Fully Dynamic Materialization Maintenance

Moritz Illich, Birte Glimm

Ulm University, James-Franck-Ring, 89081 Ulm, Germany

Abstract

Classically, consecutive updates on materialized datasets, obtained by computing every consequence for given inference rules, are considered separately without any direct interference. In contrast, we deal with *fully dynamic updates* (FDUs), where additions and deletions from different updates are processed concurrently and can directly influence each other. On the one hand, this enables immediate integration of any new update into the current processing, thus improving responsiveness. On the other hand, direct interference between additions and deletions may reverse or cancel operations. While this can prevent redundant computations, e.g., by stopping a deletion process if a fact is re-added, it might also lead to incorrect materializations if applied without further measures. In this work, we show that (i) FDUs require recursion handling and lead to incomplete processing of operations, (ii) classical incremental materialization approaches cannot fully deal with these challenges, and (iii) incremental materialization approaches can be adapted to allow for FDUs.

Keywords

materialization maintenance, fully dynamic, Datalog

1. Introduction

Answering queries on description logic (DL) ontologies with large ABoxes can be facilitated by transformations into Datalog [1], enabling the usage of optimized reasoning engines like RDFox [2] that are able to outperform state-of-the-art DL reasoners [3]. In Datalog, query answering often refers to *materialization* where we compute all consequences for a set of facts with respect to some inference rules such that every implicitly entailed fact is made explicitly available. Since updates to the original set of facts, in the form of additions and deletions, have to be reflected in the materialization too, but re-computing everything from scratch is usually expensive, we make use of *incremental materialization maintenance* algorithms that efficiently adapt a materialization. In detail, these algorithms focus on the actual changes in a materialization by computing new consequences based on added facts, as well as removing every fact that cannot be derived anymore due to introduced deletions.

Typically, incremental materialization algorithms, like *Delete/Rederive* (DRed) and *counting* [4], process one update at a time, without any direct interference between consecutive updates. In particular, a new update is not processed until the previous one is completed. In this work, we investigate *fully dynamic updates* that may overlap and influence each other. Accordingly, a new update can directly be integrated into the current processing without any delay, enabling high responsiveness. This is useful for applications with high update frequencies, like in the

DL 2023: 36th International Workshop on Description Logics, September 2–4, 2023, Rhodes, Greece

moritz.illich@uni-ulm.de (M. Illich); birte.glimm@uni-ulm.de (B. Glimm)

context of stream reasoning [5] where we answer continuous queries on data streams with respect to given background knowledge.

Related work refers to the parallelization of materialization maintenance. One approach that deals with RDF data is provided by Motik et al. [2], which should also be applicable for DRed and counting [6]. In addition, DynamiTE [7] enables parallel materialization in the context of RDF stream reasoning. Besides, fully dynamic materialization maintenance has similarities to incremental stream reasoning, as done in IMaRS [8, 9] for RDF triples annotated with window-based expiration times, or the work of Ren and Pan [10] dealing with updates on $\mathcal{EL}^{++}$ ontologies based on a truth maintenance system. However, in all of these works, concurrent execution only refers to operations within an update, while a simultaneous processing of different updates is not taken into account.

In general, additions and deletions are already processed concurrently in given approaches, but always refer to the same update. In contrast to this, fully dynamic updates also allow interference between operations from distinct updates introduced at different times. As a consequence, we can directly react to a new update and potentially utilize it to avoid redundant computations, for example, by canceling the propagation of a deletion if the related fact is re-added in the new update. Nevertheless, the additional interaction between updates also imposes some challenges regarding the correctness of the materialization maintenance. For instance, canceling an operation might lead to missing facts that actually should be present in the updated materialization. In this regard, the main goal of this work is to examine (i) the difficulties of realizing fully dynamic updates for correct incremental materialization maintenance in Datalog, (ii) to which degree DRed and the counting algorithm are already capable of handling those problems, and (iii) how we can adapt the discussed approaches to enable fully dynamic updates without sacrificing correctness.

2. Basics and Preliminaries

To formally define *Datalog*, we fix countable, disjoint sets of *predicates*, *constants* and *variables*. A *term* is a constant or a variable. An *atom* has the form $p(t_1, \ldots, t_k)$, where p is a k-ary predicate and each t_i, $1 \leq i \leq k$, is a term. We focus on unary and binary atoms only (i.e., $1 \leq k \leq 2$), which correspond to concepts/classes and roles/properties in an ontology, respectively. An atom is *ground* if it does not contain variables. A *fact* is a ground atom, and a *dataset* is a finite set of facts. A Datalog *rule* is a logical implication of the form $B_1, \ldots, B_k \rightarrow H$ where $B_1, \ldots, B_k$ are called *body atoms*, and H is a *head atom*. A rule is *safe* if variables that appear in the head also appear in a body atom. [1]

A *Datalog program* is a finite set of safe rules. Predicates that occur in the head of a rule are called *intensional* (IDB) predicates; all other predicates are *extensional* (EDB). Typically, only EDB predicates are allowed in datasets and their updates. This is without loss of generality as we can replace every IDB predicate p that is to be used in a dataset or an update by a new predicate p' and add rules of the form $p(x) \rightarrow p'(x)$ or $p(x, y) \rightarrow p'(x, y)$ depending on the

[1]For the sake of readability, examples considered throughout this text do not contain any variables or constants. Instead, we assume that the stated atoms or facts all implicitly share the same variable or constant, and may only differ in their predicate.

predicate's arity, such that p becomes an EDB predicate [11]. In materialization maintenance approaches like DRed, such a transformation is needed to identify the explicit introduction of an implicitly derivable fact as an alternative derivation [6].

Given a program P with a rule $B_1, \ldots, B_k \rightarrow H \in P$, we say that the predicate of a head H *depends* on p for p a predicate symbol occurring in $B_1, \ldots, B_k$; p is called *recursive* in P if p transitively depends on itself in P. A rule r is *recursive* if at least one of its body atoms has a predicate that depends on the head's predicate, and *non-recursive* otherwise. A program P is *non-recursive* if no predicate symbol occurring in P is recursive in P, else P is *recursive.*

A *substitution* σ is a partial mapping from variables to constants. For α a term, an atom, a rule, or a set of rules, $\alpha\sigma$ is the result of replacing each occurrence of a variable x in α on which σ is defined with $\sigma(x)$. If r is a rule and σ is a substitution mapping all variables of r to constants, then $r\sigma$ is an *instance* of r. For a rule $r = B_1, \ldots, B_k \rightarrow H$ and a dataset I, we define $r(I) = \{H\sigma \mid B_i\sigma \in I \text{ for all } 1 \leq i \leq k\}$. For a program P, we define $P(I) = \bigcup_{r \in P} r(I)$. Given a rule instance $r\sigma = B_1\sigma, \ldots, B_k\sigma \rightarrow H\sigma$, we say that the fact $B_i\sigma$, with $1 \leq i \leq k$, *(directly) derives* the fact $H\sigma$, and call $r\sigma$ a *(direct) derivation* of $H\sigma$. In general, derivation relations are transitive and may consist of a whole sequence of rule instances. A derivation is *recursive* if it allows a fact to derive itself, and *non-recursive* otherwise.

2.1. Materialization Maintenance

Let E be a dataset (called *explicit facts*). Then, let $I_0 = E$; for each $i \geq 1$, let $I_i = I_{i-1} \cup P(I_{i-1})$, and let $I_\infty = \bigcup_{i \geq 0} I_i$. The set I_∞ is the *materialization* of P w.r.t. E, denoted as $\mathsf{mat}(P, E)$.

Let E, E^+, and E^- be datasets with $E \cap E^+ = \emptyset$, $E^- \subseteq E$, and $E^+ \cup E^- \neq \emptyset$. The tuple $U = (E^+, E^-)$ is called an *update* for E where E^+ denotes the facts explicitly added to E, and E^- the facts explicitly deleted from E, respectively. Applying the update U on E leads to the updated dataset $E' = (E \setminus E^-) \cup E^+$. Let $\hat{U} = \langle U_1, \ldots, U_n \rangle$ with $n \geq 1$ denote a *sequence of updates* for a dataset E_0, where for every update $U_i = (E_i^+, E_i^-)$, with $1 \leq i \leq n$, we have $E_{i-1} \cap E_i^+ = \emptyset$ and $E_i^- \subseteq E_{i-1}$, resulting in $E_i = (E_{i-1} \setminus E_i^-) \cup E_i^+$. Based on this, we define *materialization maintenance* as the task of computing the updated materialization $\mathsf{mat}(P, E_n)$ for a given materialization $\mathsf{mat}(P, E_0)$ and a sequence of updates $\hat{U} = \langle U_1, ..., U_n \rangle$. Note that the program P stays the same, i.e., we do not consider changes to the set of rules.

Typically, the application of updates in a sequence $\hat{U}$ is strictly consecutive, such that an update U_i is not processed until the preceding update U_{i-1} is completed. Concretely, we first apply U_1 on $\mathsf{mat}(P, E_0)$ to obtain $\mathsf{mat}(P, E_1)$, then apply U_2 on $\mathsf{mat}(P, E_1)$ to obtain $\mathsf{mat}(P, E_2)$, and continue this procedure until we finally reach $\mathsf{mat}(P, E_n)$. In contrast, updates are *fully dynamic* if their sequential application allows for overlaps between updates. Specifically, the processing of an update U_i may directly start even if the preceding update U_{i-1} is not yet completed. An important aspect here is that the result of fully dynamic updates has to be the same as for strictly consecutive ones. This means that the order of the updates in the sequence is still taken into account despite the overlapping, which is relevant if $E_i^+ \cap E_j^- \neq \emptyset$ for two updates U_i and U_j. Besides, the whole sequence of updates does not need to be available directly from the beginning, instead the updates may be introduced one by one with a potential delay between them. Consequently, we also allow infinite sequences of updates.

Algorithm 1 Fully dynamic materialization maintenance

```
Input: Datalog program $P$,   dataset $I$,   (infinite) sequence of updates $\hat{U}$
1: initialize empty set $O$ for operations
2: while updates available do
3:     if new update $U$ given in $\hat{U}$ then
4:         determine explicit operations
5:     apply one of the following computation steps:
6:         (1) determine implicitly | (2) cancel | (3) execute one operation
```

To formalize incremental materialization with fully dynamic updates, we define two *operations* $add(F)$ and $del(F)$ representing the addition and deletion of some fact F. For the processing of these operations, we introduce three groups of *transition rules* that determine, cancel, or execute an operation. Let P be a Datalog program, I a dataset, O a set of operations, U an update, and $r\sigma$ a rule instance with $r \in P$ and σ a substitution. For each of the following transition rules, the statements above the line describe needed preconditions, while the ones below state the conclusion, respectively.

1. Determine an operation
   i) explicitly based on an update

$$\frac{U = (E^+, E^-)}{O = O \cup \{add(F^+), del(F^-) \mid F^+ \in E^+, F^- \in E^-\}}$$

   ii) implicitly based on a rule
      a) addition

$$\frac{\begin{array}{c} r\sigma = B_1\sigma, \ldots, B_k\sigma \rightarrow H\sigma \\ \{add(B_i\sigma) \mid 1 \leq i \leq k\} \cap O \neq \emptyset \\ B_i\sigma \notin I \rightarrow add(B_i\sigma) \in O \text{ for } 1 \leq i \leq k \end{array}}{O = O \cup \{add(H\sigma)\}}$$

      b) deletion

$$\frac{\begin{array}{c} r\sigma = B_1\sigma, \ldots, B_k\sigma \rightarrow H\sigma \\ \{del(B_i\sigma) \mid 1 \leq i \leq k\} \cap O \neq \emptyset \\ B_i\sigma \notin I \rightarrow del(B_i\sigma) \in O \text{ for } 1 \leq i \leq k \end{array}}{O = O \cup \{del(H\sigma)\}}$$

2. Cancel an operation

$$\frac{add(F) \in O \quad del(F) \in O}{O = O \setminus \{add(F)\} \text{ or } O = O \setminus \{del(F)\}}$$

3. Execute an operation
   a) addition

$$\frac{add(F) \in O}{I = I \cup \{F\}}$$

   b) deletion

$$\frac{del(F) \in O}{I = I \setminus \{F\}}$$

For example, the transition rule for determining an implicit addition extends O with an add operation for the head of a rule provided a body atom is marked for addition ($add(B_i\sigma) \in O$) and where, additionally, the other body atoms are facts in the dataset (I). Moreover, canceling an operation is non-deterministic, such that different materialization maintenance approaches may define their own conditions under which one operation removes its counterpart.

Based on these transition rules, Algorithm 1 describes the general procedure of *fully dynamic materialization maintenance*. The main idea here is that we constantly alternate between checking for a new update and performing one computation to assure the direct integration of the former. The choice of the computation step, along with needed operations and rule instances, depends on the concrete materialization maintenance approach. We assume a general *fairness* condition such that every applicable rule will be considered eventually and computations do not just focus on one update. In addition, the order of updates has to be transparent to ensure that a new explicitly introduced operation is not canceled by an old, outdated one.

2.1.1. Delete/Rederive

One way to incrementally compute updates on materializations is described by the *Delete/Rederive* (DRed) algorithm [4], which works for both non-recursive and recursive Datalog programs. The algorithm takes as input a materialization $M = \mathsf{mat}(P, E)$ together with one update $U = (E^+, E^-)$ and produces as output the updated materialization $M' = \mathsf{mat}(P, (E \setminus E^-) \cup E^+)$. The processing is separated into an overdeletion, a rederivation, and an insertion phase, where the first two handle the deletions E^-, while the last one takes care of the additions E^+.

During *overdeletion*, all facts from M that are contained in or derived by some fact from E^- are deleted. This is achieved by propagating the deletion over direct derivations. For this, we first introduce deletion operations for all facts in E^-. We then determine all implicitly following deletions and repeat this process introducing new deletions each time until reaching a fix-point. At this point, the deletions are executed. This procedure guarantees that we remove every fact from M that can no longer be derived in M', but it potentially also deletes facts that have alternative derivations and that should not be affected by the deletion. To fix this, overdeletion is followed by the *rederivation* phase, where we re-add each deleted fact that can still be derived by the remaining facts. In detail, we try to determine an addition $add(F)$ for each deleted fact F based on the remaining facts. If we find such an addition, we execute it, cancel the related deletion, and continue the search until no further rederivations are possible. Finally, the *insertion* phase adds E^+ and iteratively computes all newly derivable facts.

For an efficient execution, a *semi-naive* evaluation [12] ensures that every time we consider a rule instance to derive a fact, at least one fact that was previously not present must be involved. Thus, an added fact only leads to further derivations if it is actually new, which also guarantees termination in the context of recursive rules.

2.1.2. Counting

Aside from DRed, the *counting* algorithm [4] provides another approach for incremental materialization maintenance. The basic idea here is that we count the number of current derivations for each fact. In this context, the explicit addition of a fact based on an update also denotes a derivation. Initially, if a fact is not present, it has a count of zero. Adding a fact, due to a new derivation, increases its count. Conversely, if a derivation of a fact is not given anymore because of some new operation, the fact's count is decreased. Hence, the count directly states whether a fact can be derived (count > 0) or not (count $= 0$), where only the latter case requires an actual deletion of the fact.

The algorithm's input consists of one update and a materialized dataset where the derivation count for each fact is already computed. For some materialization $\mathsf{mat}(P, E)$, the latter can generally be achieved by updating an empty dataset with $U = (E, \emptyset)$. The actual processing starts with the explicit introduction of addition and deletion operations defined by the provided update. Based on this, we first determine directly following operations, and count how often they appear. After that, we check if operations can be canceled by updating the count for each fact F. In detail, the count is increased by the number of new $add(F)$ operations and, simultaneously, decreased by the number of new $del(F)$ operations. If the adapted count is greater than zero, we remove the deletions and keep one $add(F)$, else we remove the additions and keep one $del(F)$, respectively. Finally, we execute every new operation and repeat the procedure as long as fresh operations are introduced.

Generally, the counting approach only works for non-recursive programs, because recursive derivations enable a fact to increase its own count, thus potentially impeding its deletion. A recursive variation of counting is provided by Motik et al. [6]. Here, the materialization is represented by a sequence of multisets (called trace) that reflects the semi-naive computation of the derivations. In this regard, a fact may occur in various multisets, each time with a different count, if it is derived several times during the semi-naive processing. Specifically, this means that a fact cannot falsely influence its own count due to recursive derivations.

3. Realizing Fully Dynamic Updates

In the presence of fully dynamic updates, the general procedure of incremental materialization approaches like DRed and counting changes. As stated in Algorithm 1, we now constantly check if a new update is available, and potentially introduce related operations, each time before performing any computation. On the one hand, this means that the set of facts on which an update is currently processed can be modified due to new incoming updates. Since these changes might affect already processed facts, this can impede the semi-naive evaluation, where we typically assume that a fact should only be further processed if it is not yet present. On the other hand, we concurrently deal with several updates introduced at different times, such that a strict order or separation of specific computation steps cannot always be guaranteed. This affects especially DRed where the processing of every update goes through the sequential overdeletion, rederivation, and insertion phases. Accordingly, the rederivation or insertion phase of one update might overlap with the overdeletion phase of another one introduced later, in particular since we assume a fair alternation between operations of different updates. In contrast to this, the counting approach is not directly affected by fully dynamic updates, since it does not rely on a specific rule separation or ordering. However, in the recursive variation, a new update might modify a multiset in the trace that has already been adapted by a previous update still being processed.

Despite the changes involved in the processing of fully dynamic updates, performing DRed and counting should still result in a correctly updated materialization. Therefore, we next take a look at the different errors that might occur during incremental materialization maintenance and how DRed and counting can deal with those in the context of fully dynamic updates.

3.1. Handling Erroneous Processing

Let $\hat{U} = \langle U_1, \ldots, U_n \rangle$, with $n \geq 1$, be a sequence of updates that transforms a materialization $M = \mathsf{mat}(P, E_0)$ into a correctly updated materialization $M' = \mathsf{mat}(P, E_n)$. Note that $\hat{U}$ does not need to be final, i.e., we may still extend the sequence by further updates $U_{n+1}, \ldots, U_{n+i}$ with $i \geq 1$. In this case, the expected, correctly updated materialization is defined as $M' = \mathsf{mat}(P, E_{n+i})$, always referring to the latest update U_{n+i} in the current sequence available so far. Working with fully dynamic updates should not affect the computation of M'. But due to the possible overlapping of updates, their operations can influence each other such that we might end up with a different, wrong materialization M^*. Since the correct materialization M' is unique, this requires that M^* contains additional facts not in M', or that M^* misses facts present in M'. Having additional facts means we either performed some invalid additions or omitted some deletions, whereas missing some facts is the result of either invalid deletions or omitted additions, respectively.

In order to avoid such incorrect behavior, we therefore analyze in the following each of those four cases and check (i) their relation to fully dynamic updates, (ii) whether DRed and the counting approach can handle those cases in the presence of fully dynamic updates, and (iii) how we can adapt DRed and counting to enable fully dynamic materialization maintenance.

3.1.1. Invalid Addition

We say that an addition $add(A)$ is invalid if the fact A does not appear in the correctly updated materialization M'. Since we can assume that the explicit introduction of an addition directly related to an update is always valid, an invalid addition can only result from the implicit propagation of some addition. Suppose $add(A)$ was determined by using the rule instance $r\sigma = F_1, \ldots, F_k \rightarrow A$ (based on $F_i = B_i\sigma$ and $A = H\sigma$) together with $add(F_j)$ for some $1 \leq j \leq k$. Then, A not being present in M' requires that at least one fact F_i, with $1 \leq i \leq k$, is also not available in M' anymore, otherwise we could validly derive the fact A. This means there has to be some deletion $del(F_i)$ that removes F_i. However, $del(F_i)$ could be applied together with $r\sigma$ to determine $del(A)$ and thus delete the fact A. Accordingly, A can only be present if this deletion is canceled as well. For that, we need an addition $add(F_i)$, but as stated before, F_i is not present in M'. Hence, this addition is invalid too. In this regard, an invalid addition appears if another invalid addition prevents a deletion affecting the former.

In general, the derivation of some fact begins with an explicitly determined operation based on an update. But since such an operation is always valid, it cannot lead to an invalid addition. However, an exception is possible for recursive derivations, because they can start with an implicitly introduced fact that derives itself. Hence, an invalid addition, for which there is no deletion that could cancel it, occurs when a fact recursively (re-)adds itself and thus prevents its own deletion.

Example 1 (Invalid addition)**.** *Consider the following rules $A \rightarrow B$, $B \rightarrow C$, and $C \rightarrow A$. Assume we apply an update $U_1 = (\{A\}, \emptyset)$ to an empty dataset. With the new A, we can derive a new fact B, which furthermore allows us to derive a fact C. Now, suppose we simultaneously introduce another update $U_2 = (\emptyset, \{A\})$ which deletes the fact A. Accordingly, A must not be* *present in the updated materialization. However, so far the derived fact C has not been processed, which means that we can still apply the rule $C \rightarrow A$ leading to the invalid addition of A.*

Although the example ends with an invalid addition, other outcomes are possible as well, depending on how the operations are further processed after the second update:
1) *Deletion overtakes addition*: here, the explicit deletion of A directly causes the deletion of B, further leading to the deletion of C, thus preventing the invalid addition of A. In this case, we obtain the correct materialization where none of the facts are present anymore.
2) *Addition overtakes deletion*: using C to (re)derive a new A before the deletion of A is propagated to B cancels the deletion completely. Hence, we end up with a wrong materialization where A, B, and C are still present despite the deletion.
3) *No overtaking*: due to the deletion of A, we have to delete the derived B, but processing C also allows us to derive another A. Since the original A has already been deleted, this derived A is considered to be a new, not already present, fact. Thus, we can now use the deleted B to further delete C, but likewise also use the new A to derive a new B. For the deleted C and the new B, we can then repeat the same procedure, hence ending up in an endless cycle where we constantly delete and re-add facts without termination.

In DRed, the separation of overdeletion and rederivation usually prevents the simultaneous processing of deletions and additions needed to produce invalid additions. Because of the overlapping processing of fully dynamic updates, this strict separation does not hold anymore. Moreover, both DRed and the recursive variation of counting handle recursive derivations based on the semi-naive evaluation, where a fact is only processed if it is not already present. However, fully dynamic updates allow for cases where we already delete facts in the recursion before the processing of this recursive derivation is finished. As a consequence, we might not be able to detect the end/beginning of the recursion anymore, such that the semi-naive evaluation is no longer able to guarantee termination.

Considering a potential solution for DRed and counting to deal with invalid additions, we have to make sure that the propagation of some addition $add(A)$ does not enable the fact A to recursively re-add itself, even if A is not present anymore. Therefore, the addition process has to stop at the latest when the start of the recursion, i.e., the fact A that initiated the recursive derivation, is reached. Stopping the addition then enables the deletion to catch up and ensure termination, as described in the first situation discussed for Example 1. Concretely, this could be realized by remembering the fact A (or the related $add(A)$) that started the recursion. The remembered fact is then passed on to its derived facts that also take part in the recursion. As soon as we reach the remembered fact again, i.e., determine another $add(A)$, we stop the related processing.

Still, only detecting the end of a recursion is generally not enough. If a fact C involved in a recursion has an alternative derivation, then re-adding the fact A that started the recursion is actually valid. Accordingly, a fact B that was deleted in the recursion because of deleting A has to be re-added as well. In DRed, the explicit rederivation phase already deals with this problem. The counting approach, on the other hand, requires additional measures.

3.1.2. Invalid Deletion

As a supplement to invalid additions, we say that a deletion $del(A)$ is invalid if the fact A has to be present in the correctly updated materialization M'. This means that A has an alternative derivation (including an explicit introduction) that is not affected by any deletion and still enables the addition of A in M'. Identifying the impact of fully dynamic updates in this context requires distinguishing between the two cases where the alternative addition $add(A)$ was determined either a) before or b) after introducing the deletion $del(A)$. In the first case a), we delete the fact A although it has an alternative derivation. This issue appears even if updates are not fully dynamic. Regarding case b), introducing the alternative $add(A)$ after the deletion enables us to re-add the fact A and cancel $del(A)$. Therefore, A can only not be present if another invalid deletion $del(A)$ deletes it. Hence, we also end up in case a) where we remove a fact despite the existence of an alternative derivation. Still, a special situation occurs if the introduction of the alternative $del(A)$ results from a recursive derivation, as illustrated in the following example.

Example 2 (Invalid deletion)**.** *Suppose we extend the rules from Example 1 with a new rule $D \rightarrow A$. Moreover, assume that we start with a materialized dataset $M = \{A, B, C\}$. Given the following update $U = (\{D\}, \{A\})$, we add a new fact D and simultaneously delete the fact A. While the deletion of A causes the deletion of B, adding D allows us to rederive A. With the deletion of B, we can furthermore delete C. However, due to the rule $C \rightarrow A$, this might cause the invalid deletion of the re-added A.*

In this respect, we have the same problem as for invalid additions, which may be solved likewise by stopping the propagation of a deletion as soon as the end of the recursion is reached. Nevertheless, we also have to check if the earlier described general case a), where a fact is deleted despite the existence of an alternative derivation, can still be handled by the discussed approaches if fully dynamic updates are given. The counting approach does not have any issue, since the utilization of a fact's count directly prevents (invalid) deletions if alternative derivations are given. For DRed, invalid deletions may be seen as the eponymous part of the overdeletion phase, being resolved in the subsequent rederivation phase. However, an important aspect here is that the rederivation does not start before the overdeletion is completed to ensure that a rederivation is indeed not affected by any deletion. With fully dynamic updates, the problem is that we cannot clearly state anymore at which point the overdeletion phase is done, due to a possibly endless sequence of incoming updates. A potential solution for this can be to restrict the propagation of operations to older facts, which are given in a finite number. In detail, some deletion $del(A)$ is only used together with some fact F if F has been introduced before the deletion was executed, otherwise F could not have been applied together with the deleted fact A for some derivation.

3.1.3. Omitted Addition and Deletion

So far, we discussed cases where we obtain an incorrect materialization due to the execution of operations that in fact should have been avoided. Now we take a look at the opposite, where problems happen as a result of *not* performing certain operations. Here, we assume that this omission of operations is not the consequence of passively forgetting to perform an operation, but rather actively caused by either canceling an operation before it is executed, or removing preconditions needed to determine the operation. Formally, omitting an addition $add(A)$ requires either some deletion $del(A)$ that cancels $add(A)$, or some deletion $del(F_j)$ with $1 \leq j \leq k$, which cancels $add(F_j)$ or deletes F_j, such that a related rule instance $F_1, \ldots, F_k \rightarrow A$ cannot be used to determine $add(A)$. Likewise, omitting a deletion $del(A)$ is caused either by some $add(A)$, or by some $add(F_j)$ that prevents determining $del(A)$.

One aspect to be aware of in this context is that compared to invalid operations, the omission of operations by itself is not necessarily a problem. Indeed, it may even be beneficial by avoiding redundant computations. For instance, it does not make sense to further compute derivations based on some fact A if A has already been deleted. Likewise, re-adding a fact A prevents the need to further propagate a previous deletion of A to its derived facts. Nevertheless, ending up with a wrong materialization is possible as well. Concretely, a needed fact might be missing in the updated materialization if we omit an addition, while omitting a deletion can lead to wrong, additional facts that are actually not derivable in the correctly updated materialization M'.

Assume again that we have a rule instance $F_1, \ldots, F_k \rightarrow A$ enabling us to determine an addition $add(A)$. Furthermore, let $del(A)$ and $del(F_j)$, with $1 \leq j \leq k$, be deletions that cause the omission of $add(A)$. If $del(A)$ and $del(F_j)$ are valid, then A is neither present in the correctly updated materialization M' nor can it be derived due to the missing F_j. Hence, omitting the addition of A does not produce an error. On the contrary, if $del(A)$ and $del(F_j)$ are invalid, then A should be in M' and can even be derived due to F_j also being in M'. In this case, omitting the addition of A leads to a wrong materialization. The same argumentation can be made for omitting a deletion based on an addition. In this regard, taking care of invalid operations as discussed earlier also solves the problem of omitting operations leading to a wrongly updated materialization. However, this is not the only issue caused by omitting operations.

Usually, both additions and deletions are processed until no further changes in the current set of facts are possible, i.e., if we add a fact A, we do not stop until every (direct) derivation of A has been computed, and if we delete A, we make sure to propagate the deletion until only facts with alternative derivations are left, respectively. In contrast to this, fully dynamic updates enable the cancelation of operations before they are entirely processed, because of the direct integration of new incoming updates after each computation. Accordingly, the problem of omitting operations actually refers to the incomplete processing of operations. In other words, we cannot be sure anymore if a certain operation was applied to introduce an operation or a fact, or if it has been canceled before due to a new update. Therefore, only using the available inference rules to detect derivation relations does not suffice.

Example 3 (Deletion without derivation)**.** *Consider the following rules $A \rightarrow B$, $A \rightarrow C$, and $D \rightarrow C$. Assume we apply an update $U_1 = (\{A, D\}, \emptyset)$ to an empty dataset. Then, we can use A to derive B, and D to derive C, respectively. Now, suppose we simultaneously introduce another update $U_2 = (\emptyset, \{A\})$. Because of the rule $A \rightarrow C$, we could technically propagate this deletion of A to C. However, A was actually not used to derive C. Hence, the deletion of A should not influence C. In contrast to this, the fact B should be affected by the deletion of A, but only using the provided rules does not allow us to determine whether a fact was really involved in the derivation of some fact or not.*

Example 4 (Duplicate derivation)**.** *Consider the rules $A \rightarrow B$ and $A \rightarrow C$. Assume we apply an update $U_1 = (\emptyset, \{A\})$ to a materialized dataset $M = \{A, B, C\}$, where we already used A to derive B and C. The update deletes A, which also triggers the deletion of B. Now, suppose we simultaneously introduce another update $U_2 = (\{A\}, \emptyset)$. This allows us to rederive the previously deleted B. However, this new A may also be used to derive another C, even though C was not removed by the deletion. Accordingly, the derivation $A \rightarrow C$ would be applied twice here.*

In this regard, omitting additions can lead to false deletions not related to truly conducted derivations, whereas omitting deletions might produce redundant duplicate facts. Since the false deletion of a fact C can simply be repaired by re-adding C based on the derivation that was indeed applied to add C in the first place ($D \rightarrow C$ in Example 3), and duplicates are not relevant if we work with sets of facts, DRed can directly handle omitted operations without the need for adaptations. For the counting approach, the impact is more severe, because omitting operations means that the count associated with a fact only represents the number of actually computed derivations, not the entire number of available rule instances that could be used to directly derive the fact. Therefore, solely relying on the inference rules to detect derivation relations does generally not work. For instance, in Example 3 the fact C was only derived by D, thus having a count of value one. But letting the deletion of A affect C would decrease the counter to zero, thus deleting C, despite its alternative derivation based on D. A solution here can be to remember which rule instances were really applied, in order to ensure that an operation is only propagated over derivations that were in fact performed.

4. Conclusion

We examined fully dynamic updates for incremental materialization maintenance, where new updates are directly integrated into the current processing. For that, we analyzed the problems that may occur due to the overlapping processing of updates and how the well-known DRed and counting algorithms can handle those. In detail, fully dynamic updates impede the correct processing of recursive derivations, resulting in invalid additions and deletions, or even non-termination. This is solved by detecting and memorizing recursions, as well as identifying when a fact needs to be rederived. Moreover, fully dynamic updates cause the omission of not yet completely processed operations, potentially leading to incorrect deletions or duplicate facts. While this does not affect the correctness of DRed, the counting approach has to know if a derivation was really computed or not.

Future work may include the examination of other materialization maintenance algorithms, like Backward/Forward [13] or the combined approaches presented by Hu et al. [14]. Moreover, the proposed adaptations for DRed and counting still require further investigation to allow for concrete implementations that guarantee both correctness and efficiency. In this context, we may also explicitly consider the parallelization of operations. Besides, it would be useful to also consider the setting of Datalog with negation, and we would like to investigate how one can still answer queries on materializations under the influence of fully dynamic updates, where temporarily blocking the integration of updates might serve as a solution.

References

[1] S. Abiteboul, R. Hull, V. Vianu, Foundations of Databases, Addison-Wesley, 1995. URL: http://webdam.inria.fr/Alice/.

[2] B. Motik, Y. Nenov, R. Piro, I. Horrocks, D. Olteanu, Parallel Materialisation of Datalog Programs in Centralised, Main-Memory RDF Systems, in: Proceedings of the AAAI Conference on Artificial Intelligence, volume 28, AAAI Press, 2014, pp. 129–137. URL: https://doi.org/10.1609/aaai.v28i1.8730.

[3] D. Carral, L. González, P. Koopmann, From Horn-SRIQ to Datalog: A Data-Independent Transformation That Preserves Assertion Entailment, in: Proceedings of the AAAI Conference on Artificial Intelligence, volume 33, AAAI Press, 2019, pp. 2736–2743. URL: https://doi.org/10.1609/aaai.v33i01.33012736.

[4] A. Gupta, I. S. Mumick, V. S. Subrahmanian, Maintaining Views Incrementally, in: Proceedings of the 1993 ACM SIGMOD International Conference on Management of Data, ACM Press, 1993, pp. 157–166. URL: https://doi.org/10.1145/170035.170066.

[5] D. Dell'Aglio, E. D. Valle, F. van Harmelen, A. Bernstein, Stream reasoning: A survey and outlook, Data Science 1 (2017) 59–83. URL: https://doi.org/10.3233/DS-170006.

[6] B. Motik, Y. Nenov, R. Piro, I. Horrocks, Maintenance of Datalog Materialisations Revisited, Artificial Intelligence 269 (2019) 76–136. URL: https://doi.org/10.1016/j.artint.2018.12.004.

[7] J. Urbani, A. Margara, C. J. H. Jacobs, F. van Harmelen, H. E. Bal, DynamiTE: Parallel Materialization of Dynamic RDF Data, in: The Semantic Web - ISWC 2013, volume 8218 of *Lecture Notes in Computer Science*, Springer, 2013, pp. 657–672. URL: https://doi.org/10.1007/978-3-642-41335-3_41.

[8] D. F. Barbieri, D. Braga, S. Ceri, E. D. Valle, M. Grossniklaus, Incremental Reasoning on Streams and Rich Background Knowledge, in: The Semantic Web: Research and Applications, volume 6088 of *Lecture Notes in Computer Science*, Springer, 2010, pp. 1–15. URL: https://doi.org/10.1007/978-3-642-13486-9_1.

[9] D. Dell'Aglio, E. D. Valle, Incremental Reasoning on RDF Streams, in: Linked Data Management, Chapman and Hall/CRC, 2014, pp. 413–435. URL: http://dellaglio.org/preprints/14b1.pdf.

[10] Y. Ren, J. Z. Pan, Optimising ontology stream reasoning with truth maintenance system, in: Proceedings of the 20th ACM International Conference on Information and Knowledge Management, ACM Press, 2011, pp. 831–836. URL: https://doi.org/10.1145/2063576.2063696.

[11] M. Kaminski, Y. Nenov, B. Cuenca Grau, Datalog rewritability of Disjunctive Datalog programs and non-Horn ontologies, Artificial Intelligence 236 (2016) 90–118. URL: https://www.sciencedirect.com/science/article/pii/S0004370216300297.

[12] F. Bancilhon, Naive Evaluation of Recursively Defined Relations, in: On Knowledge Base Management Systems: Integrating Artificial Intelligence and Database Technologies, Topics in Information Systems, Springer, 1986, pp. 165–178. URL: https://doi.org/10.1007/978-1-4612-4980-1_17.

[13] B. Motik, Y. Nenov, R. E. F. Piro, I. Horrocks, Incremental Update of Datalog Materialisation: the Backward/Forward Algorithm, in: Proceedings of the AAAI Conference on Artificial Intelligence, volume 29, AAAI Press, 2015, pp. 1560–1568. URL: https://doi.org/10.1609/aaai.v29i1.9409.

[14] P. Hu, B. Motik, I. Horrocks, Optimised Maintenance of Datalog Materialisations, in: Proceedings of the AAAI Conference on Artificial Intelligence, volume 32, AAAI Press, 2018, pp. 1871–1879. URL: https://doi.org/10.1609/aaai.v32i1.11554.