations when the physical design of the underlying database is optimized.*

Proof. Suppose that a materialized view v containing a setof subgoal is supplied as an input to CReaM. In addition, suppose that v consists of n tuples. When v and the auxiliary view (that is materialized by CReaM) are indexed, CReaM maintains v in O($\log n$) time. If we prove that $\Omega(\log n)$ time is required to incrementally maintain an extension of a view with n tuples, then we would establish the optimality of CReaM.

To prove the lower bound, we reduce problem of incrementally maintaining the *partial sums* of an array of n numbers to the problem of incrementally maintaining an extension of a view with n tuples. Prior work in [7, 8, 24] have independently proven that the maintenance of partial sums of an array of n numbers requires $\Omega(\log n)$ time. Consider an array of n numbers $\{a_i\}$. The partial sums problem maintains the sum $\sum_{i=0}^{k} a_i$ for every k ($1 \leq k \leq n$) subject to updates of the form $a_i = a_i + x$, where x is a number. We reduce the instance of the partial sums problem over the array $\{a_i\}$ to an instance of the view maintenace problem in time that is polynomial in n as follows. Consider an instance of the view maintenance problem where we have two edb relations $r(A, B)$ and $s(B, C)$. The extension of $r(A, B)$ consists of the set of $n \times (n-1)$ tuples, $\{(i, j) \mid 1 \leq i \leq j \leq n\}$. The extension of $s(A, B)$ consists of the set of n tuples, $\{(i, a_i) \mid 1 \leq i \leq n\}$. Suppose that we materialize n views $v_1, v_2, \ldots, v_n$ over $r(A, B)$ and $s(B, C)$ where each v_i is defined as $v_i(S)$:- setof($(B, C), r(i, B)$ & $s(B, C), W$),

sum$(W, S, 2)$. In the definition of v_i, the aggregate sum$(W, S, 2)$ computes the sum of the 2[nd] component of the tuples $\in W$.

When an array value a_i is updated to $a_i + x$, we update the tuple $(i, a_i) \in s(B, C)$ to the tuple$(i, a_i + x)$. Since we can compute the partial sum $\sum_{i=0}^{k} a_i$ by finding the value s which is in the extension of v_k, the problem of maintaining the partial sums of the array $\{a_i\}$ reduces to the problem of incrementally maintaining the views $v_1, v_2, \ldots, v_k$. Therefore, if the number of tuples in an extension of a view that contains a setof subgoal is $O(n)$, then $\Omega(\log n)$ time is required to incrementally maintain the view.

Self-Maintenance of Aggregate Views: In theorem 4, we prove that CReaM optimally maintains views containing setof subgoals. Now, we show that by materializing auxiliary views, CReaM guarantees the self-maintainability [17] of aggregate views that are defined over single edb relations. In other words, the extension of an edb relation does not have to be accessed to incrementally maintain an aggregate view that is defined over the relation. The property of self-maintainability is desirable when access to the edb relations is restricted or when the edb relations themselves are hypothetical (such as in a LAV integration scenario [32]).

Consider a view v which is defined over an edb relation ϕ using the aggregation predicate *agg* as $v(\bar{X}, A)$:- setof$(\bar{Y}, \phi(\bar{X}, \bar{Y}, \bar{Z}), W)$, agg$(W, A)$. To incrementally maintain v, CReaM rewrites the definition of v using an auxiliary view v_a as follows.

$$v(\bar{X}, A) \text{ :- } v_a(\bar{X}, \bar{Z}, W), \text{ agg}(W, A)$$
$$v_a(\bar{X}, \bar{Z}, W) \text{ :- setof}(\bar{Y}, \phi(\bar{X}, \bar{Y}, \bar{Z}), W)$$

CReaM materializes the view v_a and incrementally computes the diffentials v_a^+ and v_a^- by applying the differential rules Δ_1– Δ_6. The differential rules Δ_1– Δ_6 compute the join of the extension of the view v_a and the differentials of ϕ i.e., ϕ^+ and ϕ^-. In a subsequent step, CReaM leverages the differentials of v_a to incrementally maintain the view v using the algorithm presented in [14]. By materializing the auxiliary view v_a, CReaM is able to maintain v without accessing the extension of ϕ, thereby, making the view v self-maintainable.

6 Related Work

The problem of incrementally maintaining views has been extensively studied in the database community [1–6, 10, 14–16, 18, 20–22, 25–27, 29–31, 33]. A survey of the view maintenance techniques is presented in [13]. The view maintenance algorithms proposed in [2, 10, 14, 15, 18, 22, 25, 26] leverage differential relational algebra to incrementally maintain views in response to changes to the underlying edb relations. For instance, the prior work presented in [22] incrementally computes the differentials (or changes) of views by applying a set of differential rules over the extensions of edb relations and their differentials.

However, only a small fraction of the prior work on incremental view maintenance [10, 14, 15, 20, 23, 26, 27] addresses the maintenance of aggregate views. The techniques proposed in [10, 14, 20, 23] incrementally maintain views having only one

aggregation operator. Furthermore, the incremental maintenance algorithms presented in [10, 14, 15, 20, 23, 26, 27] can support only a fixed set of built-in aggregate operators (such as min, max, sum, and count).

The problem of incremental view maintenance is closely related to the problem of self-maintainability [11, 12, 17, 19]. A view is self-maintainable if it can be incrementally maintained using the extension of the view and the changes to the edb relations. The view maintenance algorithms that are presented in [15, 14, 19] derive efficient self-maintenance expressions as well for certain types of updates to edb relations. Our view maintenance algorithm, CReaM, guarantees the self-maintenance of an aggregate view that is derived over a single edb relation by materializing auxiliary views.

Our work differs from prior work on incrementally maintaining aggregate views in two ways. First, we consider the maintenace of views with user-defined aggregates. Second, we propose an algorithm called CReaM which *optimally* maintains aggregate views. In addition, for the special class of aggregate views where all of the non-aggregated attributes of the underlying edb relations are passed to the view, CReaM speeds up the time taken to incrementally update the view in comparison to previous view maintenance algorithms [10, 15, 14, 20, 23] by a factor that is at least logarithmic in the size of the extension of the view.

We note that even though CReaM optimally maintains aggregate views, we could further optimize the maintenance of views that contain *monotonic* aggregates [28] when new tuples are *inserted* to the edb relations. When new elements are inserted to a set that is aggregated by a monotonic aggregate, the aggregate value either always increases or decreases. For example, the aggregate sum is monotonic over the domain of postive numbers. Therefore, if we have a view v that is defined as $v(A)$:- setof$(B, r(A, B), W)$, sum(W, S), $S > 10$ and a tuple $t \in$ extension of v, the tuple t can never be changed by insertions into the relation $r(A, B)$.

7 Conclusion

We propose an algorithm called CReaM that incrementally maintains materialized aggregate views in response to changes to edb relations by materializing auxiliary views. By materializing auxiliary views, CReaM guarantees the self-maintainability of aggregate views that are defined over a single database table. CReaM optimally maintains views containing setof subgoals and speeds up the time taken to update materialized aggregate views with n tuples that are defined over multiple edb relations and do not contain all of the non-aggregated attributes in the edb relations either by a factor of $\frac{n}{\log n}$ or $\log n$ depending on whether the physical design of the underlying database in optimized or not. For other types of aggregate views, CReaM updates the view in no more time than that is required by prior view maintenance techniques to update the view.

References

1. Blakeley, J.A., Coburn, N., Larson, P.A.: Updating derived relations: detecting irrelevant and autonomously computable updates. ACM TODS (1989)

2. Blakeley, J.A., Larson, P.A., Tompa, F.W.: Efficiently updating materialized views. SIGMOD (1986)
3. Buneman, O.P., Clemons, E.K.: Efficiently monitoring relational databases. ACM TODS (1979)
4. Ceri, S., Widom, J.: Deriving production rules for incremental view maintenance. VLDB (1991)
5. Colby, L.S., Kawaguchi, A., Lieuwen, D.F., Mumick, I.S., Ross, K.A.: Supporting multiple view maintenance policies. SIGMOD (1997)
6. Dong, G., Topor, R.W.: Incremental evaluation of datalog queries. ICDT (1992)
7. Fredman, M.L.: A lower bound on the complexity of orthogonal range queries. J. ACM (1981)
8. Fredman, M.L.: The complexity of maintaining an array and computing its partial sums. J. ACM (1982)
9. Gallaire, H., Minker, J., Nicolas, J.M.: Logic and databases: A deductive approach. ACM Computing Surveys (1984)
10. Griffin, T., Libkin, L.: Incremental maintenance of views with duplicates. SIGMOD (1995)
11. Gupta, A., Jagadish, H.V., Mumick, I.S.: Data integration using self-maintainable views. EDBT (1996)
12. Gupta, A., Jagadish, H.V., Mumick, I.S.: Maintenance and self-maintenance of outerjoin views. NGITS (1997)
13. Gupta, A., Mumick, I.S.: Materialized views. chap. Maintenance of materialized views: problems, techniques, and applications (1999)
14. Gupta, A., Mumick, I.S., Subrahmanian, V.S.: Maintaining views incrementally. SIGMOD (1993)
15. Gupta, H., Mumick, I.S.: Incremental maintenance of aggregate and outerjoin expressions. Information Systems (2006)
16. Harrison, J.V., Dietrich, S.W.: Maintenance of materialized views in a deductive database: An update propagation approach. Workshop on Deductive Databases, JICSLP (1992)
17. Huyn, N.: Efficient view self-maintenance. Views (1996)
18. Kuchenhoff, V.: On the efficient computation of the difference between consecutive database states. DOOD (1991)
19. Mohania, M., Kambayashi, Y.: Making aggregate views self-maintainable. ACM TKDE 32 (1999)
20. Mumick, I.S., Quass, D., Mumick, B.S.: Maintenance of data cubes and summary tables in a warehouse. SIGMOD (1997)
21. Nicolas, J.M., Yazdanian: An outline of bdgen: A deductive dbms. Information Processing (1983)
22. Orman, L.V.: Differential relational calculus for integrity maintenance. ACM TKDE (1998)
23. Palpanas, T., Sidle, R., Cochrane, R., Pirahesh, H.: Incremental maintenance for non-distributive aggregate functions. VLDB (2002)
24. Păatraşcu, M., Demaine, E.D.: Tight bounds for the partial-sums problem. SODA (2004)
25. Qian, X., Wiederhold, G.: Incremental recomputation of active relational expressions. ACM TKDE (1991)
26. Quass, D.: Maintenance expressions for views with aggregation. Views (1996)
27. Quass, D., Mumick, I.S.: Optimizing the refresh of materialized view. Technical Report (1997)
28. Ross, K.A., Sagiv, Y.: Monotonic aggregation in deductive databases. PODS (1992)
29. Shmueli, O., Itai, A.: Maintenance of views. SIGMOD (1984)
30. Stonebraker, M.: Implementation of integrity constraints and views by query modification. SIGMOD (1975)

31. Tompa, F.W., Blakeley, J.A.: Maintaining materialized views without accessing base data. Information Systems (1988)
32. Ullman, J.D.: Principles of Database and Knowledge-Base Systems: Volume II (1989)
33. Wolfson, O., Dewan, H.M., Stolfo, S.J., Yemini, Y.: Incremental evaluation of rules and its relationship to parallelism. SIGMOD (1991)